Enabling Real-Time Mobile Cloud Computing through Emerging Technologies

Tolga Soyata
University of Rochester, USA

A volume in the Advances in Wireless Technologies and Telecommunication (AWTT) Book Series

Information Science REFERENCE
An Imprint of IGI Global

Managing Director: Lindsay Johnston
Managing Editor: Austin DeMarco
Director of Intellectual Property & Contracts: Jan Travers
Acquisitions Editor: Kayla Wolfe
Production Editor: Christina Henning
Development Editor: Brandon Carbaugh
Cover Design: Jason Mull

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA, USA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Enabling real-time mobile cloud computing through emerging technologies / Tolga Soyata, editor.
pages cm
Includes bibliographical references and index.
ISBN 978-1-4666-8662-5 (hc) -- ISBN 978-1-4666-8663-2 (eISBN) 1. Cloud computing. 2. Mobile computing. I. Soyata, Tolga, 1967-
QA76.585.E55 2015
004.67'82--dc23

2015015533

This book is published in the IGI Global book series Advances in Wireless Technologies and Telecommunication (AWTT) (ISSN: 2327-3305; eISSN: 2327-3313)

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 8
Hardware and Software Aspects of VM-Based Mobile-Cloud Offloading

Yang Song
University of Rochester, USA

Haoliang Wang
George Mason University, USA

Tolga Soyata
University of Rochester, USA

ABSTRACT

To allow mobile devices to support resource intensive applications beyond their capabilities, mobile-cloud offloading is introduced to extend the resources of mobile devices by leveraging cloud resources. In this chapter, we will survey the state-of-the-art in VM-based mobile-cloud offloading techniques including their software and architectural aspects in detail. For the software aspects, we will provide the current improvements to different layers of various virtualization systems, particularly focusing on mobile-cloud offloading. Approaches at different offloading granularities will be reviewed and their advantages and disadvantages will be discussed. For the architectural support aspects of the virtualization, three platforms including Intel x86, ARM and NVidia GPUs will be reviewed in terms of their special architectural designs to accommodate virtualization and VM-based offloading.

INTRODUCTION

In the past decade, significant technological advances in the semiconductor technology have dramatically improved the computational and storage capability of handheld mobile devices such as smart phones and tablets. This enabled mobile devices not only to access a vast amount information instantaneously through fast communications networks, but also to perform ever more sophisticated computational tasks such as face and speech recognition, object detection and natural language processing (NLP) pervasively

DOI: 10.4018/978-1-4666-8662-5.ch008

(Wang, Liu, & Soyata, 2014). However, the performance and user experience of these resource-intensive mobile augmented-reality applications are still constrained by the relatively low performance CPU and GPU, as well as limited memory and flash storage of the mobile devices. These resource constraints cannot be easily improved due to the relative size and battery life limitations of mobile devices, as compared to mainstream desktop PCs. Therefore, many applications, which are both latency-sensitive and compute-intensive, such as real-time face recognition, are still beyond the capabilities of today's smartphones and tablets.

To overcome these resource limitations and extend the capabilities of mobile devices to the point, where they can run these resource-intensive applications, mobile-cloud computing (MCC) was introduced to leverage the cloud resources. MCC enables mobile devices to utilize powerful cloud servers to store and access a vast amount of data and process compute-intensive tasks. Mobile-cloud computing has been intensively investigated as an integration of cloud computing into the mobile environment. Utilizing cloud servers for storage is easy and there have already been many popular applications providing data backup and sharing features between the users and the cloud. Unlike storage, utilizing cloud servers for computation acceleration is not trivial. Computation offloading is a solution to alleviate resource limitations on the mobile devices and provide more capabilities for these devices by migrating partial or full computations (code, status and data) to more resourceful computers. The rapid advancement of wireless network connectivity and architectural advancements in mobile devices in recent years have made computation offloading feasible. Currently, offloading computation from mobile devices to cloud servers faces several challenges which is what most of the research in this field focuses on. These challenges are summarized below:

- **What to Offload:** The entire program cannot be offloaded for remote execution. Before offloading, the program needs to be partitioned in one of three ways: 1) manually by the programmer, 2) automatically by the compiler, or 3) at runtime. Manual partitioning will put the burden on the programmers but will potentially result in lower overhead and more flexibility. On the contrary, the automated partitioning can perform offloading on an unmodified program which is more convenient for users, but might result in a higher performance overhead. Different strategies like code tagging and dynamic prediction based on profiling can be applied to increase the performance.
- **When to Offload:** Applications may have different requirements on performance and mobile devices may have different capabilities and energy limitations. Offloading decisions need to be made based on multiple criteria, such as 1) improving performance when the remaining energy is abundant, 2) energy savings when the remaining energy is low, and 3) network conditions at runtime. These decisions can be made by statically and/or dynamically by profiling, which has an impact on the execution overhead.
- **How to Offload:** Emerging cloud computing technologies, combined with virtualization technologies provide a powerful, flexible, manageable and a secure platform for offloading. This attracted a large body of research on VM (Virtual Machine)-based offloading approaches, which study offloading at different granularities such as OS-level, application/thread-level and method-level.

To address these challenges, new schemes have been introduced to achieve seamless offloading between mobile devices and cloud servers. One of the effective approaches is the VM-based approach, where we migrate either an OS-level VM or an application-level VM to remote cloud servers, execute the compute-intensive task there and get the response back to the mobile device through either mes-

sages or joining/migrating the thread/VM. To enable VM-based offloading, which significantly differs from traditional mobile cloud computing, we need support from both the software layer (both on mobile devices and the cloud servers) and the cloud hardware layer.

The rest of this chapter is organized as follows: We will start with an introduction to general mobile cloud offloading, VM-based approaches, and the design and implementation challenges in using virtualization technologies to augment mobile-cloud computing by leveraging cloud resources. In the next section, we will provide a detailed discussion of the software support of virtualization. Software aspects of various VM-based mobile cloud offloading systems and frameworks, including OS-level VMs and application-level VMs will reviewed in detail. Architectural support for virtualization will be elaborated on in the following section, where the underlying hardware designs supporting and accelerating virtualization are reviewed. This hardware support section includes specific architectural designs by Intel x86 and ARM CPUs and NVidia GPUs for both desktop computers and mobile devices. We will conclude this chapter and provide pointers to potential future research directions in the final section.

MOBILE CLOUD OFFLOADING

The breakneck pace of the advancement of smartphone technology has turned these devices from mere "phones" to devices that are indispensible in everyday life. Sensory capabilities of mobile devices made an impressive progress with the incorporation of cameras, temperature, humidity, and acceleration sensors among others. Furthermore, a wide range of networking options like USB, WiFi, IR, and 4G made these devices ever more connected with increasing connection speeds. While networking and sensory capabilities of mobile devices are a lot less sensitive to the progress in VLSI technology, the same is not true for their computational and storage capabilities: Every new generation of VLSI (e.g., 32nm vs. 22nm) is expected to improve the performance-per-Watt metric of computational devices, whether mobile or desktop, as prescribed by the Moore's Law.

Since the technological advancement in battery energy densities (i.e., Joules per kg) is significantly slower than the VLSI technology-based energy efficiency of CPUs (i.e., performance per Watt), and the sizes of mobile devices cannot be increased beyond a certain point to improve their battery energy storage capability, the performance-per-Watt metric is expected to dominate the overall performance of mobile devices in the foreseeable future. To continue improving the performance of mobile devices, a possible approach that comes to mind is to take advantage of the improving networking capability of mobile devices by sending the entire or part of the computational task to a mainframe, like a cloud host, and receiving the result via the same network. This method is called *Mobile Cloud Offloading*, in which a mobile device accesses a host machine in cloud to perform all or part of a required computationally-intensive task. Several mechanisms are introduced in the past decades to implement this scheme, among which virtualization stands out as a viable alternative due to its advantages in utilizing and sharing computational resources. Below is a list of considerations when offloading tasks from mobile devices into the cloud.

- **Utilization:** One primary concern for cloud service providers is being able to predict the workload in the cloud at a given point in time. If the mobile tasks that are being initiated by different mobile devices were mapped to pre-determined servers in the cloud, a cloud service provider would have no way to smoothly distribute the workload for optimum efficiency. In other words, while some

devices stayed idle, others would be overloaded. Virtualization makes it possible to pool scattered tasks into several servers and leave others idle. This not only allows the cloud operator to run multiple servers at optimum load for a maximum utilization ratio (e.g., 80%) without overloading them, it also allows them to turn off certain servers for maximum power efficiency. These servers could be turned on later when the cumulative load of the datacenter increases, based on an optimum predictive allocation algorithm. When additional servers come online, the workload from other servers could be migrated to them to prepare for additional future demand.

