A Survey of Mobile Device Virtualization: Taxonomy and State of the Art

JUNAID SHUJA and ABDULLAH GANI, University of Malaya, Malaysia
KASHIF BILAL, COMSATS Institute of Information Technology, Pakistan
ATTA UR REHMAN KHAN, King Saud University, Saudi Arabia
SAJJAD A. MADANI, COMSATS Institute of Information Technology, Pakistan
SAMEE U. KHAN, North Dakota State University, USA
ALBERT Y. ZOMAYA, University of Sydney, Australia

Recent growth in the processing and memory resources of mobile devices has fueled research within the field of mobile virtualization. Mobile virtualization enables multiple persona on a single mobile device by hosting heterogeneous operating systems (OSs) concurrently. However, adding a virtualization layer to resource-constrained mobile devices with real-time requirements can lead to intolerable performance overheads. Hardware virtualization extensions that support efficient virtualization have been incorporated in recent mobile processors. Prior to hardware virtualization extensions, virtualization techniques that are enabled by performance prohibitive and resource consuming software were adopted for mobile devices. Moreover, mobile virtualization solutions lack standard procedures for device component sharing and interfacing between multiple OSSs. The objective of this article is to survey software- and hardware-based mobile virtualization techniques in light of the recent advancements fueled by the hardware support for mobile virtualization. Challenges and issues faced in virtualization of CPU, memory, I/O, interrupt, and network interfaces are highlighted. Moreover, various performance parameters are presented in a detailed comparative analysis to quantify the efficiency of mobile virtualization techniques and solutions.

CCS Concepts: • **Computer systems organization** → **Real-time operating systems**; *Embedded and cyber-physical systems*; • **Software and its engineering** → **Operating systems**; *Virtual machines;*

Additional Key Words and Phrases: ARM, mobile virtualization, mobile cloud, smartphones

ACM Reference Format:
Junaid Shuja, Abdullah Gani, Kashif Bilal, Atta ur Rehman Khan, Sajjad A. Madani, Samee U. Khan, and Albert Y. Zomaya. 2016. A survey of mobile device virtualization: Taxonomy and state of the art. ACM Comput. Surv. 49, 1, Article 1 (April 2016), 36 pages.
DOI: http://dx.doi.org/10.1145/2897164

This work is partially funded by the Malaysian Ministry of Education under the High Impact Research grant of University of Malaya, UM.C/625/1/HIR/MOE/FCSIT/03.
Authors' addresses: J. Shuja and A. Gani (corresponding author), Centre for Mobile Cloud Computing Research, (C4MCCR), Faculty of Computer Science and Information Technology, University of Malaya, 50603, Kuala Lumpur, Malaysia; emails: junaidshuja@siswa.um.edu.my, abdullah@um.edu.my; K. Bilal, Department of Computer Science, COMSATS Institute of Information Technology, 22060, Abbottabad, Pakistan; email: kashifbilal@ciit.net.pk; A. ur Rehman Khan, College of Computer and Information Sciences, King Saud University, 12371, Riyadh, Saudi Arabia; email:dr@attaurrehman.com; S. A. Madani, Department of Computer Science, COMSATS Institute of Information Technology, 45550, Islamabad, Pakistan; email: madani@comsats.edu.pk; S. U. Khan, Department of Electrical and Computer Engineering, North Dakota State University, Fargo, ND 58108-6050, USA; email: samee.khan@ndsu.edu; A. Y. Zomaya, School of Information Technologies, Building J12, The University of Sydney, Sydney, NSW 2006, Australia; email: albert.zomaya@sydney.edu.au.

DOI: http://dx.doi.org/10.1145/2897164

1. INTRODUCTION

Over the past few decades, virtualization technology has witnessed widespread utilization in server space devices and cloud data centers [Bilal et al. 2014]. Virtualization technology has been instrumental in improving resource utilization in addition to reducing hardware costs [Mijat and Nightingale 2011]. Virtualization technology aims to share physical hardware resources efficiently among multiple operating systems (OSs). Resource sharing among OSs is achieved by abstracting or *virtualizing* the underlying hardware resources. A virtualization solution is composed of three major components: a hardware device; a hypervisor or virtual machine monitor (VMM), which virtualizes the hardware; and guest OSs, also known as virtual machines (VMs) [Heiser 2008]. The hypervisor emulates the underlying hardware and manages access initiated by the guest OSs. Virtualization of server-based devices has matured and has been around for decades [Penneman et al. 2013]. Resource consolidation, energy efficiency, and fault tolerance are the major use cases of virtualization in server space devices [Shuja et al. 2014b; Mustafa et al. 2015].

Mobile virtualization is a term used for real-time virtualization of diverse systems, such as multimedia- and entertainment-driven mobile devices (e.g., smartphones and tablets), critical real-time embedded systems (e.g., automobiles and control devices), and consumer electronics [Heiser 2009a]. Efficient virtualization of mobile devices enables simultaneous execution of multiple OSs along with their software stacks without compromising the real-time characteristics of a virtualized hardware [Armand and Gien 2009]. In the mobile space, virtualization enables execution of characteristically heterogeneous OSs on a single mobile device. The applications of such a multi-OS and mixed criticality setup on a mobile device include coexisting of real-time OS (RTOS) and general-purpose OS [Heiser 2009a], consolidation of personal and enterprise profiles on a single device [Mijat and Nightingale 2011], and sandboxing of open and secure OSs [Heiser 2008]. Mobile virtualization can lead to reduced software porting costs while facilitating coexistence of new and legacy OSs. Similarly, mobile virtualization can enable concurrent execution of a general-purpose OS for high-level application facilities and an RTOS for legacy real-time application stacks. Moreover, hardware spending can be reduced by running enterprise and personal OSs concurrently on a single mobile processor [Heiser 2008]. A comparison of server and mobile virtualization issues in provided in Table I.

In the recent past, mobile virtualization did not receive attention because of limited hardware support and resource constraints of mobile devices. Mobile and embedded devices were resource constrained with limited CPU power, memory, and battery. Hosting an additional OS on a resource-constrained mobile device is challenging, as it imposed serious performance overheads and decreased the responsiveness of real-time devices. However, mobile devices have transitioned from handheld sets, which enable simple voice and message services, to smartphones, which host multiple applications with high performance requirements, such as video processing, object recognition, and interactive games [Khan et al. 2014; Ahmed et al. 2015]. Mobile device processors have evolved from a single-core system with megahertz-grade speed to multicore system with gigahertz-grade speed each. Similarly, mobile device memory has increased to a magnitude of gigabytes [Andrus et al. 2011]. Virtualization has become a more viable option in mobile design space with the advent of multiprocessor system-on-chip (MPSoC) designs [Aguiar and Hessel 2010]. Virtualization is a major advancement in mobile technology that enables new use cases, operational environments, and applications while remaining within the budget of handheld devices [Mijat and Nightingale 2011].

Table I. Comparison of Server and Mobile Virtualization

Issue/Feature	Server Virtualization	Mobile Virtualization
Device energy	High	Low
Resource constrained	No	Yes
Use case	Homogeneous OSs	Heterogeneous OSs
Hard real-time capability	No	Yes
Hardware support	Since 2005	Since 2012
Major hardware instruction set architecture (ISA)	Intel	ARM

The growth of the mobile devices has been largely powered by the evolution of ARM processors. ARM processors use reduced instruction set computing (RISC), which implies that they have low number of transistors that lead to low power, low heat dissipation, and reduced costs. ARM-based processors dominate the embedded systems market with more than 90% of market share [Ramasubramanian 2011]. The virtualization of ARM instruction set architecture (ISA) is the main focus of this survey because of the dominance of ARM processors in mobile and embedded devices. Earlier ARM architectures had no native support for virtualization. Therefore, early mobile virtualization technologies had to rely on software-enabled virtualization support. This was a performance drawback because software-enabled virtualization features add complexity to the hypervisor design [Heiser 2009b]. Mobile virtualization solutions rely heavily on paravirtualization techniques and trap-and-emulate procedures to share mobile components among guest OSs without hardware virtualization support. The paravirtual techniques and trap-and-emulate procedures result in large overhead, thereby compromising the real-time capability of mobile devices by burdening the already constrained resources [Grunenberger et al. 2012].

ARM aims to set foot in the server market with its latest 64-bit ARMv8 ISA [Ortiz 2011]. Researchers have proposed 64-bit ARMv8-based servers that provide comparable performance with low energy consumption for cloud data center applications [Rajovic et al. 2014]. However, ARM processors should support virtualization efficiently to be eligible as a replacement of Intel processors in data center environments for high-performance computing [Smirnov et al. 2013; Goodacre 2013]. ARM architecture introduced hardware virtualization extensions in the ARMv7 ISA [Lanier 2011; ARM Limited 2011]. The advent of hardware virtualization extensions has resulted in the update of mobile virtualization solutions to enable real-time responsive and efficient hypervisors [Dall and Nieh 2014]. Hardware extensions enable systems to implement full virtualization efficiently and to reduce the frequency of hypervisor invocations for the maintenance of shared resources [Heiser 2011].

The use cases of virtualization in mobile and embedded systems can be categorized as follows:

—*Coexisting heterogeneous OSs*: Mobile virtualization enables a hardware platform to host heterogeneous OSs concurrently. For example, the bring your own device (BYOD) concept in enterprises enables employees to utilize a mobile device for both enterprise and personal profiles [Ding et al. 2014; Dong et al. 2015].
—*Multicore management*: Virtualization allows flexible allocation of resources to guest OSs in contrast to symmetric and asymmetric multiprocessor OS approaches [Bortolotti et al. 2013].
—*Security*: Virtualization facilitates simultaneous execution of security-critical applications and user interactive applications on the same mobile device through logical isolation controlled by the hypervisor [Russello et al. 2012].

The best use case of mobile virtualization is to run heterogeneous OSs concurrently on a hardware platform. Concurrent heterogeneous OSs are used to address conflicting application requirements. An application OS (e.g., Android and Windows Mobile) can be run concurrently with (1) an RTOS to provide low latency for real-time embedded systems [Green Hills Software 2013; Bialowas 2010], (2) a secure OS to provide trusted computing for safety critical applications [Heiser 2009a; Gaska et al. 2010], and (3) a legacy OS for application reuse [Mijat and Nightingale 2011]. Moreover, virtualization enables a BYOD use case where an enterprise employee can enable the logical segregation of office and personal environments on the same device [Carlson 2011]. The openness of popular mobile OSs, such as Android, has left users vulnerable to malicious applications that can manipulate stored user data [Gudeth et al. 2011]. However, a secure hypervisor can isolate guest OSs while limiting the vulnerability to one guest OS [Heiser 2008]. All of the aforementioned use cases require a separate *baseband* processor for real-time applications and an *application* processor for user applications. However, the cost of extra hardware can be avoided while adding incremental functionality on a single-core system utilizing virtualization technology. The architectural abstraction provided by the virtualization offers freedom to the system designer to map logically different OSs onto the same set of shared physical resources with dynamic partitioning. With the realization of MPSoC designs, embedded systems are adopting characteristics and use cases of enterprise space systems [Heiser 2011]. In MPSoC designs, virtualization provides adaptive resource allocation by adjusting resources to a demanding OS.

Recent advances in mobile technology demand a detailed study and a comparative analysis of hardware- and software-based mobile virtualization solutions. Gu and Zhao [2012] presented a survey on the real-time issues of mobile virtualization. However, the survey does not discuss hardware-based mobile virtualization solutions. Chen [2011] focused on various challenges to smartphone virtualization. However, the study did not include comparative analysis and details of various smartphone virtualization solutions.

To the best of our knowledge, a taxonomy and state-of-the-art of mobile virtualization techniques does not exist in the literature. The key contributions of this article include the following: a taxonomy of mobile virtualization techniques; a debate on open research issues and challenges in mobile virtualization with respect to processor, memory, I//O, interrupt, and network devices; a survey on state-of-the-art mobile virtualization solutions with the recent advances in hardware virtualization support in mobile processor architectures; and a comparative analysis of recent mobile virtualization solutions that have not been covered in previous works.

The rest of the article is organized as follows. In Section 2, we analyze the formal requirements of efficient mobile virtualization in addition to classification of the ARM ISA based on sensitive and privileged instructions. We also classify the techniques of mobile virtualization. Section 3 details the state of the art in mobile virtualization. The challenges and research issues in the virtualization of CPU, memory, and interrupt controller are also listed. Section 4 provides details of various mobile virtualization techniques. Section 5 provides a detailed comparison of mobile virtualization techniques. Section 6 concludes the discussion with future directions.