- **Compatibility:** Mobile devices, like smartphones, are of various types running different operating systems (e.g., Linux, Android, iOS). Directly running mobile applications on cloud servers would be severely restrict the device-server compatibility. This would require cloud servers that support pre-determined Operating Systems with no possibility to run applications from mobile devices with different OSs. Virtualization eliminates this compatibility issue by allowing a server to run multiple Operating Systems. Compatibility also refers to the ability to be compatible with different server hardware. To exacerbate the compatibility issue, a typical datacenter contains servers that have many different server models with different hardware. Since virtual machines run on virtual hardware, issues that are introduced by hardware incompatibility are avoided. Furthermore, when datacenter servers are upgraded to better ones, the applications that are running on them can be migrated to these new servers seamlessly.
- **Isolation:** Virtualization permits the isolation of two separate applications that are sharing the same physical resources, thereby preventing any compromise of data and/or code security. Additionally, new tasks can be launched without disturbing tasks that are currently executing. New resources or system extensions can be added without modifying or suspending current resources. Not only are virtual machines are protected from each other, but also an attack on one of the virtual machines can be easily confined by the hypervisor, without spreading to another (or multiple other) VMs.
- **Ease of Deployment:** Virtual machines allow easy and fast installation of new service applications or user machines. Creation of a virtual machine is extremely simple, as it typically just requires copying an existing image and migrating it to a selected server. On the contrary, a non-VM setup of a new system requires purchasing new equipment, new hardware deployment, and OS and application software configuration, which is more time consuming and prevents rapid deployment.

VIRTUALIZATION APPROACHES

The development of modern mobile devices, such as smartphones and tablets, has become the enabling technology for pervasive or ubiquitous computing. While non-resource-intensive applications run on these mobile platforms without a problem, a set of resource-intensive applications, such as augmented reality applications including real-time face recognition, natural language processing, do not achieve satisfactory performance levels due to their computational limitations. These limitations are not expected to improve in the foreseeable future (Satyanarayanan, Bahl, Caceres, & Nigel, 2009) due to the continuous user demand for higher performance and the slow progress in battery technology, placing limitations on the power consumption of the primary computing elements on the mobile device, such as CPU and GPU.

To enable a mobile device to run such resource-intensive applications, the solution is mobile-cloud computing, where the compute-intensive parts of an application are offloaded to a remote server. These remote servers amass much higher computation capability, significantly higher flash storage and local

memory and high bandwidth, which imply that, an offloaded mobile application can be substantially accelerated when it can utilize such cloud resources. Furthermore, since the bulk of the execution of the application is being traded-off against potentially higher network traffic to transfer the application data (and application code), energy savings on the mobile device can be achieved when the ratios of network time vs. computation time are in favorable proportions.

Web services provided through XML-RPC and RESTful are commonly used for mobile applications. However, achieving energy savings and/or acceleration when offloading general-purpose computations is not trivial. Several approaches have been proposed to achieve efficient offloading from mobile devices to cloud servers. They can be categorized into two types based on whether the offloading is done transparently or non-transparently.

- **Non-Transparent Offloading:** Requires the developer to re-design the application with explicit mobile-cloud models to benefit from remote servers. To be able to offload, application-specific code has to be properly deployed on the remote server, which may require the user to have full control over the server.
- **Transparent Offloading:** Requires no modification on existing mobile applications. Programmers develop the application as if all of the code is running locally (or at most tag some functions that may be suitable for offloading using directives) and the underlying runtime will automatically transform most of the mobile applications to benefit from seamless offloading to the remote cloud servers.

Virtualization or virtual machine techniques play a key role not only in building cloud computing infrastructures but also in supporting both types of the aforementioned offloading approaches. For the *transparent offloading* approach, it is clear that an application-level virtual machine is essential for the runtime to dynamically analyze and profile the code, make offloading decisions, and partition and transparently migrate the code to the remote servers. For the *non-transparent offloading* approach, code deployment on cloud servers is not trivial due to the co-existence of various platforms, operating systems and libraries. Virtualization is able to hide the low-level details from the executing applications and provide a uniform environment to each application.

Due to the high latency through the internet which significantly hurts the offloading performance, the edge-server or *cloudlet* idea has been proposed as an accelerator in public areas like coffee shops to provide one-hop offloading services to the mobile devices (Soyata T., Muraleedharan, Funai, Kwon, & Heinzelman, 2012; Satyanarayanan, Bahl, Caceres, & Nigel, 2009). To provide offloading services as part of a public network infrastructure, the inherent sharing characteristics of virtualization can be utilized, combined with the ability of isolation and self-management on the cloudlet. To deploy the code, users of the mobile device need to have full control over a remote server. So, for purposes of maintenance, security, and privacy, user actions should be properly limited and isolated. To achieve this goal, a VM-based architecture is the inevitable choice.

Virtualization can be achieved at different levels from the hardware level to the application level. Based on these different virtualization levels, common VM-based approaches can be categorized into OS-level and application-level categories.

- **OS-Level VMs:** Provide better flexibility and ability for the customization of the virtualization environment. However, due to its large overhead, there are challenges in a real deployment using this approach. The large overhead of OS-level VMs are due to the mechanism of deployment which requires one has to transfer the entire VM image over the network before the VM can start. This weakens the acceleration benefit and hurts application responsiveness due to the latency incurred during the transmission of the VM image.
- **Application-Level VM:** Is a process running on top of the operating system of the remote server. It is relatively light-weight and the cost to migrate is relatively low compared to the OS-level VMs. The application, however, has limited flexibility in customizing the VM. For example, an application may not be able to use libraries written in other languages, which may translate to an eventual performance overhead.

VM-BASED OFFLOADING

Virtual Machines have become increasingly more popular in modern cloud computing, since they introduce a new way to run multiple operating systems on a single machine via the decoupling of physical resources. VM-based offloading is one of the most common approaches for offloading computational tasks to cloud server nodes. Compared to a traditional server management framework, the usage of VMs significantly improves certain operational aspects of cloud computing: 1) It enables rational and economical resource partitioning among users, 2) increases resource utilization at the datacenter, 3) lowers the power consumption at the datacenter, 4) it gives administrators higher flexibility for process deployment, and 5) lowers the application programming burden of the programmers.

On the other hand, the virtualization framework for application offloading introduces challenges such as 1) runtime management and data transfer overhead, 2) higher demand for design optimization, 3) new security concerns. Virtualization is usually implemented via the usage of hypervisors. We will now characterize the features of hypervisors and list the strengths and challenges of VM-based offloading.

Hypervisor Layer

Virtualized environments are the foundation of most cloud computing infrastructures. They are usually implemented via the use of a Hypervisor, which is generally a software layer that lies between the Virtual Machines and the physical hardware. Hypervisors can be divided into two major categories (Popek & Goldberg, 1974) .

Type 1 Hypervisors: are directly installed on the physical hardware, and therefore do not require a host operating system (OS) and can have direct access to the underlying physical hardware. QEMU, Xen and Microsoft Hyper-V are Type 1 hypervisors. Kernel-based Virtual Machine (KVM), on the other hand, makes its host OS a Type 1 hypervisor.

Type 2 Hypervisors: are installed above a host OS, and run within the environment provided by the OS. Therefore, the host OS would have direct access to the underlying hardware and is responsible for hardware source and service management. The hypervisor serves as the second layer and simulates virtual machine environments. The widely-adopted VMware Workstation and VirtualBox are two typical Type 2 hypervisors.

We will provide a brief introduction to the three mainstream open-source hypervisors: QEMU, Xen and KVM in terms of their Hypervisor mechanism for virtualization.

- **QEMU:** In its broadest sense, QEMU is a generic hardware emulator. It can be used standalone to create a virtual machine environment, but more often QEMU is executed under Xen or KVM to support device virtualization to the guest. In this case, QEMU provides simulation for peripherals including PCI Bridge, VGA card, mouse/keyboard, hard disk, CD-ROM, network adapters, sound card, etc. (QEMU), using dynamic translation between virtual and physical devices.
- **XEN:** Xen is an open-source Type-1 hypervisor running directly on hardware and is fully responsible for the resource management of the host machine. It utilizes para-virtualization, which permits it to achieve a near-native performance. Since its publication in 2003, Xen has been widely adopted as the basis of many commercial or open-source applications and is in use in the largest cloud environments today (Xen). The most distinguished feature of Xen is that it has a specialized VM with special privileges, named The Control Domain (or Dom0). Dom0 is a customized Linux kernel that can handle resource and I/O access directly, and exposes device control to guest VMs via emulators.
- **KVM:** In the open-source hypervisor projects, the Kernel-based Virtual Machine, or KVM, is a relatively new product which was first introduced in 2006. Soon after its introduction in February 2007, KVM was merged into the Linux kernel (version 2.6.20). KVM provides a complete virtualization environment in which virtual machines appear as normal Linux processes and integrate seamlessly with the rest of the system (Kivity, Kamay, Laor, Lublin, & Liguori, 2007). After this first integration, KVM became the main virtualization package in mainstream Linux OS (e.g. Ubuntu, Fedora).