2. TAXONOMY OF VIRTUALIZATION TECHNIQUES

This section presents a thematic taxonomy of mobile virtualization techniques. Mobile virtualization techniques can be basically categorized into type-1 or type-2 virtualization. Type-1 virtualization techniques host a guest OS directly over the underlying hardware. Alternatively, the guest OSs in type-2 virtualization are virtualized over a host OS [Gu and Zhao 2012]. Mobile virtualization solutions are further classified based

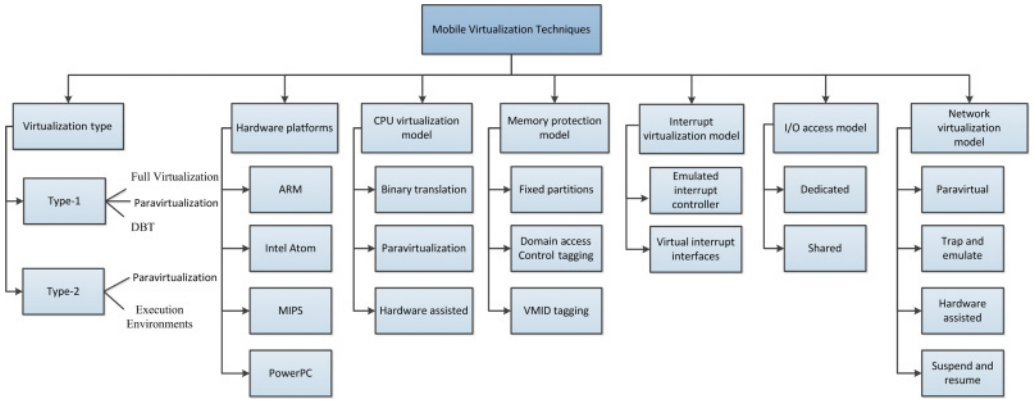

Fig. 1. Taxonomy of mobile virtualization techniques.

on hardware platforms. ARM processors dominate the mobile and smartphone market. However, Intel Atom, microprocessor without interlocked pipeline stages (MIPS), and performance optimization with enhanced RISC performance computing (PowerPC) ISA are also utilized in notebooks, network devices, and game consoles [Do 2011]. The CPU virtualization model describes the virtualization technique utilized to trap the sensitive nonprivileged instructions in the ARM ISA. Memory protection can be provided by three techniques: fixed partitioning, register-based memory tagging, and VM-based memory tagging. Similarly, interrupts can be virtualized by either a trap-and-emulate procedure or a virtual interrupt interface configured by the interrupt controller. I/O access can be shared among all the guest OSs or be dedicated to a guest based on priority. Network functionality can be virtualized by one of the four techniques: paravirtual interfaces based on hypercalls, the trap-and-emulate procedure for network interface access, hardware-assisted multiple interface access to the guest OSs, and the suspend and resume routine. Moreover, the requirements of classic virtualizability, such as equivalence, efficiency, and resource control, categorize an ISA as classically virtualizable or else [Penneman et al. 2013]. Figure 1 presents a taxonomy of mobile virtualization issues discussed in the article.

In the sections that follow, we first elaborate on the generic ARM architecture and other mobile hardware platforms. Then we list the formal requirements and taxonomy of mobile virtualization techniques.

2.1. Hardware Architectures and ARM

Although ARM captures nearly 90% of the mobile and embedded device market, other processor architectures that have their share in the market exist [Do 2011]. Intel Atom, IBM PowerPC, and MIPS are other major stakeholders in mobile and embedded space devices [Sud et al. 2012; Inoue et al. 2008]. RISC-based architectures are commonly utilized in mobile and embedded devices because of their low-power and low-cost operations.

ARM is an RISC architecture that is mostly deployed in battery-powered devices, such as smartphones, laptops, tablets, and embedded systems. An RISC-based architecture means that the required transistors for ARM processors are significantly fewer than those for typical complex instruction set computing (CISC) processors. Therefore, ARM processors possess desirable traits for battery-powered devices with reduced costs, heat, and power use [Penneman et al. 2013]. ARM architectures prior to ARMv7 define six privileged processor modes (i.e., IRQ, FIQ, SVC, UND, ABT, and SYS) and

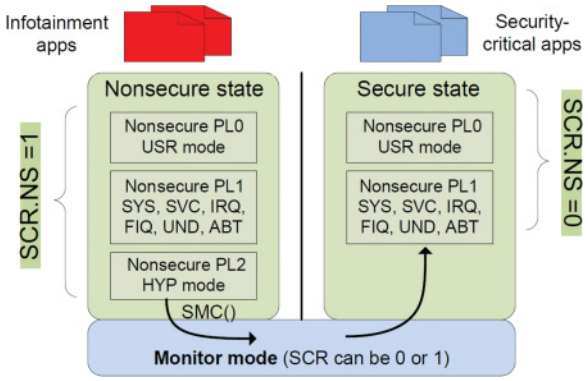

Fig. 2. ARM architecture—secure and nonsecure states.

an unprivileged mode (i.e., USR). The privileged modes lie in the highest privilege level (PL1), whereas the unprivileged mode lies in the lowest privilege level (PL0). All privileged modes are exception modes except for the SYS mode. The ARM ISA defines six exception types. Each exception defines a static jump to address called *exception vector*. The ARM ISA has 16 general-purpose registers: R0 to R15. All processor modes have banked (duplicate) copies of general-purpose registers. The current program status register (CPSR) stores the current state of the processor in the form of current instruction set, processor mode, memory endianness, and interrupt mask bits, among other information. Five saved program status registers (SPSR) are used to save a copy of the CPSR when an exception is taken [Penneman et al. 2013; Suzuki and Oikawa 2013]. ARM supports only memory-mapped I/O (MMIO). ARM architectures typically have two cache levels: a virtually indexed I/D L1 cache and a physically indexed L2 cache. The ARM ISA is divided orthogonally into secure and nonsecure states [ARM Architecture Group 2009, 2014b]. An additional superprivileged mode—the monitor mode—acts as a bridge between secure and nonsecure states. The SCR.NS bit of the secure configuration register (SCR) defines the current security state of the processor. The monitor mode allows manipulation of SCR.NS bit from a PL1 or higher privilege level. The major difference between ARMv7 and earlier ARM architectures is the hardware assists for virtualization. The ARMv7 adds new a hypervisor (HYP) mode and several new instructions that support hardware-assisted full virtualization. ARMv7 defines a new high privilege level (PL2) for the hypervisor mode. The previously defined privileged modes, known as the kernel modes, can run at the normal OS privilege level (PL1), whereas user applications run at the unprivileged level (PL0). Moreover, ARMv8 migrates from 32-bit ISA to 64-bit ISA. The ARM architecture with security extensions is depicted in Figure 2 [Mijat and Nightingale 2011].

Intel Atom is a low-voltage counterpart of the x86 architecture. Intel Atom processors utilize the Bonnel architecture, which translates CISC-based x86 instructions into RISC-based micro-ops. There are few x86 instructions that translate into more than one micro-op. Therefore, the Bonnel architecture–based Intel Atom achieves good performance per watt ratio [Totoni et al. 2012]. Intel Atom processors are commonly utilized in mobile Internet devices. Intel Atom processors come with hardware embedded support for virtualization [Sud et al. 2012]. PowerPC is another RISC architecture that recently introduced virtualization support in its upcoming processors. However, the majority of PowerPC virtualization solutions are based on legacy processors that do not support hardware-assisted virtualization. PowerPC ISA was originally intended for the personal computer. The 32-bit versions of PowerPC have been licensed for network

and mobile devices [PowerPC 2009]. The MIPS architecture has no native support for virtualization because of the presence of sensitive instructions and a single privileged processing mode. Therefore, MIPS suffers from the same anomaly as ARM mobile virtualization solutions. The MIPS architecture is widely used in embedded systems, video game consoles, and network routers [Aguiar 2014].

2.2. Formal Requirements of Virtualization

A processor ISA generally defines a hierarchy of privilege levels that enable an OS kernel and user applications to have different access rights to system resources. The OS kernel runs in the highest privilege mode (i.e., kernel mode), whereas the user applications run in the unprivileged user mode. The formal requirements of virtualization divide the instructions of an ISA into three categories [Penneman et al. 2013]:

—*Privileged instructions*: These instructions must be executed in a privileged mode; otherwise, they trap when executed in an unprivileged mode. Trap is a synchronous event that sends a call to the OS event handler.
—*Sensitive instructions*: These instructions may either change the context of the hardware resources or their behavior depends upon the configuration of hardware resources.
—*Innocuous instructions*: These instructions do not require privileged mode execution and do not change the context of hardware resources.

Sensitive instructions are further classified into control-sensitive instructions and behavior-sensitive instructions [Penneman et al. 2013; Aguiar and Hessel 2011]. Control-sensitive instructions try to change the allocated memory resources or the processor mode. An example of control-sensitive instruction is the `SVC` instruction in the ARM ISA that issues a supervisor call to the OS and changes the processor mode to a privileged mode [Penneman et al. 2013]. The execution of a behavior-sensitive instruction depends on the processor mode or its location in physical memory [Penneman et al. 2013; Heiser 2007]. The store return state `(SRS)` instruction in the ARM ISA is a behavior-sensitive instruction, as it is undefined in hypervisor mode and unpredictable in user mode [ARM Architecture Group 2014a]. Meanwhile, modern architectures, including ARM, do not contain location-sensitive instructions that can bypass address translation to reveal physical addresses [Penneman et al. 2013].

The requirements of a classic virtualizability state that a hypervisor should have include the following three properties [Penneman et al. 2013; Popek and Goldberg 1974].

—*Equivalence*: The hypervisor should provide an interface to the guest OSs that is essentially identical to the hardware platform. Therefore, all innocuous instructions are executed without the intervention of a hypervisor.
—*Efficiency*: The performance of a virtualized guest should be comparable to that of the native system with a dominant subset of instructions executed directly by the processor.
—*Resource control*: The hypervisor should control access to all physical resources and should be able to regain control of the allocated resources.

The resource control property requires that the hypervisor remains in control of the hardware resources and arbitrates access from the guest OSs. This implies that sensitive instructions that might change the context of hardware resources should be controlled by the hypervisor in a virtualized environment. Moreover, privileged instructions that must be executed in privileged mode require hypervisor intervention as the guest OS executes in unprivileged user mode.

2.3. Mobile Virtualization Techniques

Virtualization technologies can be broadly categorized as type-1 (bare-metal) and type-2 (hosted) virtualization techniques [Gu and Zhao 2012]. In type-1 virtualization, the hypervisor runs directly over the hardware in a privileged mode as a special type of OS. The virtualized guest is hosted over the hypervisor in an unprivileged mode. Therefore, type-1 hypervisors have more control over the hardware resources and arbitrate access of hardware resources among multiple guest OSs. Alternatively, the hypervisor in type-2 virtualization techniques executes in an unprivileged mode over the privileged OS. Therefore, type-2 hypervisors have lesser control over the hardware resources. Mobile virtualization techniques are further classified as follows.

2.3.1. Full Virtualization. An ISA supports full virtualization if the set of sensitive instructions is a subset of privileged instructions [Popek and Goldberg 1974]. Full virtualization is a virtualization technique in which the hypervisor presents to the guest OS the exact replica of the hardware platform [Vahidi and Ekdahl 2013]. The guest OS is hosted unmodified and is unaware of its virtualization. If a processor ISA provides a separate privilege level for the hypervisor, sensitive nonprivileged instructions automatically trap to the hypervisor and are emulated on behalf of the guest OS. Full virtualization can be further classified into hardware-assisted full virtualization and dynamic binary translation (DBT)-based full virtualization. Full virtualization supported by the processor ISA is known as hardware-assisted full virtualization. Full virtualization is efficient if most of the code is executed natively with a small portion of instructions trapping to the privileged mode. Otherwise, frequent traps to the hypervisor mode require context switches between unprivileged guest and privileged hypervisor mode, thereby adding latency to the emulation [Smirnov et al. 2013]. Although a trap instruction in the ARM ISA consumes considerably fewer cycles in the pipeline than in the x86 ISA, the high frequency of traps in modern OSs still prohibits usage of the trap-and-emulate procedure [Varanasi and Heiser 2011]. In modern pipelined architectures, an exception or trap drains the pipeline, and the branch caused by this exception or trap cannot be predicted by the branch prediction unit [Heiser 2007].

ARM architectures prior to ARMv7 do not have a separate privilege level for hypervisor. In such a case, DBT-based full virtualization techniques are applied to trap sensitive nonprivileged instructions to the hypervisor [Penneman et al. 2013]. DBT techniques translate sensitive nonprivileged instructions to sensitive privileged instructions at runtime. In DBT, the hypervisor has to scan each guest instruction, recognize sensitive nonprivileged instructions, and replace them with sensitive privileged instructions that trap to the hypervisor or branch instructions that avoid the performance overhead of trap-and-emulate procedure [Aguiar and Hessel 2011]. In this manner, DBT techniques can run unmodified guest OSs. However, DBT techniques require extra memory space for the code translation process. DBT techniques store recurring translated code blocks for future reuse to overcome dynamic translation overhead [Adams and Agesen 2006]. Therefore, DBT techniques burden resource-constrained mobile devices in terms of the memory footprint. In the ARM ISA, the program counter (R15) can be modified by several ALU instructions and is explicitly visible. This makes DBT virtualization for the ARM ISA challenging because the instruction translation process must monitor the instruction flow based on the program counter [Penneman et al. 2013].