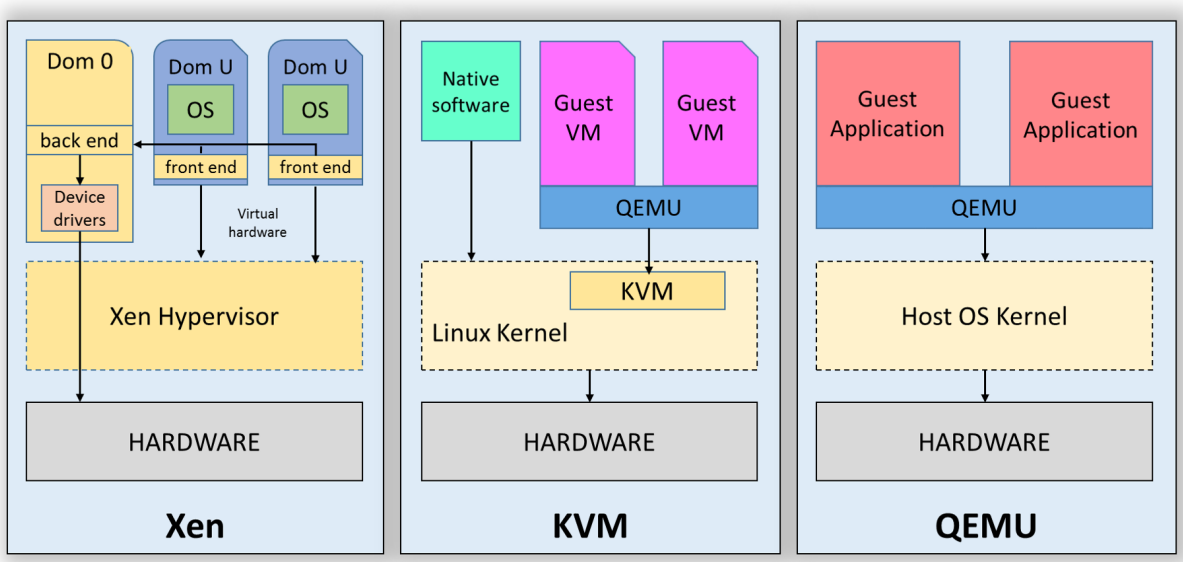
KVM requires CPU virtualization extensions (Intel VT or AMD-V) and is used together with QEMU. It consists of a loadable kernel module (*kvm.ko*) and a processor-specific module (either *kvm-intel.ko* or *kvm-amd.ko*) at its core. QEMU is also used to provide virtualization for peripherals such as hard disk, CD-ROM and network adapters. KVM is a Type 2 hypervisor while its usage makes the host Linux OS a Type 1 hypervisor. This structure of KVM makes it very different from Xen and QEMU, as shown on Figure 1.

Hypervisor Features

A hypervisor connects the guest VM to the host machine by simulating a near-host environment. To accomplish this task, a hypervisor should incorporate a set of services. We present a brief overview of these services below.

- **CPU Virtualization:** For a completely virtualized CPU, there are a set of requirements that must be met (Popek & Goldberg, 1974). For a CPU architecture that allows virtualization, neither sensitive instructions nor privileged instructions must also be treated as privileged instructions. Since virtual CPUs must behave like real CPUs for more than one VM, the Hypervisor translates and schedules virtual CPU instructions from different guests to the physical CPUs appropriately. This is achieved by allowing a specific guest to have exclusive use of a CPU for a period of time, after which this guest is interrupted and the exclusive use is passed onto another guest. During this

Figure 1. Comparison of Xen, KVM, and QEMU.



switch, the CPU state of the first guest is saved, and the state of the next guest is loaded before the control is passed onto it. This process is repeated to provide fair and even access to every guest (Chisnall, 2008). This is called Symmetric Multiprocessing (SMP).

- **Memory Virtualization:** Modern CPUs (e.g., x86 and ARM) include a Memory Management Unit (MMU) which performs virtual-to-physical memory address translation. This translation is accelerated by the CPU hardware by providing traps for privileged instructions to map the virtual addresses to corresponding physical addresses. However, Guest VMs cannot directly access the MMU, which would imply that the hypervisor would lose control of the VMs. The hypervisor provides MMU functionality to the guests by utilizing a soft MMU, which utilizes shadow page tables to accomplish its task. Every access to an actual page table invoked by a VM is intercepted by the soft MMU and is replaced by an access to a corresponding shadow page table.
- **Interrupt/Timer Virtualization:** A hypervisor should be able to virtualize and manage the interrupt/timer, the interrupt/timer controller of the guest OS, as well as the access of the guest OS to the controller. At the same time, it must virtualize all interrupts/timers going into the guest OS. Both of these would cause considerable overhead. Modern CPUs also provide hardware extensions for Interrupt/Timer virtualization. For example, ARM defines a Generic Interrupt Controller (GIC) architecture and Generic Timer Architecture. Recent versions of GIC and Generic Timer introduce virtualization extensions that help manage the virtual interrupts/timer from the hypervisor, thereby substantially reducing the overhead caused by the interrupts/timer virtualization. This means that the hypervisor can directly send virtual interrupt/timer without translation (Dall & Nieh, 2014).

- **I/O Virtualization:** I/O virtualization includes two parts of drivers: front-end and back-end. A hypervisor emulates I/O devices by a device emulator running in the host OS or directly on the hardware. KVM and Xen employ QEMU's simulators which have full access to hardware devices by default to implement back-end simulation. On the other hand, front-end drivers are also needed in guest VMs to simulate the usual I/O requests sent by the guest OS. Hypervisor is responsible for the communication between front-end and back-end.

Nevertheless, the case for graphics cards (GPUs) is a lot more complicated. A GPU keeps its own graphics memory, works independently from the motherboard, and involves large data transfers. Meanwhile, most GPUs do not provide state saving/recovery. These features of GPUs make GPU-virtualization a lot more challenging in many aspects, like VM state-switching and multiprocessing management (Chisnall, 2008).

CHALLENGES IN VIRTUALIZATION

Despite many of its advantages, virtualization is not without challenges. We will summarize the primary challenges facing virtualization in this section.

- **Overhead of Deployment and Management:** In Mobile-Cloud computing, offloading can be done via either a VM clone or a VM migration. Additionally, an interactive status transfer is needed for VM management. Both of these require an additional computation and introduce a resource overhead. VM deployment introduces computational resources in several aspects (Shiraz, Abolfazli, Sanaei, & Gani, 2013) including VM creation, VM configuration and VM startup, as well as application deployment.
- **Security Problem:** The security vulnerability of the hypervisor is another challenge that deserves attention. Since hypervisors serve as a new software layer between the guest OS and the physical machine or the host OS, any attack on the hypervisor implies a risk of attack directly to the CPU, memory, I/O, or a combination of these resources. Furthermore, since cloud computing involves big data management, networking and remote control, any vulnerability of the hypervisor translates to service disruptions, data confidentiality breaches, reputation fate-sharing among other issues.
- **GPU Virtualization:** Nowadays, GPUs are receiving widespread adoption in a range of platforms ranging from individual computers to supercomputers. GPU-accelerated computing offers high performance computing (HPC) at a large scale due to its cost-effective, and power-efficient features (e.g., the mobile-cloud face recognition application). However, the use of GPU-acceleration is still at a relatively low level, as compared to CPU-only implementations. Sharing and managing GPU resources in a hypervisor faces several big challenges:

First challenge is to determine how to share GPU resources among several VMs. Historically, most GPUs keep memory outside of OS's main framework, and they do not allow access to multiple concurrent applications. Furthermore, they are unable to save and restore the state of applications. This makes GPU virtualization for cloud computing complex and fragile. As of the preparation of this book chapter, only

a few advanced GPUs, like the Nvidia Kepler family, support virtualization to some degree. The new Maxwell family from Nvidia is expected to address this issue much better and full GPU Virtualization is expected to be accomplished within the next upcoming Nvidia generations.

Second challenge is the virtualization overhead, as encountered in CPU virtualization. Although this overhead is unavoidable, some work on GPU virtualization, such as gVirtuS, allows transparent access to the GPU and is independent from the hypervisor, with only a slight overhead introduced relative to actual GPU implementations (Giunta, Montella, Agrillo, & Coviello, 2010). This project is the joint product of the University of Napoli Parthenope and the Open Source Lab initiative and can be accessed at http://osl.uniparthenope.it/projects/gvirtus/.

SOFTWARE SUPPORT FOR VIRTUALIZATION

To leverage virtualization technologies for fast and transparent mobile-cloud offloading, additional modifications need to be made to the traditional virtualization software platforms. Two virtualization approaches are introduced to enable mobile devices to seamlessly offload computation tasks to remote cloud servers. The first one is at the OS-level, which migrates a customized full-fledged virtual machine from the mobile device to the server. The VM image is customized by the developers to accelerate their application on mobile devices. The communication between the mobile and the cloud and the offloading procedure are explicitly defined by application developers. The second approach is at the thread-level, where the underlying application VM (e.g. Java VM) is extended with the ability to profile the program and migrate threads to remote servers. Program partitioning and offloading procedures are transparent to developers and users. The two approaches will be discussed in detail in the following sections.

OS-Level VM

As mentioned in the previous sections, OS-level VMs provide the developers with the flexibility in customizing the VM and offering specific acceleration services to their applications. Additionally VMs offer an isolated environment and good manageability on the remote servers which enables the deployment of self-managing cloudlets. However, one of the greatest challenges with OS-Level VMs in mobile-cloud computing is the high latency to transfer the necessary states to the remote server. In order to provision a customized VM in the cloud, disk images and a VM snapshot (which is typically several gigabytes in total), have to be completely migrated from the mobile device before the VM can be resumed to provide offloading services for the mobile. This migration will take at least five minutes even on a fast 802.11n network at peak network throughput. This time delay is clearly unacceptable for mobile users, especially considering that, one of the advantages of virtualization is application performance acceleration.

To achieve the goal of both high performance and manageability, the VM-based Kimberley architecture was proposed (Satyanarayanan, Bahl, Caceres, & Nigel, 2009) to accelerate the VM migration process. A cloudlet, defined as a self-managed datacenter in a box, was introduced in Kimberley. The cloudlet is able to support a few users at a time and maintains only the soft state, thereby making a loss of connection acceptable. When a mobile client connects to the cloudlet, it notifies the Kimberley Control Manager (KCM) on the cloudlet to download a small VM overlay from either the Internet or the mobile client. A VM overlay is the difference between the memory snapshot and disk of a base VM and the customized VM. Several base VMs are pre-installed on the cloudlet. Application developers

will choose one of the base VMs, build their customized VM on top of this base and generate the VM overlay after their customization is complete. When the VM overlay is delivered, a technique called dynamic VM synthesis applies the overlay to the base VM and launches the target VM. Since the size of the overlay is usually much less than the size of the customized VM, the latency of the synthesis is greatly reduced. After the computation is complete, the KCM can simply shutdown the VM and free the resources, providing self-manageability that only needs minimal maintenance.

The Kimberly system was implemented on a Nokia N810 tablet running Maemo 4.0, and the cloudlet infrastructure was implemented on a desktop computer running Ubuntu Linux where VirtualBox was used to provide the VM support. System performance was evaluated by considering the size of VM overlays and the speed of the synthesis operation. According to experimental results, the size of the generated VM overlays is around 100-200 MB for a collection of Linux applications, which is an order of magnitude smaller than a full VM image that can be as large as 8 GB. The processing time for VM synthesis ranged from 60 to 90 seconds and has plenty of potential room for improvement through further optimizations such as parallelized compression and decompression and VM overlay pre-fetching. (Ha, Pillai, Richter, Abe, & Satyanarayanan, 2013) showed that the latency of dynamic VM synthesis can be further optimized by pipelining the transmission and deduplication of the VM overlay. There will be redundant data within and between the memory snapshots and disk images since most of the data in the memory is originally loaded from the disk. The size of overlay can be significantly reduced by exploiting the redundancy and find the minimal set of data that is necessary to build the customized VM launch. The tradeoff between the size of the overlay and the computation complexity can provide significant performance improvement by carefully choosing the delta algorithm and the granularity of the chunks. Additionally, since there are strong boundaries enforced by the hypervisor between the guest and the host systems, it is difficult for the outside layer to accurately interpret higher level abstractions inside a VM, which is called a *semantic gap*. Such a gap prevents a further reduction of the size of the overlay. In (Ha, Pillai, Richter, Abe, & Satyanarayanan, 2013), authors bridge the gap for disks by exploiting the TRIM support and introspecting the file system inside the VM. For memory, since there is no standard (like TRIM for disks), bridging the gap requires the modification of the guest OS and significant effort on maintaining and tracking the memory structure changes.

Another technique used in the paper is pipelining the transmission of VM overlays and *Early Start*. To keep both the network and the CPU busy at the same time to reduce the latency, it is straightforward to divide the large overlay into small chunks and pipeline the chunk transmission and processing so that the transmission latency, as well as computational latency can be partially coalesced. Furthermore, observing that the VM does not really need all of the chunks to be transferred before it can be resumed, we can order the chunk transmission in a way where the earliest needed chunks are transferred first. This way, the VM can be resumed even when a small portion of the overlay is transferred. In the paper, authors use static proofing to obtain the order of the accessed chunks and use on-demand transmissions for the out-of-order chunks. Utilizing all of the previously mentioned techniques, authors evaluated the fully optimized VM synthesis in terms of *first response time* using five software packages. The results show that, except for AR application, the first response time for all other four applications come within ten seconds. They also compare the VM synthesis to the remote installation approach and the fully optimized VM synthesis approach has significant performance advantages in every application except AR.

Application-Level VM

As some VM-based frameworks involve OS-level VM cloning and complete application offloading, others focus on *encapsulating* applications from its operating system. This may exploit application partitioning in which only parts of an application are offloaded. In other words, instead of the physical hardware, the applications themselves are virtualized. This is called an application-level VM. Although this framework has limited usage since every application cannot be virtualized this way, the resource burden is significantly reduced for applications that can take advantage of this form of virtualization.

Application partitioning schemes fall into two main categories: static and dynamic (Chun & Maniatis, 2010). In earlier cloud environments, static partitioning was typically used due to its easier design and reduced resource management burden. However, static partitioning does not provide any optimization for a diverse cloud environment and workload patterns. A dynamic partition algorithm is preferred, since an optimized partial execution between mobile devices and the cloud is not only determined by the application itself but also the mobile platform capability, cloud environment, network speed and specific instantaneous workloads in the cloud.

A typical application-level VM is implemented using software dynamic translation (SDT) (Scott, et al., 2003). The VM keeps sets of byte-code instructions which are physical hardware and operating system independent. Since the translation needs to provide runtime monitoring and function appendix by code modification, based on existing application code, SDT can modify the existing byte-code, injecting additional code and control the code execution. The VM framework lies between applications and the cloud host operating system. The VM operates by decoding, translating and storing the applications' instructions initially. On the host machine, the VM then takes control of application execution by capturing a snapshot and synchronizing the current state, including counters, pointers, PC, condition, registers etc. Instructions for which a context switch is needed are processed next. Applying dynamic translation and offloading, the VM operations are flexible and modular and diverse forms of offloading can be implemented.

CloneCloud (Chun, Ihm, Maniatis, Naik, & Patti, 2011) is an example application-level VM framework which exploits dynamic application partitioning and partition offloading. Mobile-cloud computing under CloneCloud is performed in several steps:

1. Application partitioning is automatically performed according to a partitioning algorithm. The mechanism aims at find a fixed execution point, upon which application is migrated between the mobile devices and the cloud. The algorithm optimizes the partitioning ratio considering network properties (not the network condition), computing capacity of the mobile and the cloud, as well as the estimated energy consumption. Not all of the execution points are valid as there are many constraints for execution availability at every given point. To ensure that a given partitioning is legal, a Static Analyzer is used to identify all possible partitions.
2. The states (e.g. counters, registers, memory, etc.) of the mobile platform and the cloud platform are timely synchronized, during which execution process is suspended.
3. CloneCloud migrates application operations at thread-level, in which way multi-threading is allowed. With the source byte-code transferred, application can perform distributed execution and virtualized computation can be conducted seamlessly.
4. The results of clone execution in the cloud are finally re-integrated back to the mobile platform. When a thread reaches its re-integration point, execution is suspended. It is then packed and merged into the original process.

Networking Virtualization

For a mobile device user performing computation in the cloud via a wireless network and a cloud service, network virtualization is one of the essential and fundamental elements. For a large-scale cloud computing system with large VM image collections and a large data center, networking is important in the following areas: accessing specific VM images, transmitting images between devices and servers, accessing the data center and migrating application offloading. Using traditional networking schema in cloud computing faces several limitations: scalability, flexibility and automatic management.

From the standpoint of the cloud service provider, network congestion is difficult to properly gauge. A cloud service system may start with tens of tenants but suddenly grow to hundreds. Without networking virtualization, physical devices, like routers, must be upgraded to meet incremental requirements. Traditional physical devices are not designed for cloud computing, so the entire networking system must be suspended and wait for device upgrades. Considering a similar case as above, when the cloud system grows and network requirements increase, it is difficult for cloud operators to use heterogeneous networking gear from different vendors, which makes the management and provisioning very difficult. This would result in high management costs, as well as wasted resources and increased overhead.

Using networking virtualization, networking is abstracted from the underlying physical hardware. Operators can manage networking aspects such as specific connection patterns, switching, routing, and security easily. Thus, networking can be organized as a high-level integration and automatically allocated, which helps economic use of resources and reduces energy consumption.

Network performance affects the overall performance of a cloud computing system significantly in terms of both execution speed and resource consumption. Although networking can be virtualized and applied just as in the case of the virtualization of other devices, networking service takes place at different levels in different protocols, depending on the cloud system model. These higher layer models include Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) (Dinh, Lee, Niyato, & Wang, 2013). Generally, IaaS provides more flexible protocol selection to vendors, while PaaS and SaaS usually provide provider-determined network service to customers (Amies, Sluiman, Tong, & Liu, 2012).