2.3.2. Paravirtualization. In paravirtualization, the sensitive nonprivileged instructions in the guest OS are replaced by explicit hypervisor calls (hypercalls). The hypervisor presents the guest OS a high-level application binary interface (ABI) for communication of critical instructions through hypercalls. As paravirtualized systems avoid the

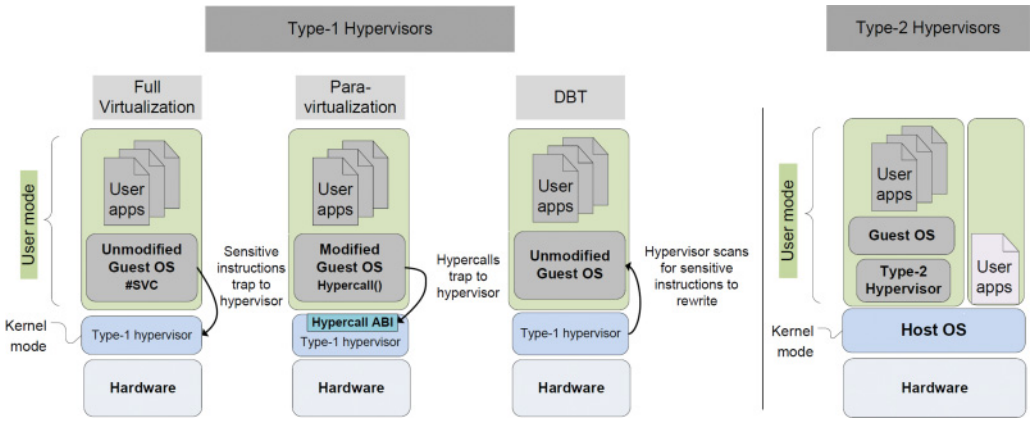

Fig. 3. Type-1 and type-2 virtualization techniques.

trap-and-emulate procedure to access the privileged resources, they usually outperform fully virtualized systems on the same hardware [Varanasi 2010]. The guest OS directly communicates with the hypervisor instead of indirectly invoking the hypervisor through virtual resource access. However, the engineering cost of this performance gain is significantly high in terms of kernel-level code changes required for hypercall interfaces [Forbes 2007]. Moreover, each hypervisor ABI presents a different interface to the guest OS. The process of developing and maintaining a patch for each set of guest OS and hypervisor interface is expensive [Penneman et al. 2013]. Furthermore, some OSs are not open source, whereas others do not allow redistribution of OS binary with modifications. The majority of mobile virtualization solutions apply paravirtualization to some extent because ARM ISAs until ARMv7 are not classically virtualizable. Figure 3 illustrates the taxonomy of mobile virtualization techniques.

Lightweight paravirtualization is an amalgam of paravirtualization and DBT [Dall and Nieh 2010]. Lightweight paravirtualization implements nonintrusive guest OS modifications and automated script-based translation that is architecture specific [LeVasseur et al. 2008]. Script-based translation identifies sensitive instruction from assembler files and replaces them with hypercalls to the hypervisor. Microkernel-based paravirtualization techniques are also found in some of the popular mobile virtualization solutions [Armand and Gien 2009; Heiser and Leslie 2010]. The basic concept of a microkernel involves reducing the kernel code to a minimal set and moving OS services from a kernel space to a user space. The services are implemented as user-level servers and fast interprocess communication (IPC) mechanism is provided [Armand and Gien 2009]. The aim of the microkernel-based OS approach is to achieve generality and flexibility. Microkernels can be utilized for virtualization, as they provide minimal software abstraction for the underlying hardware resources [Heiser and Leslie 2010]. Type-2 virtualization techniques also rely on paravirtualization for a nonvirtualizable ARM ISA. Moreover, type-2 virtualization techniques can utilize process virtualization to host multiple execution environments [Dall et al. 2012; Chen et al. 2015]. Mobile virtualization techniques are classified and compared in Table II.

Techniques other than full virtualization are expensive in terms of software engineering costs [Heiser 2011]. However, full virtualization adds complexity on the hardware level [Penneman et al. 2013]. DBT-based full virtualization allows for unmodified guest OSs. However, the performance overhead of the trap-and-emulate procedure with DBT techniques remains significant. On the contrary, paravirtualized solutions avoid the performance overhead of the trap-and-emulate procedure but add complexity to the

Table II. Classification of Type-1 Mobile Virtualization Techniques

Virtualization Technique	Category	Principle	Advantages	Disadvantages
Full virtualization	Hardware assisted	Utilize hypervisor mode and virtualization extensions	• Lower hypervisor complexity • Unmodified guest OS	• Sensitive nonprivileged instructions in ISA • Frequent traps leading to expensive context switches
	Dynamic binary translation	Translate sensitive nonprivileged instructions dynamically	• Applicable to nonvirtualizable platforms • Allows unmodified guest OS	• Higher hypervisor complexity due to scanning of each instruction as potential sensitive nonprivileged • Overhead of trap-and-emulate procedure • Memory space required to store translated instructions
Paravirtualization	Pure paravirtualization	Patch guest OS to replace sensitive nonprivileged instructions with explicit hypercalls	Scalability to several hardware and OS platforms	• Kernel-level patch required for guest OS • Patch updates required to keep up with upstream versions of guest OS • No standard hypervisor interfaces • Closed-source guest OS can not be patched
	Lightweight paravirtualization	Script-based identification and replacement of sensitive nonprivileged instruction	Smaller patch required to identify sensitive instructions from assembly code	Separate script required for each guest OS to identify sensitive nonprivileged instructions
	Microkernel-based paravirtualization	Reduce kernel size and move OS services to user space	Smaller kernel size provides better security and efficiency	Efficient communication paths (IPC) required for user–kernel space channels

design of both the hypervisor and the guest OS. Both paravirtualization and DBT techniques are versatile, which enable virtualization of architectures that are not classically virtualizable. Moreover, paravirtualized solutions must maintain an updated set of guest OS and hypervisor for evolving hardware and software systems [Penneman et al. 2013; Li et al. 2015].

3. STATE OF THE ART AND CHALLENGES

ARM architectures until ARMv7-A are not classically virtualizable, as they did not meet the requirements of classic virtualization [Suzuki and Oikawa 2013]. The ARMv7 ISA introduces virtualization extensions that enable ARM to implement hardware-assisted full virtualization [Do 2011]. The sharing of the physical resources and execution of guest OS in unprivileged user mode raises the challenge of CPU, memory, interrupt, I/O, and network functionality virtualization. Moreover, smartphones are composed of a plethora of devices that the user applications commonly utilize, such as camera, GPUs, and communication interfaces [Zhang et al. 2012]. Multiplexing these devices among multiple OSs requires novel approaches that are different from server virtualization techniques [Andrus et al. 2011]. Hardware virtualization support for the x86 ISA was introduced a decade ago, and quantitative comparison of software and hardware virtualization techniques have been detailed [Adams and Agesen 2006]. However, quantitative comparison of software and hardware approaches to ARM virtualization has not been investigated thus far because of the recent availability of ARM virtualization extensions. In the following sections, we will discuss the state of the art in CPU, memory, interrupt, I/O, and network functionality virtualization. Moreover, the

Table III. ARMv7 Sensitive and Privileged Instructions

Instruction	Operation	Sensitive	Privileged
CPS	Change processor mode	√	×
LDC	Load coprocessor	√	×
LDM (user registers)	Load multiple user registers	√	×
LDM (exception return)	Load multiple user registers and SPSR to CPSR	√	×
MCR	Move to coprocessor from register	√	×
MRC	Move to register from coprocessor	√	×
MRS (SPSR)	Move to register from SPSR	√	×
MSR (CPSR and SPSR)	Write to system registers	√	×
RFE	Return from exception	√	×
SEV	Send event	√	×
SRS	Store return state	√	×
STC	Store coprocessor	√	×
STM (user registers)	Store multiple registers from user mode	√	×
SVC	Supervisor call	√	√
SUBS PC, LR	Exception return without the use of stack	√	×
WFE	Wait for event	√	×
WFI	Wait for interrupt	√	×

challenges to software- and hardware-based mobile virtualization techniques for various ARM architectures are elaborated.

3.1. CPU Virtualization

ARM architectures prior to ARMv7 contain sensitive nonprivileged instructions and only two privilege levels. Therefore, virtualization techniques for earlier ARM architectures are forced to either place both the hypervisor and the guest OS in the same privilege level or run the guest OS in user mode. The hypervisor must trap all instructions for emulation if the guest OS also runs in the privileged mode. On the contrary, sensitive nonprivileged instructions do not produce desired results when executed in user mode. Therefore, the virtualization solutions had to either adopt hypercalls or perform DBT for sensitive instructions while hosting the guest OS in user mode [Wu 2013]. Although the hypercalls must be adopted for each set of guest and host OS, DBT techniques must trap and emulate both sensitive privileged and nonprivileged instructions, resulting in low performance and high virtualization overhead. Special measures are required for the security of guest OS page tables in case the guest OS and the user applications run at the same privilege level [Hwang et al. 2008]. The cost of the trap-and-emulate procedure can be avoided with static binary translation techniques that rewrite sensitive instructions with branch instructions that do not require trap and corresponding context switches [Smirnov et al. 2013].

Researchers [Penneman et al. 2013; Suzuki and Oikawa 2013] performed a formal analysis of the ARMv7 ISA to categorize sensitive and privileged instructions. The analysis shows that the ARMv7 ISA is not classically virtualizable because of the presence of sensitive nonprivileged instructions. Sensitive nonprivileged instructions are those that intend to change the state of the system resources but do not have the privilege to do so. Special care is required to trap sensitive nonprivileged instructions in a virtualized system. The summary of their findings on the ARMv7 ISA are provided in Table III.

The ARMv7 virtualization extensions add a new hypervisor (HYP) mode and several new instructions that support hardware-assisted full virtualization. The hypervisor mode has its own set of banked registers. The hypervisor mode hosts the hypervisor in a high privilege mode (PL2). The previously defined privileged modes, known as the

kernel modes, can run at the normal OS privilege level (PL1). The user applications lie at the unprivileged level (PL0). The introduction of hypervisor mode allows the guest OS to run in the same privilege level as in a nonvirtualized environment while the hypervisor controls guest OS access to the shared resources. The hypervisor syndrome register (HSR) saves the context of the event that caused the trap for emulation support when an instruction traps to the hypervisor. This frees the hypervisor designer from saving the trap information in the software [ARM Architecture Group 2014a]. The hypervisor call (HVC) instruction is added to the ISA to invoke the hypervisor mode. All sensitive nonprivileged instructions listed in Table III except for the send event instruction (SEV) can be trapped to the hypervisor with the help of configurable traps. The hypervisor mode is turned off along with all virtualization extensions when a single OS operates without a hypervisor.

The introduction of hardware virtualization extensions has not completely addressed the problem of CPU virtualization. SEV instruction is the only type of sensitive nonprivileged instructions in the ARMv7 ISA that cannot be configured to trap [Penneman et al. 2013]. In a multiprocessor architecture, software executing on different cores can communicate with one another through SEV instruction. Therefore, a configurable trap for (SEV) instruction is necessary for multicore ARM virtualization. Moreover, emulation support is not provided for the write-back LDR instruction [Varanasi and Heiser 2011]. The LDR instruction allows adding an immediate value to the address register of the previous load instruction. The previous load instruction is not stored in the HSR register and needs to be loaded explicitly from guest memory. The support for multicore virtualization becomes a critical requirement for mobile hypervisors with the recent growing trend in multicore devices. Most mobile hypervisors also lack support for multicore virtualization [Aguiar and Hessel 2011]. Furthermore, the hypervisor mode can only virtualize the nonsecure state of the ARMv7 ISA. Changes to the device firmware are required to host multiple OSs in the secure world of ARM [Vahidi and Ekdahl 2013].