When virtual machines are created, IP Addresses are initially allocated. IP Addresses can be generated by the system, reserved as provisioned for VMs, and by using VLANs. Using Internet Protocol Version 6 (IPv6), networks can be created with more available IP addresses and higher security levels. Inside the virtual machines, hypervisors can share a single physical network interface with multiple virtual machines. Hypervisors usually provide virtual networking in three ways: bridging, routing and Network Address Translation (NAT). Under bridge mode, a hypervisor serves as a data transfer interface and the virtual machine is exposed to the Ethernet directly. In routing mode, the hypervisor goes into the network layer and makes the virtual network interface externally visible at the IP level.

Large-scale clouds can emulate more IP addresses than is otherwise physically available by hiding the network of virtual machines from the external network. In this case, NAT is needed as it enables communication with the internet using a hidden virtual machine address. NAT assigns virtual IP addresses, or private local IP addresses which are different from the host IP. As the virtual IP addresses are invisible on the Internet, NAT could create a massive amount of internally-accessible IP addresses for purposes of serving a large numbers of virtual machines. At runtime, NAT software keeps a routing table and changes the IP address information in the data packets based on the table, in which way hypervisors can forward incoming and outgoing data packages.

ARCHITECTURAL SUPPORT FOR VIRTUALIZATION

The x86 and ARM architecture were never designed for virtualization. With the increasing demand for cloud computing inside enterprise data centers, where virtualization has become a standard practice, pure software-based virtualization, without explicit native hardware support, suffered serious performance penalties. Hardware vendors like Intel and AMD have responded to the demand for virtualization with new processor extensions including Intel VT-x and AMD-V. These hardware-assisted virtualization techniques reduce the performance overhead of the traditional approaches such as binary translation and no longer require changes inside the guest operating system. In the following subsections, the architectural support for virtualization in x86, ARM and Nvidia GPU will be discussed in detail.

x86 Virtualization Support

As Popek and Goldberg stated in (Popek & Goldberg, 1974), for a virtualizable CPU architecture, any instruction that is control-sensitive and related to resource configuration must be privileged. However, x86 microprocessor architecture has such features that makes it unable to meet this demand, therefore making it very challenging to support full virtualization on an x86 architecture.

Challenges with X86 Virtualization

In order to implement security in accessing resources, a modern x86 architecture, for example, IA32 (Intel Architecture, 32-bit), provides an instruction segregation mechanism, in which a direct access to pivotal functionality, such as CPU control and memory access, are privileged. To achieve this, x86 CPUs provide four privilege levels, 0, 1, 2 and 3, from most privileged (Ring 0) to least privileged (Ring 3). This model is usually described as a ring structure, named Ring 0 to Ring 3. In practice, Ring 1 and Ring 2 are rarely used by the operating system developers, since for most cases the protection mechanism only has a concept of privileged and unprivileged instructions and the benefits to Ring 1 and 2 are negligible. Therefore, we will just talk about the other two levels in this section: Ring 0, where kernel components of the OS run, and Ring 3, where most user applications run.

The IA32 architecture includes 16 instructions that run in Protected mode which cannot be accessed by user applications. If any code that is running in a Ring greater than 0 attempts to execute one of these instructions, a Protection Fault exception is generated. The list of privileged instructions includes LGDT, LLDT, LTR, LIDT, MOV (control register), MOV (debug register), LMSW, CLTS, INVD, WBINVD, INVLPG, HLT, RDMSR, WRMSR, RDPMC and RDTSC (INTEL-IA-PartGuide, 2010). These instructions are mainly related to loading/writing-to registers that control CPU operation, control/debug, cache state, TLB, model-specific, timestamp etc. For a fully virtualized CPU, the guest OS must be able to run some its components in Ring 0 (highest privileged level). However, hypervisor must have privileged control and occupy Ring 0, and it cannot allow its guest OSs such control. The only solution for the hypervisor is to run the guest OSs in less privileged Ring 1, 2, or 3, which is called ring de-privileging.

The primary task of the hypervisor is to have its guest OS function just like in an non-virtualized CPU environment. However, ring de-privileging introduces plenty of challenges in regard to this requirement (Neiger, Santoni, Leung, Rodgers, & Uhlig, 2006). For example, some registers contained information related to CPU control, which can only be written in Ring 0 but can be read in higher Rings. A guest OS that needs to write to one of these registers could end up not being able to function seamlessly, as in an

non-virtualized environment. Furthermore, there are some instructions that operate on segmented memory. If the guest OS executes one of these instructions, hypervisor may not be able to properly rearrange the memory mapping in a virtualized way. To find a solution to these challenges, much effort is made in terms of both software and hardware alternatives. As far as software solutions, there are two alternatives:

- **Binary Translation:** Binary translation is made popular by VMware and is widely used in its products. The general idea of binary translation is that, the hypervisor scans the instruction stream from the guest and re-encodes the privileged instructions into a virtual version. One disadvantage of this approach is the performance penalty for scanning and encoding, especially for I/O intensive applications. On the other hand, some special software like debuggers that require setting breakpoints, makes hypervisor design extremely complicated. This is due to the binary translation changing the actual code, and even the order of the breakpoints and instructions, which makes the debugging process extremely challenging.
- **Paravirtualization:** Unlike binary translation, paravirtualization goes to the root of the problem: the guest operating system. For paravirtualization, the guest OS is modified in a way that privileged instructions are replaced by hypercalls. Thus, the guest OS communicates with the hypervisor via hypercalls and avoids the aforementioned troublesome instructions. However, since the hypervisor is required to handle these interrupts with an extra layer, a performance penalty is introduced.

Despite these software-only solutions, INTEL has made significant effort to facilitate / accelerate virtualization via hardware support, which is called *hardware-assisted virtualization*. Hardware-assisted virtualization can be conceptualized as allowing hypervisors to run in a Ring "-1" which would free the precious Ring 0 for guest VMs. We will talk about the details of Hardware Assisted Virtualization below.

Intel Hardware-Assisted Virtualization

Intel's virtualization extensions for the 32-bit x86 architecture, VT-x, was first introduced in 2005. VT-x refers to a new mode added to the processor, named virtual-machine extensions (VMX) which support virtualization for multiple virtual machines. VMX includes two new CPU operations: VMX root operation and VMX non-root operation. Employing VMX mode, generally a hypervisor runs in VMX root operation and its guest operating systems runs in non-root operation.

- **VMX Root Operation:** Is much the same as an ordinary processor operation from the hypervisor's view when it is operating as a host. What is different is that, it allows a series of VMX instructions. On the other hand, virtual processors for VMs, running in VMX non-root operation, are modified in certain ways to support virtualization. Certain instructions are trapped by the CPU, instead of directly executing on the CPU. This causes transitions between the hypervisor and the VMs, which are called *VM exits*. With VM exits, the access boundary of the guest OSs is limited, therefore the host hypervisor can retain control of the CPU resources. Similarly, when transitioning from hypervisor operation to guest operation, a *VM entry* occurs.
- **VMX Non-Root Operation:** To manage the VMX non-root operation, as well as the two transition operations, namely VM entries and VM exits, VT-x introduces a new data structure called virtual-machine control structure (VMCS). VMCS contains a guest-state area for VMX non-root

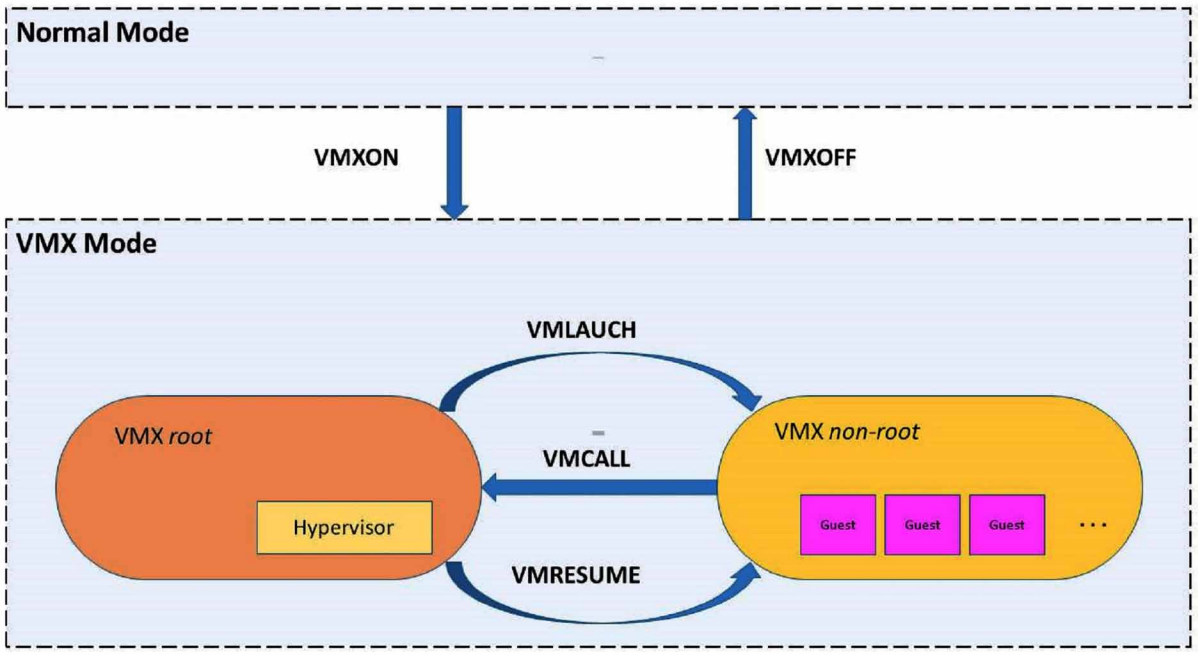
operation, and a host-state area for VMX root operation. Upon each transition, VM exits and entries save the current processor state into the corresponding state area in VMCS, and load the next required processor state. VMX mode is implemented by a series of instructions extensions. Table 1 shows a list of the VMX instructions.