3.2. Memory Virtualization

Virtualized systems encounter challenges while sharing memory resources among multiple guests, such as overhead of managing shadow page tables (SPTs), protection of memory spaces from unauthorized access, and translation lookaside buffer (TLB) maintenance due to context switches [Ding et al. 2012]. In a virtualized system, the virtual addresses of a process do not directly translate to the machine physical address. The guest OS translates the guest virtual address to the guest physical address. The hypervisor intercepts the guest OS access to physical memory and performs the second stage of address translation while translating the guest physical address to the machine physical address with the help of SPT [Mijat and Nightingale 2011]. SPT is a copy of the guest OS page tables that enables the hypervisor to synchronize itself with guest memory mappings. Prior to ARMv7, only one stage of the two-level address translation was supported by the hardware page table walker. The second stage of the address translation had to be performed by an inherently slow software process. Moreover, the hypervisor had to trap all guest page table accesses to be reflected in the SPT for coherency. The guest OS and its applications resided in the same user mode because previous ARM versions only provided two levels of privilege, thereby making the guest OS memory space vulnerable from its own applications. Therefore, the hypervisor had to maintain access permissions for privileged and nonprivileged memory space [Ding et al. 2012]. Previous hypervisors used ARM domain access control registers (DACRs) to provide memory protection among a hypervisor, guests, and user applications [Douglas 2010]. Moreover, previous ARM ISAs do not implement any TLB tagging mechanism. Therefore, multiple guest OS entries cannot reside in a single TLB. A

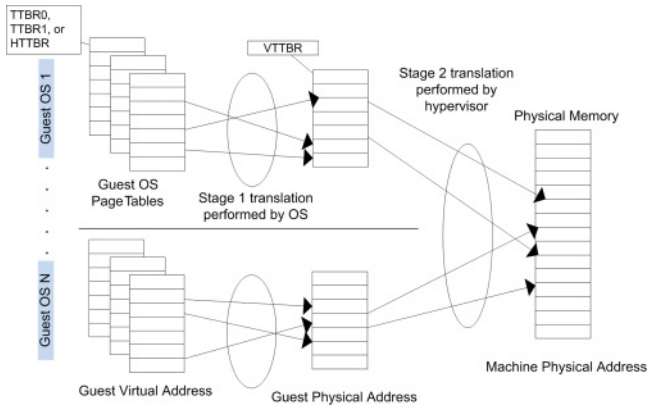

Fig. 4. Two-stage address translation.

TLB flush operation is performed when a context switch between guest OSs and a hypervisor occurs, and the TLB is repopulated for the live OS.

ARMv7 introduces the system memory management unit (SMMU) architecture that can perform two-stage hardware page table walk and TLB tagging. The first set of address translation is controlled by the guest OS with the help of the translation table base register (TTBR), whereas the second stage of translation is controlled by the hypervisor with the help of virtual TTBR (VTTBR). Given that both stages of address translation are performed in hardware, the hypervisor does not need to maintain an SPT and thereby avoids the cost of implementing a page table walk in the software [Varanasi 2010]. The TLB tags page table entries with a guest-specific virtual machine identifier (VMID) and a process-specific address space identifier (ASID). The TLB tagging mechanism enables the coexistence of multiple guest OSs and hypervisor entries in a single TLB cache [vIritical Project 2013; Goodacre and Sloss 2005]. Moreover, the page table tagging mechanism also provides memory protection and helps the hypervisor avoid the cost of a TLB flush on each context and guest switch. Furthermore, ARMv7 introduced the 40-bit addressing that leads to 1TB of physical memory space instead of the normal 4GB space resulting from 32-bit addressing. The guest OS is still presented with a 32-bit address space. However, the large physical address space reduces the address map congestion between multiple guest OSs and their applications [Goodacre 2011]. The SMMU does not free the hypervisor of invocation during the the memory management process. The second stage address translation for each page table access made by guest OSs is supervised by the hypervisor. If a context switch occurs, the hypervisor must configure the second stage of page table translation registers accordingly. Figure 4 depicts the two-stage address translation performed in hardware [ARM Architecture Group 2014a].

3.3. Interrupt Virtualization

ARM architecture provides a generic interrupt controller (GIC) to route interrupts generated by guest OSs to the CPU and also route interprocessor interrupts. A GIC compromises an interrupt distributor and a CPU interface for each core. The interrupt distributor configures the GIC on system boot and delivers the interrupt received from devices to a CPU interface. A CPU interface routes acknowledge (ACK) and end of interrupt (EOI) events to the designated CPU core. Software virtualization techniques must emulate the GIC on each interrupt and maintain shadow structures for each guest OS to share GIC among multiple guest OSs. On the ARM ISA, interrupts are

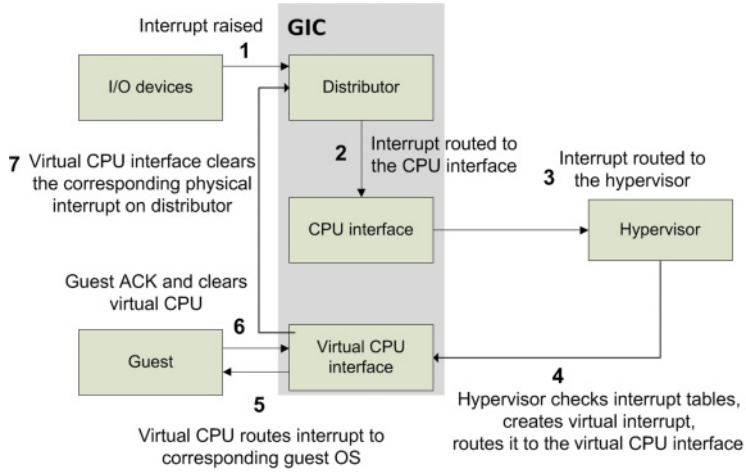

Fig. 5. Interrupt in the virtual environment.

configured globally—that is, either all interrupts invoke the hypervisor or all interrupts are directed to the guest OSs [Varanasi and Heiser 2011]. The hypervisor loses control of the hardware if interrupts are trapped to the kernel mode. However, the hypervisor must emulate the interrupt of CPU interface for each such event if each interrupt is trapped to the hypervisor mode.

ARMv7 introduces support for interrupt virtualization in the form of virtual GIC (VGIC) [Brash 2010]. The VGIC introduces a virtual CPU interface to communicate with guest OSs. When the hypervisor receives a physical interrupt, it either processes the interrupt itself or forwards it to a guest OS based on interrupt priority and target guest OS [ARM Architecture Group 2010a]. The ARMv7 ISA can manage a separate interrupt state for each guest OS. ACK and EOI events are also supported by the virtual CPU; hence, they do not trap to the hypervisor. Moreover, interrupts can be individually configured on per guest basis [Penneman et al. 2013]. The interrupt distributor still requires emulation by the hypervisor, as a virtual distributor interface is not supported by current ARM virtualization extensions. In a virtualized multicore environment, core-to-core communication may require modification of interrupt distributor, which in turn requires emulation. A flow chart of a hardware assisted virtualization of interrupt intended for a guest OS is depicted in Figure 5.

3.4. I/O and Timer Virtualization

The I/O devices in a virtualized environment can be either dedicated to a guest or shared among multiple guests. SMMU can support one-stage address translation to enable device drivers to be programmed into the guest OS user space for dedicated access [Goodacre 2011]. The interrupt vector tables of the remaining guests need correct memory mapping to access the dedicated device. In case of sharing, the trap-and-emulate or paravirtualization approach can be applied for device sharing. The trap-and-emulate approach requires that the hypervisor trap each I/O access and emulate it on behalf of the guest OS. In the paravirtualized approach, a modified API is presented by the hypervisor to the guest OS to access the device drivers. A paravirtualized I/O device has a front-end driver residing in the guest OS containing hypercall mechanism for I/O access and a back-end driver residing in the hypervisor that answers the hypercalls. I/O mechanisms in the ARM ISA are based on MMIO devices accessed through load/store instructions. The guest OS cannot access the memory directly and has to go through

two-stage address translation, one of which is controlled by the hypervisor. A load/store instruction is not sensitive if it operates on the virtual address space assigned to the guest OS. Prior to the ARMv7 ISA, access outside the allocated memory region can only be trapped to the hypervisor, as user mode and exception modes are located at the same privilege level. However, in the ARMv7 ISA, access outside the allocated memory region can be trapped to the hypervisor or any of the exception modes [Penneman et al. 2013]. The syndrome register can save information about the source and destination of the load/store instruction, which simplifies the task of hypervisor in emulation of the trapped instruction. Most kernel virtual machine (KVM)-based mobile virtualization solutions utilize QEMU for device emulation support [Ding et al. 2012; Dall and Nieh 2014].

OSs frequently use a timer to manage scheduled processes over a time slice. Granting access to the guest OS to manipulate the physical timer can lead to guest control over the CPU for long and unscheduled periods. In a virtualized environment, trapping each access to the physical timer can lead to performance overhead [Dall and Nieh 2014]. The ARMv7 ISA introduces a virtual timer and virtual counter for timer virtualization [ARM Architecture Group 2010b]. The hypervisor can access the physical timer, whereas the guests can access the virtual timers from kernel space. ARM supports one virtual timer per physical core that contains a value that is a 64-bit offset of the physical timer [ARM Limited 2011]. The guest can program virtual counters without trapping to the hypervisor. However, virtual timers can only raise hardware interrupts to trap to the hypervisor [Dall and Nieh 2014].

3.5. Network Functionality Virtualization

The virtualization of mobile network interfaces can be achieved by applying various software- and hardware-based techniques, namely paravirtualization, trap and emulate, hardware-assisted virtualization, and guest OS suspend and resume [Lee et al. 2015; Andrus et al. 2011].

In software-based techniques, the hypervisor manages the physical network interface and bridges it to multiple guest OSs. Hardware-based virtualization techniques allow network devices to advertise their ability to be shared among multiple guest OSs [Liang and Yu 2015]. The most convenient manner for virtualizing communication interfaces among multiple guest OSs is to provide paravirtual interfaces. In paravirtualization, modified front-end drivers in the guest OSs can communicate with the back-end driver in the hypervisor that exposes optimized hardware interface [Gu and Zhao 2012]. In software-based trap-and-emulate approach, the hypervisor manages physical resources (NIC, baseband, etc.) and bridges them between multiple virtual entities. Emulated network interfaces offer great compatibility, as all guest OS can utilize them in the same manner as the physical interface. The hypervisor has to trap and emulate each access made to the physical interface. However, emulated network interfaces have high overhead for mobile devices because of a large number of traps [Xia et al. 2011].

Single root I/O virtualization (SR-IOV) provides hardware assistance for virtualization by utilizing virtual NIC interfaces that can be assigned to separate guest OSs [Xia et al. 2011]. An SR-IOV-enabled network interface provides duplicate network components, network addresses, and queues that can be directly assigned to a guest OS. SR-IOV still face overheads, as the hypervisor has to handle interrupts generated by virtualized NIC [Shea and Liu 2012]. The suspend and resume method for virtualization works by handing control of the underlying hardware to the guest OS executing in the foreground. The state of the other guest OSs is stored in the background. When the system context switches a foreground VM to the background, the state of devices, including network interfaces, is stored, and a VM from the background is scheduled to

execute in the foreground with privileged access to all of the mobile subsystems [Dall et al. 2012].

Virtualizing network functionality means that the network hardware, baseband, and link layer capabilities have to be shared among multiple guest OSs [Grunenberger et al. 2012; Liang and Yu 2015]. Network connectivity in mobile devices is often provided through multiple communication interfaces such as WiFi, cellular radio, and Bluetooth. The challenges to network subsystem virtualization are listed as follows:

—*Standardization*: Unlike memory and CPU virtualization, no standard set of techniques for network subsystem virtualization exists. In particular, no specific support for network interface virtualization exists in most of the mobile virtualization solutions [Shea and Liu 2012].
—*Network stack duplication*: Mobile devices rely on multiple layers of network stack to distinguish data from different processes. Virtualization lead to duplication of network stacks and significant overhead for each guest [Andrus et al. 2011].
—*Mobility*: The mobility of mobile devices leads to unstable network conditions. Wireless connectivity based on device location and network availability makes network conditions unstable for mobile devices, such as the dynamic IP address [Xia et al. 2011].
—*Proprietary radio stack*: Each mobile device vendor provides its own proprietary radio interface stack. The radio interface stack is a blackbox, as user-level radio interface layer (RIL) libraries and the baseband processor details are proprietary and closed source [Andrus et al. 2011]. Therefore, it is impossible to virtualize any aspect of the radio interface stack at the kernel level without device vendor support.
—*Caller ID duplication*: Subscriber identification modules (SIMs) are an integral part of mobile system communications. Aside from hardware plurality, no virtualization standard exists to create multiple instances of SIM with separately assignable caller IDs for guest OSs [Andrus et al. 2011].

The majority of mobile virtualization solutions do not include virtualization of network functionality tailored for resource-constrained and tightly integrated mobile devices [Dall et al. 2012; Akhunzada et al. 2015]. Moreover, hardware assistance in mobile processors, such as ARM, does not contain support for network function virtualization other than generic support for I/O virtualization. Therefore, network functionality virtualization customized for mobile devices must be investigated in future mobile virtualization solutions. Table IV provides a summary of challenges to virtualization of mobile device components and the corresponding solutions.