VMX operations incorporate all necessary instructions to complete a full lifecycle of a guest virtual machine, by allowing the guest to enter and exit and the host to manage the guest OS and stay in control. Figure 2 shows the lifecycle of VMX operations.

Table 1. Additional instructions introduced by INTEL's VT-x.

Instruction	Description
VMXON	Enter VMX operation
VMCLEAR	Inactive VMCS
VMPTRLD	load the VMCS pointer for an active VM
VMWRITE	Initial/write fields in the current VMCS
VMLAUNCH	Create a VMCS launching
VMCALL	Exit from VMX non-root operation
VMREAD	Read fields in the current VMCS
VMRESUME	Resume VM execution in VMX non-root operation
VMPTRST	Store the pointer to an active VM to memory
VMXOFF	Leave VMX operation

Figure 2. Operation transitions in Intel VT-x.



- **VMXON:** The primary entry point to VMX mode is the VMXON instruction. After executing VMXON, the processor would be placed in VMX_ROOT mode.
- **VMCLEAR:** To ensure that the VMCS region is in a pure state before activated, VMCLEAR must be executed before VMPTRLD. VMCLEAR will allocate a new VMCS region in memory and set its state to "clear." Also, the previous VMCS pointer will be invalidated.
- **VMPTRLD:** Executing VMPTRLD initializes a pointer to a new allocated VMCS region for each guest virtual machine.
- **VMWRITE:** After VMPTRLD, the hypervisor will issue a sequence of VMWRITE instructions to create several memory regions in the VMCS. These regions include host-state fields, guest-state fields, VM-exit control fields, VM-entry control fields, and VM-execution control fields.
- **VMLAUNCH:** After the above procedures, VMCS is successfully initialized and is ready to use. The hypervisor will then launch the new created virtual machine by applying VMLAUNCH. Simultaneously, VMCS state will be changed to "launched."
- **VMCALL:** To implement the *VM exit* functionality that is previously mentioned, the VMCALL instruction is used. If the software wants to request a service from host processor, it will VMCALL that service. The hypervisor will implement a VMCALL through one of the hardware-assisted traps.
- **VMREAD:** VMREAD is used to access specific VMCS fields. For example, if a VM exit occurs, the hypervisor uses VMREAD to access the exit-reason field in the VMCS. Depending on different exit reasons, hypervisor might want to access other fields in VMCS.
- **VMRESUME:** VMRESUME can be used to resume a VM execution or launch a guest on the same virtual processor in a "launched" VMCS. For example, after an exception, the hypervisor can resume the state of a virtual machine by VMRESUME.
- **VMXOFF:** If a hypervisor wants to shut down and leave VMX mode, it executes VMXOFF.

In summary, VT-x provides a solution to some of the challenges that exist in pure-software virtualization, by introducing assistance directly from the x86 hardware. Using VT-x, guest operating systems can work at *Ring 0 equivalent* privilege levels. Also, a paravirtualization system can make some use of the hardware-assisted virtualization features to optimize certain operations. For example, a hypervisor can benefit from performing I/O through hardware virtualization extensions rather than software emulated I/O which has higher overhead and is therefore less efficient.

AMD Hardware-Assisted Virtualization

Similar to Intel's VT-x, AMD also provided a set of instruction extensions to assist virtualization, called AMD-v, with the code name "Pacific." The list of the added instructions are shown in Table 2 (AMD64-Virtualization, 2005). The virtual mode of AMD-v is named SVM. AMD-v also provides a mechanism for mode switching and transitioning between the hypervisor to the guest through VMRUN and #VMEXIT instructions. The AMD instruction #VMEXIT is similar to the VT-x instruction VMXOFF which facilitates a VM exit. The guest OS uses VMMCALL which causes the processor to generate an exception and the guest OS requests services from the hypervisor via the use of this instruction, much like the VMCALL in the Intel implementation. AMD-v also introduces a data structure similar to VMCS, called Virtual machine control block (VMCB). A VMCB is also maintained for each guest virtual machine, and contains a set of Control areas and State areas. Control areas contain control and information determine the source of #VMEXIT. The state of virtual processors is stored in the State areas.

Table 2. Additional Instructions introduced by AMD-V.

Instruction	Description
CLGI	Clear Global Interrupt Flag to 0
INVLPGA	Invalidate selective TLB mapping
MOV(CRn)	Move between general registers and control registers
SKINIT	Secure Init, allowing activating of trust software
STGI	Set Global Interrupt Flag to 1
VMLOAD	Load processor state from control block (VMCB)
VMMCALL	Call hypervisor
VMRUN	Run virtual machine
VMSAVE	Save the state of a VM into control block (VMCB)

In addition to these features similar to VT-x, AMD-v provides several extra features related to the x86-64 architecture. Some AMD CPUs contain an integrated memory controller, which allow hypervisors to handle memory management. To achieve memory management, AMD-v includes Shadow Page and Nest Paging mechanisms. Shadow Page mechanism allows a hypervisor to modify the OS's page table and remap memory partitions. Nest Paging allows two levels of memory address translation performed in hardware. Nest Paging keeps a Nest Paging Table to translate guest physical memory addresses to host physical addresses, so that the guest OS can fully control and use its own page tables. Since the translation is done by hardware, it can achieve a near-native performance.

AMD-V also introduces a special Device Exclusion Vector (DEV) interface. Each DEV is kept in an exclusive protection zone. DEV takes charge of upstream accesses for their permission and limit the address zones for devices. This mechanism allows it to protect memory mapped I/O (MMIO) and DRAM from abuse.

ARM Virtualization Support

CPU virtualization for the ARM architecture is a relatively new field of research with slow growth compared to the x86 architecture. ARM brings additional challenges in virtualization (Hwang, et al., 2008) compared to the x86 architecture. In this section, ARM virtualization will be described briefly.

Challenges with ARM Virtualization

As previously mentioned, a typical x86 architecture introduces a four-level privilege system which introduces challenges for virtualization. The case is even worse for ARM which has only one privileged mode and one unprivileged mode. Such a limited scheme forces the guest OS and the applications to run in the same unprivileged mode, and makes the protection of the guest OS more difficult. In the ARM architecture, cache is virtually tagged and therefore there is no ASID attached to the TLB. This feature results in a very high flushing frequency on state switches if we try to distribute memory among guest OSs and applications. To provide support for easier virtualization, starting from ARMv7 in 2010, a new virtualization extension (VE) was introduced as an optional feature on ARM CPUs. VE, along with the previously introduced extension Large Physical Addressing Extensions (LPAE), allow an efficient hardware-assisted implementation of a hypervisor possible in the ARM architecture.

ARM Virtualization Extensions

The principle of virtualization extensions on the ARM architecture is very similar to that of in the x86 architecture, which is, in short, the introduction of a new "-1" privilege ring which is even more privileged than the kernel mode. ARM has two working modes: secure mode and non-secure mode, and VE are only available under the non-secure mode. The newly introduced privileged level is called the *hyp mode*. Similar to the x86 virtual extension operation, certain additional instructions are introduced to facilitate hardware-assisted virtualization in the ARM architecture. For example the "hvc" instruction is used to enter the virtualized operation mode. Without an ASID for TLB, the hyp mode has a register named VMID which keeps a stable mapping to physical memory during state switches. This allows the ARM architecture to eliminate the problems with the aforementioned high flushing rates.

NVidia GPU Virtualization Support

Many compute-intensive applications and games heavily rely on the acceleration attained from Graphics Processing Units (GPUs). Many of the applications in this category are good candidates for a cloud computing environment. Although virtualization of such applications in the cloud would provide all of the previously mentioned benefits of isolation, resource flexibility and security, one of the major drawbacks of virtualization in a cloud computing environment is the lack of support for high-performance GPU virtualization. One primary hurdle for GPU virtualization is the fact that, unlike a CPU which is design to be shared by multiple processes, GPUs usually assumes no multiplexing. Also, the high memory bandwidth demands from a GPU will cause significant overhead in a virtualized environment. Recent advances in virtualization technologies have enabled virtual machines to directly access physical GPUs and exploit their hardware's acceleration using an I/O Pass-through technique. Meanwhile, GPU manufactures like NVidia have also equipped their products with virtualization support assisted by the GPU hardware, similar to the hardware-assisted virtualization that the CPU manufacturers introduced.