4. MOBILE HYPERVISORS

Several commercial and open source mobile virtualization solutions exist in the market. The issue with purely commercial solutions, such as INTEGRITY [Green Hills Software 2013] and WindRiver [Bialowas 2010] is that they are not open source. Moreover, their limited documentation makes parametric comparison hard. However, significant deployment of some open source mobile virtualization technologies such as OKL4 [Heiser 2007] and Xen [Suh 2007] can be found in the commercial market. Such mobile virtualization technologies provide the best of the both worlds with detailed feature comparison and commercially available products. Several research groups around the globe have been focusing on various issues of mobile virtualization, such as security [Vahidi and Ekdahl 2013], MPSoC virtualization [Bortolotti et al. 2013; vIrtical Project 2013], hardware-assisted virtualization [Varanasi and Heiser 2011], and server-grade ARM virtualization for data centers [Smirnov et al. 2013]. We will discuss mobile virtualization solutions categorized based on their virtualization technique in the sections that follow. Some of the mobile virtualization solutions follow a hybrid approach by

Table IV. Challenges to ARM Virtualization and Hardware/Software Solutions

Hardware component	Challenge	Software solution	Hardware support (ARMv7)
CPU	Sensitive non-privileged instructions	DBT, paravirtualization	Sensitive instructions be a subset of privileged instructions
	Two Privilege levels	Hypervisor resides in PL1, guest OS in unprivileged level (PL0)	Add another higher privilege level (PL2) for hypervisor
Memory	Two-level address translation	Maintain SPT in the hypervisor	Two level address translation with SMMU
	Access permissions to memory space of guest OS and hypervisor	Fixed memory partitions	TLB tagging with VMID
Interrupt	Sharing GIC among multiple OSs	Emulate GIC for each interrupt	VGIC and CPU interfaces for interrupt processing
I/O	Access to shared I/O devices	Trap-and-emulate OR paravirtualize device drivers	Syndrome register for storage of source/destination of load/store instruction
Network function	Virtualization of communication interfaces	Trap-and-emulate OR paravirtualize device drivers	SR-IOV enabled network interfaces providing duplicate network components to guest OSs

virtualizing different parts of the hardware with different techniques. We categorize hybrid virtualization techniques based on their major influences and adoptions. The main focus of this article is to discuss ARM-based mobile virtualization solutions. For this study, we selected ARM-based mobile virtualization solutions based on the recent publication in major field journals and conferences. However, we have discussed a mobile virtualization solution based on each of Intel Atom, PowerPC, and MIPS architectures to present trade-offs between these hardware platforms and corresponding virtualization solutions. Moreover, several ARM instruction set emulators also exist, such as Qemu [Bellard 2005] and gem5 [Binkert et al. 2011]. The purpose of ARM emulators is to translate instructions from one ISA to another. ARM emulators facilitate testing of mobile applications on x86-based systems by translating an ARM ISA to an x86 ISA [Rege et al. 2013]. Therefore, ARM emulators are not categorized as mobile virtualization solutions that allow hosting of multiple OSs on a single hardware platform [Nikounia and Mohammadi 2015]. However, Qemu is used as a base for instructions translation and device emulation in many mobile and server virtualization solutions [Smirnov et al. 2013]. We do not consider Java virtual machine (JVM)-based virtualization solutions in this study. JVM-based virtualization solutions enable applications to execute across heterogeneous execution environments, whereas the focus of our study is hardware virtualization that enables hosting of multiple OS instances on a mobile device.

4.1. Paravirtualized Solutions

4.1.1. Xen-Based Solutions. Xen is the major proponent of paravirtualization techniques in x86 and ARM architectures [Hwang et al. 2008]. As the ARM processors with hardware virtualization extensions are making their way into the market, paravirtualization solutions are still the only viable option for the majority of mobile devices.

The first effort to virtualize ARM with Xen utilized the StrongARM (ARMv4) ISA and Xen 1.2 kernel [Ferstay 2006]. The Xen-based hypervisor was able to paravirtualize a single mini-OS guest. With only two privilege levels available on the StrongARM, Xen executes in kernel mode, whereas the guest OS and its applications execute in user

mode. To protect the guest OS from its malicious applications, Xen mediates access to the virtual address space. As the guest OS executed in user mode, it utilizes hypercalls to perform privileged functionality. To handle exceptions, the guest OS registers its exception descriptor table with the hypervisor. Interrupts are replaced by lightweight asynchronous events from the Xen to the guest OS. The Xen is mapped to the highest 64MB address space with read/write access enabled from the kernel mode. With no TLB tagging mechanism available in StrongARM, a TLB flush was required on each context switch. A nonoverlapping memory scheme was devised to eliminate the TLB flushes with domain ID tagged TLB entries. However, nonoverlapping memory partitions offer inflexible design for virtualized memory systems. The solution does not provide support for I/O paravirtualization. The device I/O is managed with asynchronous events and data rings to transfer data to and from the guest OS.

Hwang et al. [2008] and Suh [2007] started customizing Xen for ARM with split mode virtualization. Linux kernel 2.6 was utilized as the guest OS on an ARMv5 ISA. To minimize modification to the guest Linux for execution in user mode, the user mode is split into two logical modes (user mode and kernel mode). The hypervisor controls logical mode switches based on events. Exception handling is managed by the hypervisor, which saves the processor state to the virtual registers and issues an upcall to the kernel mode to deliver the exception to the guest OS. The guest uses hypercall to return control back to the hypervisor after exception completion. Sensitive instructions present in the ARMv5 ISA are replaced by hypercalls. ARM DACR bits are used to differentiate user, kernel, and hypervisor address space. Nonoverlapping memory partitions are utilized for the user, guest, and hypervisor to avoid the cost of TLB maintenance. Xen on ARM provides both split and coordinated device driver access among guest OSs [Suh 2007]. Further works on the Xen ARM include enhancing real-time capability by reducing the I/O latency and providing paravirtualization support for the vector floating-point (VFP) unit [Yoo et al. 2013]. A simple Xen/ARM paravirtualization requires modification of approximately 4,500 lines of code in the Linux kernel [Suh 2007].

EmbeddedXEN [Rossier 2012] runs two guest OS instances that are concatenated with the binary image file of the hypervisor. One of the guest OSs runs in the controller domain and hosts back-end device drivers, whereas the other host runs in the user domain and hosts front-end drivers. During system boot, the hypervisor parses the binary image file in ELF format to extract the two guests OSs. This results in a system design where the hypervisor completely trusts the guest OS and executes it in the privileged mode. EmbeddedXEN design leads to a modified hypercall that does not require a context switch as the guests jumps to a predefined hypervisor code for all privileged instructions; linear mapping of main memory for page tables, as the guest OSs and hypervisor trust each other and reside in same address space; and the guest OS having privileged access to the I/O devices. EmbeddedXEN design is useful if the guest OSs to be hosted are predefined such that they can be concatenated with the hypervisor in a single disk image. However, the EmbeddedXEN leads to extremely rigid design that does not allow modification of guest OS binary and only the binary concatenated statically is trusted.

4.1.2. ITRI. ITRI is a hybrid hypervisor that utilizes both lightweight paravirtualization and static binary translation for multiguest virtualization [Smirnov et al. 2013]. ITRI is based on KVM Linux kernel that is customized for the nonvirtualizable ARMv7 MPSoC board. KVM is part of the mainline Linux kernel family that provides the basic functionality to turn a Linux kernel into a hypervisor. The host OS or the hypervisor controls the SOC from SVC mode while guest OSs run in user mode. Sensitive nonprivileged instructions are patched with software interrupt (SWI) instructions using a static

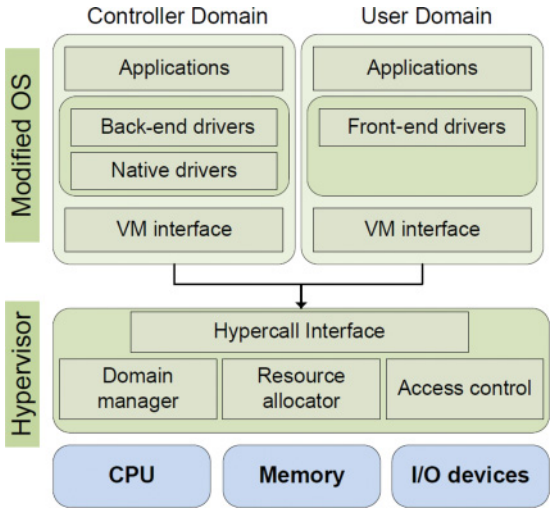

Fig. 6. Paravirtual mobile hypervisor.

binary translation in ITRI. Compiled code of a set of sensitive instructions is appended to the guest binary image. ITRI uses branch instructions to the compiled code to avoid context switches on SWI instructions. Memory is virtualized using SPT, and guest SPT modifications are notified to the hypervisor using hypercalls. Front-end I/O drivers in the guests utilize the back-end drivers hosted in the ITRI for I/O emulation [Russell 2008]. The QEMU process that creates a guest environment and corresponding static memory address space can be accessed from the user space through the input and output control (ioctl) interface. ITRI paravirtualizes the exception handler to recognize exceptions that do not require a context switch and can be fulfilled by the guest OS. ITRI is the only server-grade ARM hypervisor that supports guest migration. To migrate a guest, ITRI copies the guest state iteratively to the target machine. ITRI keeps track of the dirty pages with write protection faults enabled on all page entries.

4.1.3. MobiVMM. MobiVMM [Yoo et al. 2008] provides mobile virtualization with the help of a paravirtualized Linux port and hardware abstraction layer (HAL) that is an essential part of many paravirtualization solutions. HAL abstracts the hardware functionality and implements a set of paravirtualized drivers that can be accessed with hypercalls. Sensitive instructions are replaced by hypercalls into the HAL of MobiVMM. The hypervisor has a reserved memory region in the highest address space while guest OSs and their applications share an overlapping memory space. The nonoverlapping memory region of the hypervisor helps to avoid TLB flush on context switches between the hypervisor and the guests. MobiVMM utilizes preemptive CPU scheduling for the real-time guest and pseudopolling for interrupt handling. Figure 6 depicts a paravirtualized solution for mobile virtualization.

4.1.4. Proteus Hypervisor. PowerPC is another RISC architecture that recently included virtualization support in its upcoming processors [PowerPC 2009]. However, coinciding with ARM virtualization solutions, most PowerPC virtualization solutions are based on earlier versions of the architecture that did not include hardware assists [Mittal et al. 2013]. Proteus [Gilles et al. 2013] is a PowerPC architecture–based hypervisor that supports MPSoC designs. Proteus is a hybrid hypervisor that implements both full- and paravirtualization without specific hardware support. Although hardware support for virtualization was released in newer PowerPC devices, Proteus is based

on previous releases of PowerPC and implements full virtualization with the trap-and-emulate procedure. For paravirtualization, Proteus implements a microkernel by moving I/O device drivers to user space. Supervisor space only executes hypercall handler, VM scheduler, and IPC modules. Proteus design is symmetric multiprocessing (SMP); when a guest traps to the hypervisor, the hypervisor takes control of its assigned core. A small part of the hypervisor runs on each core to support full virtualization and forward interprocessor interrupts. Proteus executes the hypervisor in supervisor mode, and the guests OSs are executed in problem mode. Proteus hypervisor implements a semaphore-based solution for access to shared devices for MPSoC.

4.2. Lightweight Paravirtualized Solutions

4.2.1. KVM for ARM. Dall and Nieh [2010] proposed a KVM-based mobile virtualization solution for ARM. The KVM-based solution paravirtualizes the Linux guest by utilizing an automated script that modifies the guest at runtime to issue hypercalls to the KVM hypervisor instead of sensitive instructions for emulation. The automated script employs regular expression-based classification of sensitive nonprivileged instructions present in the assembly code of the guest OS. An encoding for all sensitive nonprivileged instructions with their operands is defined to be expressed as an SWI instruction. The guest OS exception vector tables are remapped to the hypervisor vector table to handle exceptions. The KVM for ARM hypervisor manages SPT for memory virtualization. Level-1 page table entries are tagged with 4-bit DACR tags for memory protection. KVM for ARM appends the DACR tags with the SPT entries. For device virtualization, a built-in QEMU emulator is utilized.

4.2.2. ARMvisor. ARMvisor is another KVM- and QEMU-based lightweight ARM paravirtualization solution [Ding et al. 2012]. The QEMU creates a guest environment and emulates I/O devices using the ioctl interface provided by the KVM modules. The guest utilizes lightweight traps that are emulated in the KVM kernel, whereas heavyweight traps are directed to the QEMU for emulation. An SWI instruction is inserted before each sensitive nonprivileged instruction in the assembler code to generate a trap to the hypervisor. ARMvisor maintains two SPTs (kernel SPT and user SPT) to manage page table permissions and avoid TLB flushing on a context switch between kernel and user space. However, maintenance of two SPTs adds complexity to the page table synchronization process. ARM DACR are utilized to tag page table entries with guest user space and guest kernel space. Page table entries are marked as write protected to trap modifications for the SPT in the hypervisor. The number of trap and emulate instructions is reduced by replacing instructions that require only read/write access without privilege emulation with branch instructions that do not trap to the hypervisor. Shadow register file (SPF) space is allocated in the memory for storage of shadow registers required for translation of trap-and-emulate instructions to branch instructions. Figure 7 shows the generic structure of a lightweight paravirtualization solution.

4.3. Microkernel-Based Solution

4.3.1. OKL4. Heiser [2007] introduced the microkernel technology for mobile virtualization based on the L4 microkernel family. The microkernel approach reduces the kernel size by moving basic kernel services out of the kernel space and implements them as servers in the user space. These services communicate with the microkernel through an IPC mechanism. High-bandwidth communication channels are formed between subsystems by setting up shared address mappings. The basic goal of the microkernel approach is to reduce TCB for formal security verification and to provide a general kernel space by separating the kernel policy from mechanisms. The principle of separation helps in virtualizing hardware where a privileged microkernel acts as

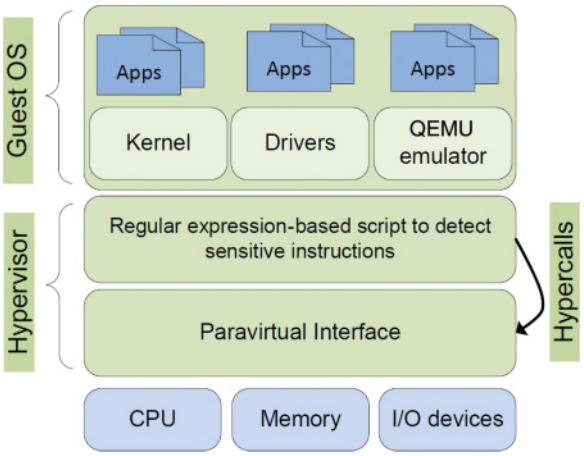

Fig. 7. Lightweight paravirtual mobile hypervisor.

a hypervisor and communicates with the guests utilizing IPC. The OKL4 microkernel provides a paravirtualized HAL for the Linux guest. Guests are multiplexed on the same processor in abstract execution time as threads [Heiser and Leslie 2010]. Systems calls and interrupts when received at the microkernel are transformed into IPC messages that are efficiently redirected to the guest OS or its application. Device drivers are implemented as separate user space processes that communicate through an IPC mechanism. To ensure system security, the OKL4 microkernel implements a policy module that defines which devices are accessible from guests and who can communicate with whom. The performance of the IPC mechanism that lies at the core of the microkernel architecture defines the efficiency of the microkernel virtualization solution.

4.3.2. Thin Hypervisor. The thin hypervisor is a low-footprint mobile virtualization solution that paravirtualizes a single FreeRTOS guest [Douglas 2010]. The thin hypervisor is based on the ARMv5 architecture simulated on the open virtual platform [OpenVZ 2015]. FreeRTOS is paravirtualized such that it consists of the FreeRTOS core functionality and ARMv5 ISA–dependent functionality to implement a hypercall mechanism for sensitive nonprivileged instructions. The FreeRTOS guest does not maintain virtual memory, and each task is dynamically allocated memory from a heap of blocks. This approach frees the thin hypervisor from SPT management while the hypervisor itself resides in the highest 32MB address space. The hypervisor hypercall interface implements 10 hypercalls for ARMv5 ISA dependent code, interrupt handling, and MMU wrappers. Kernel memory protection is provided by wrapper functions that hide kernel API behind a wrapper interface. The wrapper interface manages access to the kernel API by user tasks. The hypervisor memory is protected through hypervisor controlled MMU. Researchers optimized the thin hypervisor assembly code base for a reduced footprint and implemented an application for security critical tasks [Do 2011]. Vahidi and Ekdahl [2013] enhanced the thin hypervisor to virtualize an ARMv7 TrustZone-based device is such a way that it provides a global platform trusted execution environment (TEE) that can host a trusted application (TA). To virtualize ARMv7, the device BIOS is programmed to boot hypervisor in the secure world. The client application can access the security features provided by the TA residing inside the secure world through a remote procedure call (RPC) mechanism.

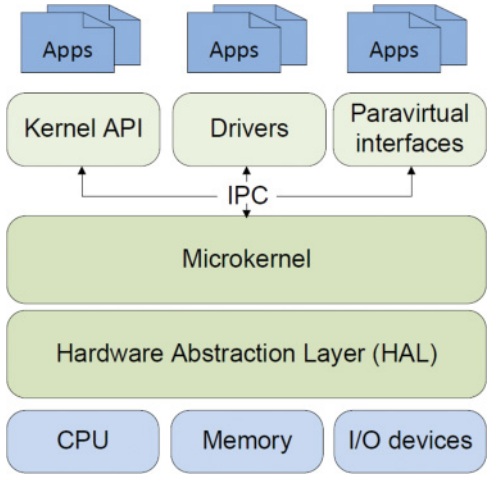

Fig. 8. Microkernel-based mobile hypervisor.

4.3.3. Virtual Hellfire Hypervisor. The MIPS architecture had no native support for virtualization due to the presence of sensitive instructions and a single privileged processing mode. Therefore, MIPS suffered from the same anomaly as ARM mobile virtualization solutions. However, hardware assists for virtualization were announced for MIPS in 2013 [Aguiar 2014]. The virtual Hellfire hypervisor (VHH) is based on an earlier MIPS architecture that does not provide hardware assists for virtualization. VHH extends both microkernel-based paravirtualization and FPGA-based full virtualization support for virtualization of the MIPS architecture [Aguiar and Hessel 2011; Aguiar 2014]. For paravirtualization of the MIPS ISA, the VHH presents a generic virtualization architecture based on the Hellfire OS. A HAL defines the basic hardware functionality over which the paravirtual microkernel resides. The guest OS is paravirtualized to replace sensitive instructions with hypercalls to a HAL. Fixed memory partitions are allocated to each virtual domain by the hypervisor on system boot. I/O devices are managed in a specific virtual domain making them inaccessible from other domains. A set of modules are defined in the VHH that carry out tasks such as domain scheduling, interrupt manager, and clock manager. The intracluster communication between CPUs in the same cluster is done through hypercall interface while setting up shared memory regions among clusters. Intercluster communication over the network-on-chip (NoC) architecture is done via wrapper functions. VHH was extended to support full virtualization by FPGA-based hardware modifications that include new instructions to replace sensitive nonprivileged instructions [Aguiar 2014]. Figure 8 shows the structure of a microkernel-based mobile hypervisor [Heiser 2007].

4.4. Hardware-Assisted Full Virtualization Solution

4.4.1. Hardware-Assisted OKL4. With the advent of ARMv7 hardware extensions for virtualization, many mobile virtualization solutions were updated to utilize these extensions for efficient virtualization. Varanasi and Heiser [2011] and Varanasi [2010] updated the OKL4 microkernel for hardware-assisted mobile virtualization. The OKL4-based hardware-assisted full virtualization solutions supports only two guest OSs (Linux) with static address space for the hypervisor. The solution provides an asynchronous inter-VM communication (IVC) mechanism utilizing virtual registers and shared memory buffers that are set up during device boot. The sender copies message to the receiver virtual register and raises a virtual interrupt for the receiver guest OS.

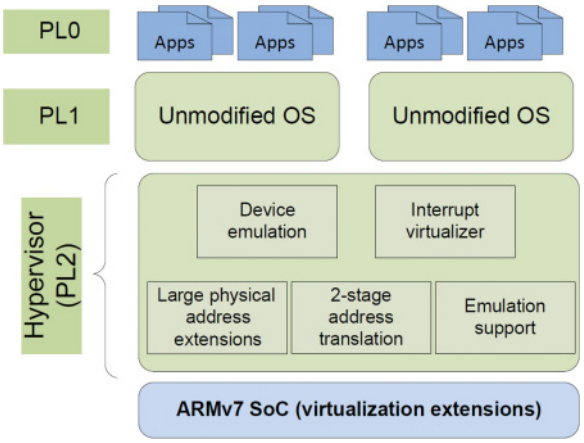

Fig. 9. Hardware-assisted mobile virtualization.

The solution implements fixed VM scheduling with no guest priority. The ARMv7 ISA provides enough information for the hypervisor to emulate any sensitive instruction when trapped other than the write-back instruction, which adds an immediate value to the address register of load instruction. Additional code is added to the hypervisor to support emulation of the write-back instruction. Device pass through allows a guest OS to own a device and does not have virtualization overhead if device interrupts are forwarded to the device owner and proper memory regions are mapped in guest interrupt handlers. Shared devices such as the universal asynchronous receiver transmitter (UART), GIC, and the timer are hosted in a separate VM and can be accessed via an IVC mechanism. The solution lacks support for VM priority scheduling, dynamic VM creation, and multicore utilization. The OKL4 pico version was modified for the hardware-supported full virtualization on ARM and simulated on the ARM Fast Model simulator [ARM 2015]. A hardware-based full virtualization solution is depicted in Figure 9.

4.4.2. KVM/ARM. Dall and Nieh [2014] modified the KVM for ARM solution to implement a hardware-assisted full virtualization KVM/ARM solution. However, their approach was architecturally different from the earlier KVM for ARM solution. KVM/ARM-implemented hardware-assisted full virtualization on the ARMv7 multicore architecture with split-mode hypervisor and merged it into open source mainline Linux 3.9 kernel. Split-mode virtualization allows the Linux-based KVM hypervisor to be split in hypervisor mode (lowvisor) and kernel mode (highvisor) to leverage a virtualization extension in the hypervisor mode and coexist with other CPU modes for basic kernel services. The lowvisor configures hardware to protect different executing domains, switches the executing domains with world switches, and handles interrupts that trap to the hypervisor. The highvisor provides default kernel services such as memory allocation, page table translation, and instruction emulation. However, the split-mode virtualization adds complexity to the hypervisor design. The cost of a context switch due to a trap doubles in split-mode virtualization due to an additional context switch between the lowvisor and highvisor. A trap to the hypervisor from a guest OS firsts traps into the lowvisor and then is routed to the highvisor. The highvisor maintains the lowvisor address space that requires shadowing of lowvisor page tables. Lazy context switching is employed to decrease the performance overhead of virtualization. If a switching VM does not access a particular register, its context is not switched until it is accessed. Scheduling a guest for execution requires the following from the

highvisor (1) store all highvisor context and configurations on hypervisor mode stack; (2) load guest configuration onto the hardware and configure devices (VGIC, timer, etc.) for the guest; (3) configure hypervisor mode to trap on interrupt, `SMC, WFI, WFE` instructions, and access to configuration registers. Lazy traps are for devices that are less likely to be accessed from the guest; (4) enable stage-2 translation and load VM-specific page table entries to the SPT; and (5) trap into the user or kernel mode. The guest will trap to the lowvisor on occurrence of an event, such as hardware interrupt or stage-2 page fault. KVM/ARM uses stage-2 address translation to limit guest access to specific memory regions. KVM/ARM utilizes QEMU for user space device emulation. The I/O mechanism is based on load/store instructions on MMIO regions that are inaccessible from any guest. Therefore, such instructions trap to the hypervisor that routes the access to a specific QEMU-emulated device based on the MMIO address. KVM/ARM utilizes the VGIC to inject virtual interrupts to the guest OSs. KVM/ARM emulates the GIC distributor in the highvisor to trap interrupts from emulated devices.

4.4.3. KVM for Intel Atom. Intel Atom processors come with embedded hardware support for virtualization (Intel VT) that allows hosting of hardware-accelerated VMs on mobile devices. Mobile virtualization based on Intel Atom processors has many advantages over other mobile processors, such as (1) efficient virtualization solutions built on hardware virtualization support, which has matured over the past decade, and (2) virtualization solutions supporting VM migrations for energy efficiency and computational offloading [Khan et al. 2014; Sud et al. 2012]. Sud et al. [2012] proposed a KVM-based mobile virtualization solution for computational offloading from resource-constrained mobile devices to resource-rich servers. KVM for Intel Atom x86 architecture is built upon hardware extensions for virtualization and has a performance advantage over paravirtual and trap-and-emulate approaches. Moreover, KVM also supports live migration of guest OSs. Furthermore, KVM for the Intel x86 has been enhanced to provide real-time scheduling and low-latency virtualization overhead using VM priority scheduling [Zuo et al. 2010].

4.5. DBT-Based Full Virtualization

4.5.1. ViMo. Researchers have implemented a DBT-based full mobile virtualization solution for ARMv6 architecture called *ViMo* [Oh et al. 2010]. ViMo implements various modules in the hypervisor for virtualization of multiple guests. The code tracer scans the guest code until a critical instruction is found. The traced code until the critical instruction is stored in a cache as a basic block. The critical instruction in the basic block is replaced with an encoded SWI instruction with emulation information. ViMo assigns each guest a fixed contiguous memory partition and schedules guests on a round-robin basis. The interrupt controller and other devices are emulated in the hypervisor for guests. The dynamic code translation process scans, translates, and stores each instruction that leads to a performance overhead that is prohibitive for resource-constrained mobile devices. Figure 10 illustrates a DBT-based full virtualization solution.