Recent advances in hardware have enabled virtualization systems to achieve one-to-one mapping between an I/O device and a VM instance. This allows VMs to use non-virtualization-friendly I/O devices without software emulation (e.g., Network Interface Cards (NIC) and GPUs). One problem with these I/O devices is the use of the traditional Direct Memory Access (DMA) mechanism which will violate the memory isolation enforced by the hypervisor among VMs. A new configurable I/O Memory Management Unit (IOMMU) provided by Intel VT-d and AMD-Vi allows the hypervisor to reconfigure the interrupts and DMA of the physical devices in a way where they are directly mapped into certain guest VMs and the DMA requests will pass through the hypervisor, incurring less overhead while preserving the isolation.

Even with the ability to map one device to one VM, it is still difficult to virtualize devices like GPUs. For these virtualization-unfriendly GPU devices, the hypervisor needs to translate commands from VMs so that it appears to the GPU as if the commands are coming from a single system. Even with such a mapping, multiple GPUs need to be physically installed in a server so that the hypervisor can achieve the mapping, which is not flexible or cost-efficient. To address the problem, NVidia introduced the GRID vGPU technology (Nvidia-Grid-VGPU) in the Kepler architecture to make GPUs more virtualization-friendly. This technology enables multiple VMs to share true GPU hardware acceleration without compromising the graphics experience. With the GRID vGPU technology, each VM has its dedicated memory in the GPU and the native graphics commands of each VM are passed directly onto the GPU, without any translations by the hypervisor.

CONCLUSION AND FUTURE WORK

In this chapter we discussed the state of art in VM-based mobile-cloud offloading techniques in detail both in terms of its software and architectural aspects. We introduced the general structure of multiple widely-adopted virtualization platforms (hypervisors), which are Xen, QEMU, and KVM. We documented the way each one of these hypervisors allow the virtualization of a variety of resources, such as CPU, memory, interrupt/timer, I/O, and network. We listed the challenges of running a hypervisor in a cloud computing environment, since each one of these resources have unique characteristics which cause a different form of a challenge.

We discussed how a guest OS (i.e., a Virtual Machine, or VM) runs CPU instructions as if there is no hypervisor. Kimberley and CloneCloud are the two VMs which run at the OS level and application level, respectively. We discussed these two popular VMs in detail which run without ever leaving the hypervisor's control. We discussed the current improvements to different layers of these two virtualization systems made especially for mobile-cloud offloading purposes. Also, two approaches at different offloading granularities, application and thread-level, were reviewed and their advantages and disadvantages were discussed.

While the virtualization of resources can be achieved using pure software approaches, this typically has a high performance penalty. In 2005 and going forward, almost every CPU manufacturer introduced a form of hardware assistance mechanism for the virtualization of various resources. This is through the introduction of instruction extensions that run on different privilege levels. We detailed the hardware assisted virtualization mechanisms that are introduced by Intel, AMD, ARM, and Nvidia. These platforms are Intel's VMX in the x86 architecture, AMD's AMD-v, ARM's VE (virtualization extensions), and Nvidia's Grid vGPU for GPU virtualization.

As mentioned in this chapter, there still exists challenges yet to be addressed to enable the practical use of VM-based mobile-cloud offloading in daily life, which opens the door to potential directions for future work. One of the challenges is the overhead of the deployment and management of a VM-based offloading system. Offloading via traditional VM clones or VM migration approaches may incur significant communication latency over the Internet and further investigations are required to reduce the transmission and processing overhead of VM-based coarse-grained offloading approaches. Another question that needs to be addressed is the security and privacy for both the offloading service providers and the users. It is important for the service providers that they can detect and prevent malicious code being offloaded to their systems. On the other hand, it is important for the users to assure that their offloaded code and data are secured against other parties including the service providers. Additionally, users also need to be able to verify that their offloaded code is actually executing as expected and is returning the correct results (i.e., functional verification), which requires further investigation in the future.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation grant CNS-1239423 and a gift from Nvidia corporation.

REFERENCES

Alling, A., Powers, N., & Soyata, T. (2015). Face Recognition: A Tutorial on Computational Aspects. In *Emerging Research Surrounding Power Consumption and Performance Issues in Utility Computing*. IGI Global.

AMD64-Virtualization. (2005). *Secure virtual machine architecture reference manual*. AMD Publication.

Amies, A., Sluiman, H., Tong, Q. G., & Liu, G. N. (2012). *Infrastructure as a Service Cloud Concepts*. Developing and Hosting Applications on the Cloud.

Chisnall, D. (2008). *The Definitive Guide to the Xen Hypervisor*. Pearson Education.

Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., & Patti, A. (2011). Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems* (pp. 301--314). ACM. doi:10.1145/1966445.1966473

Chun, B.-G., & Maniatis, P. (2010). Dynamically partitioning applications between weak devices and clouds. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing \& Services: Social Networks and Beyond* (p. 7). ACM. doi:10.1145/1810931.1810938

Dall, C., & Nieh, J. (2014). KVM/ARM: the design and implementation of the linux ARM hypervisor. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (pp. 333-348). ACM. doi:10.1145/2541940.2541946

Dinh, H. T., Lee, C., Niyato, D., & Wang, P. (2013). A survey of mobile cloud computing: architecture, applications, and approaches. In *Wireless communications and mobile computing* (pp. 1587–1611). Wiley Online Library. doi:10.1002/wcm.1203

Giunta, G., Montella, R., Agrillo, G., & Coviello, G. (2010). *A GPGPU transparent virtualization component for high performance computing clouds* (pp. 379–391). Springer-Verlag. doi:10.1007/978-3-642-15277-1_37

Ha, K., Pillai, P., Richter, W., Abe, Y., & Satyanarayanan, M. (2013). *Just-in-time provisioning for cyber foraging* (pp. 153–166). MobiSys.

Hwang, J.-Y., Suh, S.-B., Heo, S.-K., Park, C.-J., Ryu, J.-M., Park, S.-Y., & Kim, C.-R. (2008). Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE* (pp. 257--161). IEEE. doi:10.1109/ccnc08.2007.64

INTEL-IA-PartGuide. (2010). Intel® 64 and IA-32 Architectures Software Developer's Manual.

Kivity, A., Kamay, Y., Laor, D., Lublin, U., & Liguori, A. (2007). kvm: the Linux virtual machine monitor. *Proceedings of the Linux Symposiu*, pp. 225-230.

Neiger, G., Santoni, A., Leung, F., Rodgers, D., & Uhlig, R. (2006). *Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization*. Intel Technology Journal.

Nvidia-Grid-VGPU. (n.d.). Retrieved from VIRTUAL GPU TECHNOLOGY: http://www.nvidia.com/object/virtual-gpus.html

Popek, G. J., & Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. In *Communications of the ACM* (pp. 412–421). ACM. doi:10.1145/361011.361073

QEMU. (n.d.). *QEMU Emulator User Documentation*. Retrieved from http://qemu.weilnetz.de/qemu-doc.html

Ravi, V. T., Becchi, M., Agrawal, G., & Chakradhar, S. (2011). Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th international symposium on High performance distributed computing* (pp. 217--228). ACM. doi:10.1145/1996130.1996160

Satyanarayanan, M., Bahl, P., Caceres, R., & Nigel, D. (2009). The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Transactions on Pervasive Computing*, 14-23.

Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J. W., & Soffa, M. L. (2003). Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (pp. 36--47). IEEE. doi:10.1109/CGO.2003.1191531

Shiraz, M., Abolfazli, S., Sanaei, Z., & Gani, A. (2013). A study on virtual machine deployment for application outsourcing in mobile cloud computing. *The Journal of Supercomputing*, *63*(3), 946–964. doi:10.1007/s11227-012-0846-y

Soyata, T., Muraleedharan, R., Ames, S., Langdon, J., Funai, C., Kwon, M., & Heinzelman, W. (2012). COMBAT: Mobile Cloud-based cOmpute/coMmunications infrastructure for BATtlefield applications. [Baltimore, MD.]. *Proceedings of the Society for Photo-Instrumentation Engineers*, *8403*, 84030K–84030K, 84030K-13. doi:10.1117/12.919146

Soyata, T., Muraleedharan, R., Funai, C., Kwon, M., & Heinzelman, W. (2012). Cloud-Vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture. *Computers and Communications (ISCC), 2012 IEEE Symposium on*, 59-66.

Wang, H., Liu, W., & Soyata, T. (2014). Accessing Big Data in the Cloud Using Mobile Devices. In P. R. Dek (Ed.), *Handbook of Research on Cloud Infrastructures for Big Data Analytics* (pp. 444–470). Hershey, PA, USA: IGI Global; doi:10.4018/978-1-4666-5864-6.ch018

Xen. (n.d.). *Xen Project Software Overview*. Retrieved from Xen Wiki: http://wiki.xenproject.org/wiki/Xen_Overview#Documentation

KEY TERMS AND DEFINITIONS

AMD-v: AMD's hardware-assisted virtualization technology. AMD-V involves similar features for instructions extension as VT-x. Besides that, AMD-V also provide several modes that help hypervisor to handle memory-partition.