4.6. Type-2 Virtualization Solutions

4.6.1. VMware Mobile Virtualization Platform. The VMware mobile virtualization platform (MVP) [Barr et al. 2010] is a type-2 mobile virtualization technique that provides an end-to-end enterprise-level mobile virtualization solution for the BYOD use case. MVP allows hosting of a single guest OS over a host through type-2 lightweight paravirtualization of the ARMv7 ISA. A MVP daemon (mvpd) runs as a superuser in the host OS and manages MVP process capabilities. A kernel module mvpkm manages authentication, execution, and world switches for the guest OS. Device access is provided by paravirtualized guest drivers and guest hypercalls that are intercepted by the MVP

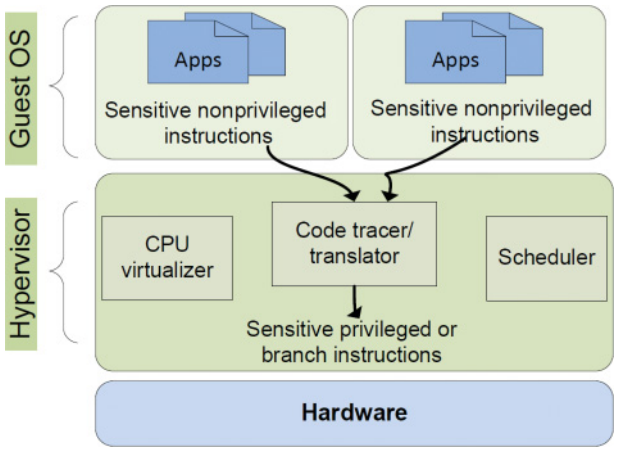

Fig. 10. DBT-based full virtualization.

hypervisor. Sensitive instructions are paravirtualized with 1:1 hypercalls or in-place instructions for emulation. Memory virtualization is supported by ASID-tagged page tables and single hardware address translations. SPTs are maintained by the VMM for guest virtual address to guest physical address translation. The guest image is stored on the secondary mobile storage due to memory constraints on primary storage. Block-level encryption is applied to the guest image to mitigate physical security threats to secondary storage. The guest OS is provided with paravirtualized networking capabilities that operate at socket level through paravirtualized TCP (PVTCP). The client of PVTCP running in the guest OS intercepts all network service requests made by user applications and proxies them to the offload engine residing in the hypervisor. The offload engine creates a proxy socket for the request and performs the network service on behalf of the application. Socket-level interception avoids the overhead of running through separate network stacks of both the guest and hypervisor. MVP supports hosting of a single Android guest OS.

4.6.2. Cells. Cells [Andrus et al. 2011; Dall et al. 2012] is a type-2 mobile virtualization technique that allows multiple virtual phones (VPs) to coexist in the form of isolated Android environments. A foreground VP has access to the most of hardware resources, such as the GUI, whereas other VPs run in the background. Cells utilizes virtual namespace mechanisms to multiplex resources among VPs. The kernel-level device namespace mechanism allows device drivers and kernel modules to tag data structures and callback functions with corresponding namespace identifiers. When a VP foreground-background change occurs, the callback functions are called that store the context of the foreground VP and load the context of the background VP to be executed. Cells also provides wrapper functions for device drivers that multiplex access to the device and communicate on behalf of the original device driver. To virtualize device configurations and propriety hardware, Cells uses a user-level namespace proxy mechanism. A root namespace that is part of the TCB initializes all VPs on system boot and manages VP configurations. A VP cannot access any data or device outside its namespace, thus ensuring isolation. The Cells architecture defines three types of hardware access: shared access, exclusive access, and no access. Cells multiplexes single SIM using a VoIP server to allocate distinct phone numbers to each VP. Cells applies union for read-only files of all VPs so that multiple VPs can coexist on limited memory space of mobile devices. Cells has the advantage of being lightweight, as a single OS

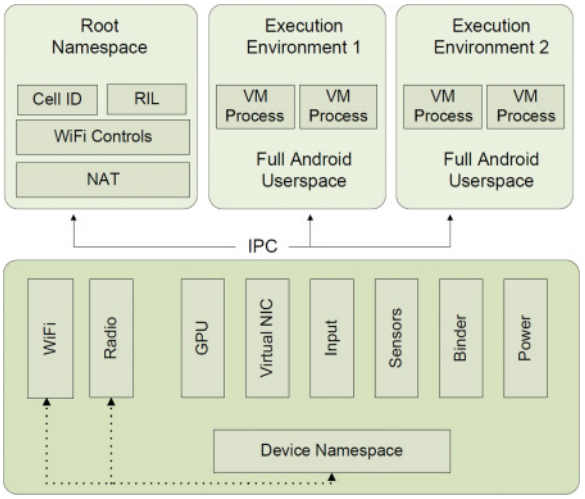

Fig. 11. Type-2 hypervisor–based execution environment.

environment manages multiple Android execution entities. Moreover, the kernel-level namespace mechanism allows virtualization of all hardware resources, unlike type-1 virtualization techniques. However, Cells does not provide a minimal TCB and supports only Android as VP entities. A generic type-2 mobile virtualization solution with execution environments in depicted in Figure 11 [Andrus et al. 2011].

Most mobile virtualization solutions are categorized as type-1 solutions, as the virtualized OSs are hosted directly over the hardware. As the ARM ISA does not fulfill requirements of classic virtualizability, most of the virtualization solutions are paravirtual. However, paravirtualization techniques require modification to guest OS kernel. Lightweight paravirtualization techniques aim to limit guest OS kernel modifications by automatically recognizing sensitive nonprivileged instructions and replacing them with hypercalls. Microkernel-based paravirtualization techniques focus on a small kernel size so that the mobile hypervisor has a smaller memory footprint and can be formally verified for security. On ARM, unmodified guest OSs can be hosted by DBT-based virtualization solutions. However, DBT-based full virtualization solutions have a large memory footprint and are thus performance prohibitive for resource-constrained mobile devices. Since the recent introduction of ARM virtualization extensions, hardware-assisted full virtualization solutions have also been developed. KVM/ARM [Dall and Nieh 2014] and the hardware-assisted OKL4 [Varanasi and Heiser 2011] utilize hardware assists in ARMv7 to efficiently trap and emulate sensitive instructions with lower virtualization overheads. However, there are two major issues in the context of hardware-assisted full virtualization solutions for ARM. First, there is no true full virtualization solution for the ARM ISA, as it contains sensitive instructions that require the trap-and-emulate procedure. Second, the performance advantage of hardware-based mobile virtualization over DBT, paravirtual, and type-2 virtualization solutions has not been investigated so far.

Both Intel and ARM are trying to capture the markets that are uncharacteristic of their architectures. For example, Intel is improving its mobile processors (Intel Atom) with low-density 14nm designs that are ideal for battery-powered mobile and embedded devices [Girbal et al. 2013]. Similarly, ARM is trying to set foot in the server market with 64-bit processors [Smith 2008]. However, there are several trade-offs to the application of ARM processors in the server domain and Intel processors in the mobile space [Aroca

and Gonçalves 2012; Bila et al. 2015]. Although the RISC and CISC debate might never settle down, ISAs, such as Intel Atom, can address issues that have not been properly handled by ARM architecture in mobile devices. For instance, hardware support for virtualization in the Intel x86 architecture is mature compared to the ARM architecture. Therefore, hardware-assisted full virtualization solutions that have low memory and CPU footprint are common for the Intel architecture [Blem et al. 2015]. On the contrary, ARM devices are projected to be utilized in high-performance computing environments. Therefore, ARM virtualization solutions need to embed support for guest OS migration in their designs [Suzuki and Oikawa 2013]. Virtualization solutions for Intel-based processors efficiently support guest OS migration schemes used for resource management in data centers [Ahmad et al. 2015a; Shuja et al. 2014a]. However, due to resource constraints and limited network connectivity, process migration techniques are commonly employed to relocate workloads in mobile devices [Satyanarayanan 2015; Ahmed et al. 2015]. In resource-rich server devices, OS migration strategies are employed to consolidate computations on fewer numbers of devices [Shuja et al. 2014b]. In this context, ARM-based mobile virtualization solutions lag behind Intel counterparts due to immaturity of hardware virtualization assists. Moreover, the energy efficiency and power consumption of mobile and embedded devices is effected due to virtualization. Hosting of multiple OSs can lead to higher resource utilization, thereby increasing the power consumption [Marcu and Tudor 2012].

5. COMPARISON

Based on the details of mobile hypervisors presented in Section 4, it is evident that each solution has different characteristics. Mobile devices are resource constrained, and virtualization leads to performance overhead in any case. As mobile processors based on the ARMv7 ISA are still new in the market, a few mobile virtualization solutions have been able to utilize the hardware virtualization support [Varanasi and Heiser 2011; Dall and Nieh 2014]. Mobile virtualization solutions need to maintain real-time capability of mobile devices while hosting multiple OSs on resource-constrained hardware. The resource constriction, tightly coupled architecture, and real-time capability enforce several informal requirements for the mobile virtualization techniques [Heiser 2007]. In this section, we will compare mobile virtualization solutions based on several important parameters that need to be considered in a mobile hypervisor design. Table V presents the feature comparison of various mobile hypervisors discussed in Section 4.

—*Type*: Type-1 mobile virtualization solutions offer direct command over the hardware to the hypervisor. Alternatively, type-2 hypervisors can execute stripped down guest OSs with low overhead. Paravirtualized solutions offer less complexity in hypervisor design; however, they require a separate patch for each platform-OS combination. Lightweight paravirtualization solutions are less intrusive than conventional paravirtualization solutions but are platform dependent. Microkernel-based solutions offer security and small footprint with minimal kernel approach. However, microkernel-based solutions are still dependent on paravirtualization of the guest OS. Moreover, microkernel-based solutions require an efficient IPC mechanism for real-time capability. Hardware-assisted full virtualization solutions are the most efficient and allow unmodified guest OS and minimal emulation support [Ahmad et al. 2015a].

—*Real-time capability*: Mobile and embedded devices require real-time capability; therefore, they utilize OSs that exhibit real-time capability and are essentially different from general-purpose OSs. The use case of hosting heterogeneous OSs simultaneously over a mobile device requires that the mobile hypervisor maintains real-time capability with bounded interrupt latencies [Armand and Gien 2009]. The resource

Table V. Comparison of ARM Mobile Virtualization Techniques

Hypervisor	VT	RT	Sc		VE	PA	Se	MP	In	VM-M
			M-OS	*M-C*						
Ferstay et al. Xen solution [Ferstay 2006]	Type 1, paravirtual	Low	No, single OS	No	No	ARMv5	No	Yes	High	No
Hwang et al. Xen solution [Hwang et al. 2008]	Type 1, paravirtual	Low	Yes, multiple OSs	No	No	ARMv5	No	Yes	High	No
EmbeddedXEN [Rossier 2012]	Type 1, paravirtual	Low	Yes, two OSs	No	No	ARMv5, 6, 7	Yes	Yes	High	No
ITRI [Smirnov et al. 2013]	Type 1, paravirtual and DBT	Low	Yes, multiple OSs	Yes	No	ARMv7	No	Yes	High	Yes
MobiVMM [Yoo et al. 2008]	Type 1, paravirtual	High	Yes, multiple OSs	No	No	ARMv6	No	Yes	High	No
Proteus [Gilles et al. 2013]	Type 1, paravirtual and full	Medium	Yes, multiple OS	Yes	No	PowerPC	No	Yes	High	No
KVM for ARM [Dall and Nieh 2010]	Type 1, lightweight paravirtual	Low	No, single OS	No	No	ARMv5	No	Yes	Low	No
ARMvisor [Ding et al. 2012]	Type 1, lightweight paravirtual	Low	No, single OS	No	No	ARMv6	No	Yes	Low	No
OKL4 [Heiser 2007]	Type 1, microkernel and paravirtual	Medium	Yes, multiple OSs	No	No	ARMv5	Yes	Yes	High	No
Thin hypervisor [Douglas 2010]	Type 1, microkernel	High	No, single OS	No	No	ARMv5	Yes	Yes	High	No
VHH [Aguiar and Hessel 2011]	Type 1, microkernel and full	High	Yes, multiple OSs	Yes	No	MIPS	No	No	High	No
Hardware-assisted OKL4 [Varanasi and Heiser 2011]	Type 1, hardware-assisted full	Low	Yes, static number of guests	No	Yes	ARMv7	No	No	Low	No
KVM/ARM [Dall and Nieh 2014]	Type 1, hardware-assisted full	Low	Yes, multiple OSs	Yes	Yes	ARMv7	No	Yes	No	No
KVM for Intel Atom [Sud et al. 2012]	Type 1, hardware-assisted full	Medium	Yes, multiple OSs	No	Yes	Intel Atom	No	Yes	No	Yes
ViMo [Oh et al. 2010]	Type 1, DBT-based full	Low	Yes, multiple OSs	No	No	ARMv6	No	Yes	No	No
VMware MVP [Barr et al. 2010]	Type 2, paravirtual	Low	Yes, multiple OSs	Yes	No	ARMv7	Yes	Yes	High	No
Cells [Andrus et al. 2011]	Type 2, execution environment	Medium	Yes, multiple VPs	No	No	ARMv7	Yes	Yes	High	No

VT: Virtualization Technology; RT: Real-time Support; Sc: Scalability; M-OS: Multi-OS; M-C: Multi-Core; VE: Virtualization Extension; PA: Processor Architecture; Se: Security; MP: Memory Protection; In: Intrusiveness; VM-M: VM Migration.

sharing may lead to contention and wait that needs to be prioritized for real-time and performance-critical guests. The measure of real-time capability of a virtualization solution depends on three factors: guest priority scheduling for access to drivers and interrupts, an RTOS patch for paravirtual solutions, and a hypervisor memory and processing footprint. Guest priority scheduling allows an RTOS to have priority access to the CPU and device driver over general-purpose OSs. Paravirtualized solutions also need to provide a patch for an RTOS to support real-time capability in mobile virtualization solutions. Additionally, a hypervisor with a minimal footprint consumes fewer CPU and memory resources. Furthermore, hypervisor efficiency is coupled with its kernel size. Small kernel size results in a lower footprint such that real-time response in mobile devices is not degraded [Do 2011].