ARM: ARM Holdings is a publicly-traded company that licenses low power-consumption CPU architectures to companies such as Nvidia, Samsung, and many others. This allows the licensees to quickly develop products that require CPU cores. ARM is the dominant architecture for smart-phones and many other low-power devices.

Augmented Reality: A family of emerging applications that supplement computer-generated information with acquired real-time information to augment the information content. An example is an application that super-imposes a dress -from an existing database- on a person without having to actually wear that dress.

Central Processing Unit (CPU): This central piece of hardware controls the movement of the data from the main memory into its cores and executes a program that is written by the developer. It is possible that, multiple Operating Systems (OS) are running on a CPU (i.e., virtualized). Virtualization allows a seamless transition from one OS to another. This is done through hardware support that is built into the CPU hardware (e.g., the x86 architecture).

CloneCloud: A web-based system applied for mobile-cloud offloading. CloneCloud handles the communication and storage tasks in mobile-cloud system and can automatically do partition for a smartphone application and distributed the task between cloud servers and the smartphone via network, in which way help user save energy on smartphones and get better performance.

Cloudlet: An intermediate computationally capable device that has direct WiFi access to a mobile platform and WAN access to the cloud. A cloudlet can be used to perform numerous tasks, such as preprocessing the information received from the mobile device. This could ease the computational burden the mobile device, thereby improving its perceived performance, as well as power consumption.

CPU Cache Memory: Composed of L1, L2 and L3 cache layers (stylized L1$, L2$, and L3$), the purpose of CPU cache memory is to allow quick access to frequently used memory locations by buffering them within the cache hierarchy. The lower the cache memory level is, the smaller (but, faster) the cache is. For example, the Nehalem CPU has a 64KB L1$, which can be accessed in 4 cycles, however, L2$ is 256KB, while it requires 11 cycles for access. L3$ requires 50 cycles and is shared by all cores, however, it is 8MB. In some server CPUs, L4 is available.

CPU Main Memory: Typically in the Gigabyte (GB) range, CPU memory is the highest latency, but high throughput storage medium within the memory hierarchy of the CPU. Access to CPU memory is done through a row buffer, thereby making CPU memory not byte-addressable.

Crowd-Sourcing: Crowdsourcing is to outsource a task which is usually huge to a broad, loosely defined external group of people or devices that are willing to help. In mobile-cloud computing, crowdsourcing is to offload a heavy computation task to the nearby mobile devices through wireless network to accelerate the task or improve the quality of the result.

Face Recognition: A computationally-intensive process that associates the faces in a picture with a known set of faces that exist in a fixed database. Typical implementations consist of three steps for this process: Face Detection identifies the location of the faces in a picture, Projection converts these faces into coordinates in a different vector space called Eigenfaces, and the final Face Recognition step is the search for the closest match in the database.

GPU Main Memory: An example GPU memory type is GDDR5. The biggest difference of GPU memory as compared to CPU memory is its parallelism. GPU memory has 16 banks, and is capable of providing consequent memory locations to 16 threads in parallel. However, if some of the threads cannot make use of these consecutive locations, certain data elements will be wasted, requiring more cycles to feed data to those threads.

GPU Virtualization: Many I/O devices like GPUs are usually not design to be virtualization-friendly. GPU virtualization provides a way for multiple VMs to share a single GPU by adding PCI pass-through support and enabling GPUs to save registers and do context switching.

Graphics Processing Unit (GPU): This device is connected to the CPU through an I/O bus, such as PCIe (PCI Express). GPU code is responsible for explicit data transfers from CPU memory (main memory) and the GPU memory (Global Memory). The GPU code (composed of multiple "kernels") executes inside the GPU cores while using the data within the GPU's Global memory. GPU virtualization is a lot more challenging to implement than CPU virtualization.

gVirtuS: Referring to the GPU Virtualization Service. GVirtuS is an open source project that enables GPU virtualization by giving access of GPU to the virtual machines in a transparent way. GVirtuS currently can only runs on NVidia CUDA based GPUs but is going to be applied on other GPUs in the future. GVirtuS is usually used for remote GPU sharing and can get a relatively satisfactory performance.

Hypervisor: A software or hardware layer lies upon host machine or host OS. Hypervisors provide exclusive virtual runtime environments include CPU, memory and other resources for the virtual machines, and also manage their operation. A hypervior may directly run on the host hardware machine or within a host operating system.

I/O Virtualization: I/O virtualization is to consolidate multiple I/O devices into a single one which is shared by multiple VMs and is dynamically allocated to different entities to achieve better flexibility and overall utilization of the system.

IA32: Intel's third generation x86 architecture. In a broad sense, it also refers to all 32-bits x86 architecture versions (not only Intel's).

IaaS (Infrastructure as a Service): A type of cloud computing model. IaaS denotes the case that a service provider provides a whole physical computer infrastructure, or a virtual machine when applying virtualization technology to the users.

Kimberley: A Virtual machine that is designed to accelerate the virtualization of mobile-cloud application by introducing multiple optimizations for the transfer of the VM image. These optimizations such as VM overlays and VM synthesis aim at reducing the starting latency of a VM due to the long transfer delays required for transferring a VM image.

KVM: A Linux subsystem with full name Kernel-based Virtual Machine, also a type 2 hypervisor which provides virtualization extension to Linux kernel. KVM is only able to work on CPUs with hardware-assisted virtualization extension. When merging into the Linux kernel mainline, KVM turns the Linux operation system into a type 1 hypervisor.

Memory Virtualization: Memory virtualization is to create a distributed memory pool for a cluster by decoupling and gathering the memory from individual systems in the cluster. The pool, which overcomes the physical limitation of traditional memory, can be accessed and managed throughout the entire cluster.

Mobile-Cloud Offloading: Due to the hardware limitations on mobile devices, such as tablets and smart phones, it is feasible to run certain mobile applications in the cloud by outsourcing the application to cloud server. This concept, i.e., mobile-cloud offloading, also saves energy when the cost of communication (i.e., the transfer of data and code) amortizes the cost of computation (i.e., the energy required to execute the program by the CPU and/or GPU).

PaaS (Platform as a Service): As a kind of cloud computing model, PaaS denotes the case that a service provider provides a computation environment including OS, storage device and other servers as a software development platform to the users.

QEMU: A generic device emulator that performs virtualization for kinds of hardware. Working as an independent hypervisor, QEMU can either run a single program (user-mode) or a complete operating system. Besides, QEMU also serves in many other hypervisors, like Xen and KVM, as peripherals emulator.

SaaS (Software as a Service): As a kind of cloud computing model, PaaS denotes the case that a service provider provides a service upon an exact software application, for which users would get access to a certain application without concerning about its maintenance.

Thread: A CPU core could house more than one thread. For example, in the INTEL Nehalem CPU architecture, 4C/8T implies (4 cores, 8 threads). This means that, each core can execute two threads. The ability to execute multiple threads affords each core more "options" for execution. While these two threads share many resources in the core (e.g., fetch and decode units, as well as the L1$ and L2$), this doesn't necessarily hurt the performance, since core-intensive and memory-intensive threads could make a good pair, utilizing the resources much more efficiently than a single thread would.

Virtual Machine (VM): A software that simulates real physical hardware or operating system environment in which programs can execute like in a physical machine. A virtual machine may support the execution form a computing process, an application, a complete operating system to multiple guest operating systems.

VM Clone: A way of virtual machines creation. VM clone means make a copy of an existing virtual machine instead of reinstalling guest OS and/or applications. Creating a VM clone can either fully copy the mother virtual machine and make an independent clone, or just make a clone from snapshot and sharing virtual disk with the mother virtual machine.

VM Image: A copy of the entire state of a virtual machine. A capable virtual machine monitor is able to create and store its VM Images in certain formats (e.g. raw, qcow2, vmdk and vdi) via which the virtual machines can be restored to the same state afterwards.

VM Migration: A way of virtual machines creation. VM Migration, also called Live Migration, means moving a running virtual machine from one host to another with the same virtualized environment. VM migration is done by transferring the full state, including memory, network and other devices to the destination hardware.

VT-x: Intel's hardware-assisted virtualization technology. VT-x involves a set of architectural instructions extension for IA32 CPUs, including VMX root operations and VMX non-root operations. VT-x makes CPU virtualization much simpler in which way reducing the hypervisor complexity and software size.

x86: Introduced by INTEL Corporation in the 80's, x86 is the dominant architecture for server products and Windows-based desktop and laptop computers. Apple also started using x86 CPUs in their laptops and desktops in the late 2000's, which increased the x86 market share even more.

Xen: A wide-used type1 hypervisor that allows multiple operating systems operating as virtual machines on the same host machine. Xen is well-known as its using of paravirtualization, which can run modified paravirtualized guest operating systems in order to get high performance on x86 architecture.