—*Scalability*: As MPSoC mobile devices are making their way to the market, mobile hypervisors need to be scalable to both the number of guests and cores in a MPSoC design. However, most mobile hypervisors are not multicore capable [Varanasi and Heiser 2011; Barbalace et al. 2015]. Moreover, some mobile virtualization solutions host a single OS [Ferstay 2006], a design that compromises the basic virtualization objective. Scalable mobile virtualization solutions are also necessary to support

server virtualization use cases, such as resource and workload consolidation [Jones 2011; Jararweh et al. 2014].

—*Utilization of virtualization extensions*: The efficiency of a mobile hypervisor largely depends on utilization of virtualization extensions. The ARMv7 ISA provides hardware virtualization extensions that help limit the complexity of the hypervisor by providing hardware support for processor, memory, I/O, and interrupt virtualization. Mobile hypervisor solutions based on previous versions of the ARM ISA need to be updated to utilize the hardware extensions and reduce the software complexity of virtualization. For example, as the ARMv7 ISA provides hardware-based two-stage address translation, removal of code that maintained the SPTs and the second stage of software-based address translation can lead to considerable downsize in the code base [Brash 2010].

—*Processor architecture*: Hardware assists are necessary for an efficient, low software overhead mobile virtualization solution. Hardware assists in most CICS-based processors, such as ARM and MIPS, are relatively more immature than in x86 processors. Therefore, x86-based processors are ideal candidates for a lower-overhead mobile virtualization solution. However, CISC-based processors dominate the mobile and embedded systems market. Hardware assists in CISC-based processors have been introduced in recent years, and corresponding mobile virtualization solutions have been developed.

—*Security*: The security requirement implies that the hypervisor code base should be protected from unauthorized access from guest OSs. A minimal hypervisor code base helps in formal verification and validation of security [Blackham and Heiser 2012]. Formal verification of security is an essential requirement for embedded devices used in avionics, automobiles, and IP-protected systems. Moreover, the security of the hypervisor is critical, as it is the only software running in privileged processor mode.

—*Memory protection*: As the hypervisor, guest OS, and user applications share the same address space in virtual memory, memory protection mechanism needs to be defined [Brakensiek et al. 2008; Lee and Hsueh 2013]. The ARMv7 ISA allows the page table entries to be tagged with ASID. However, previous ARM architectures required specific memory protection mechanisms, such as fixed memory partitioning and DACR tagging [Aguiar and Hessel 2011].

—*Nonintrusive*: A mobile virtualization solution must require minimal changes to the guest OS and enable reuse of native device drivers and legacy software with no modifications [Heiser 2008]. Pure paravirtualization-based solutions are highly intrusive, whereas lightweight paravirtualization solutions require fewer guest OS modifications. Hardware-assisted full virtualization solutions do not require any guest OS modifications.

—*VM migration*: Energy-efficient ARM-based servers are an alternative to performance-efficient Intel-based servers in data center space. Due to recent energy cost escalations, ARM-based servers are being considered for data center deployment. In such a case, ARM virtualization solutions should support efficient guest migration—a technique that is widely adopted in data centers for resource management and consolidation [Smirnov et al. 2013; Ahmad et al. 2015b].

The complexity of a mobile virtualization solution can be categorized by its type. DBT-based solutions such as ViMo [Oh et al. 2010] are the most complex, as they require the hypervisor to scan, classify, and replace sensitive instructions at runtime. On the other hand, hardware-assisted full virtualization solutions, namely KVM/ARM [Dall and Nieh 2014] and KVM for Intel Atom [Sud et al. 2012], have less software complexity due to utilization of virtualization extensions. However, it is advocated that

the complexity transfers from the hypervisor software to the underlying hardware architecture for hardware-assisted full virtualization solutions [Heiser and Leslie 2010]. Paravirtualized solutions require modifications to the guest OS binary. Not all OSs can be paravirtualized due to their propriety nature and closed source code. Therefore, paravirtualized solutions are less flexible in terms of hosting multiple heterogeneous OSs [Penneman et al. 2013].

Real-time capability of the virtualization solutions is categorized from high to low based on factors such as RTOS patch for paravirtualization, RTOS priority scheduling over GPOS, and minimal hypervisor footprint for efficient virtualization. MobiVMM [Yoo et al. 2008], the thin hypervisor [Douglas 2010], and VHH [Aguiar and Hessel 2011] support two of the aforementioned factors while providing real-time priority scheduling of guest OSs and minimal virtualization overhead. Therefore, real-time capability of these virtualization solutions is high. OKL4 [Heiser 2007] and Cells [Andrus et al. 2011] support minimal hypervisor footprint, whereas KVM for Intel Atom [Sud et al. 2012] and Proteus [Gilles et al. 2013] provide priority scheduling of RTOS. Therefore, their real-time capability is medium. Other virtualization solutions lack real-time features. ITRI [Smirnov et al. 2013], VHH [Aguiar and Hessel 2011], KVM/ARM [Dall and Nieh 2014], VMware MVP [Barr et al. 2010], KVM for Intel Atom [Sud et al. 2012], and Proteus [Gilles et al. 2013] are truly scalable mobile hypervisors with both multicore and multiguest support. Hypervisors such as ARMvisor [Ding et al. 2012] do not support hosting multiple OSs and MPSoC architectures. Alternatively, the EmbeddedXEN [Rossier 2012] virtualization solution does not provide true scalability by limiting the number of hosted OSs. Hardware virtualization extensions for various mobile processors, namely ARM and PowerPC, are not yet mature [Suzuki and Oikawa 2013]. Therefore, most mobile virtualization solutions are not built upon hardware virtualization extensions. However, the hardware-assisted OKL4 [Varanasi and Heiser 2011], KVM/ARM [Dall and Nieh 2014], and KVM for Intel Atom [Sud et al. 2012] have a performance advantage over other mobile virtualization solutions, as they utilize the hardware virtualization extensions.

Security is an important consideration in the design of mobile hypervisors. Mobile virtualization solutions except for OKL4 [Heiser 2007] have not provided formal verification of hypervisor security. EmbeddedXEN [Rossier 2012] provides high hypervisor security, as the guest OS binary is preappended with hypervisor code. However, such a design cannot scale the number of guest OSs. The VMware MVP [Barr et al. 2010] and Cells [Andrus et al. 2011] virtualization solutions define security architectures based on root and virtual space identifiers to isolate root and virtual domains. Similarly, the thin hypervisor [Douglas 2010] provides security measures in terms of encryption schemes and MMU-based access control. Other mobile virtualization solutions provide low security without explicit mechanisms for hypervisor space protection from unauthorized access. Most mobile hypervisors utilize either fixed memory partitioning [Oh et al. 2010] or a DACR bit for memory protection [Hwang et al. 2008; Dall and Nieh 2010]. VHH [Aguiar and Hessel 2011] does not apply any memory protection mechanism. Alternatively, the hardware-assisted OKL4 [Varanasi and Heiser 2011] does not utilize memory virtualization and hosts a static number of guest OSs. KVM/ARM [Dall and Nieh 2014], ViMo [Oh et al. 2010], and KVM for Intel Atom [Sud et al. 2012] are the only mobile hypervisors that do not require modification of guest OS binary, as they provide support for full virtualization. Similarly, KVM for ARM [Dall and Nieh 2010] and ARMvisor [Ding et al. 2012] are slightly intrusive, as they utilize lightweight paravirtualization. The hardware-assisted OKL4 [Varanasi and Heiser 2011] is also lightly intrusive, as it provides hardware-assisted full virtualization with small modification to the guest OS for emulation of write-back instruction. Proteus [Gilles et al.

2013] supports both full- and paravirtual mobile virtualization solutions. The paravirtual solution is intrusive, as it requires patching of the guest OS. The remainder of mobile virtualization solutions are highly intrusive, as they employ pure paravirtualization techniques that require extensive modifications to the guest OS binary. ITRI [Smirnov et al. 2013] and KVM for Intel Atom [Sud et al. 2012] are the only mobile hypervisors that provide the guest OS migration feature. However, the VM migration support in Intel-based processors is mature compared to ARM-based processors due to their usage in cloud data centers [Shuja et al. 2014b].

6. CONCLUSION

In this article, we provided a detailed analysis of software- and hardware-based mobile virtualization solutions. We presented a formal analysis of the ARM architecture and highlighted the issues for an efficient mobile virtualization. A detailed taxonomy of mobile virtualization solutions was also defined. The state of the art in software- and hardware-based techniques for mobile virtualization were presented. We also identified various issues and challenges to virtualization of CPU, memory, interrupt, I/O, and network functionality. We presented an in-depth analysis of various mobile virtualization solutions, and scalability, performance, and security issues of each solution were determined. We presented a comparison of mobile virtualization solutions based on various desirable features. Moreover, we debated performance trade-offs between ARM and other hardware ISA-based mobile virtualization solutions.

The nonvirtualizable ARM architecture results in virtualization solutions that rely heavily on software-based techniques for resource multiplexing. We observed that the cost of implementing software-based virtualization solution that consumes CPU cycles and memory space on resource-constrained mobile devices is always high. Efficient IPC mechanisms for resource sharing need to be developed to lower the overhead of paravirtualization-based solutions. Paravirtualization solutions necessitate the development of standard hypervisor interfaces that dynamically scale to new hardware and software. Moreover, static binary translation solutions that have lower overhead than that of DBT-based solutions must be examined for mobile virtualization. Efficient mobile virtualization solutions have become feasible with the emergence of hardware virtualization extensions that support virtualization tasks in the majority of mobile processors. However, many frontiers must be explored in the field of mobile virtualization. Pure hardware-based mobile virtualization solutions, which utilize virtualization extensions embedded in newer ARM processors, must be developed. Similarly, mobile virtualization solutions with a minimal hypervisor footprint need to be investigated for resource-constrained mobile devices. A quantitative comparison of software and hardware mobile virtualization techniques has not been investigated thus far. Mobile device virtualization demands novel techniques for sharing device components, such as camera and communication interfaces. Furthermore, research challenges remain in the context of embedding network virtualization functionality in wireless access–based mobile devices. Heterogeneous MPSoC architectures are utilized in mobile and embedded devices to benefit from the specific property of each core. However, the virtualization of heterogeneous MPSoC architectures leads to a complex hypervisor design, given that individual properties of each ISA in the MPSoC must be considered. The aforementioned issues need to be tackled with resource-efficient techniques for resource-constrained mobile devices.

REFERENCES

Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for X86 virtualization. *ACM Sigplan Notices* 41, 11, 2–13.

Alexandra Aguiar and Fabiano Hessel. 2010. Embedded systems' virtualization: The next challenge? In *Proceedings of the 21st IEEE International Symposium on Rapid System Prototyping (RSP'10)*. IEEE, Los Alamitos, CA, 1–7.

Alexandra Aguiar and Fabiano Hessel. 2011. Virtual Hellfire hypervisor: Extending Hellfire framework for embedded virtualization support. In *Proceedings of the 12th International Symposium on Quality Electronic Design (ISQED'11)*. 1–8.

Alexandra da Costa Pinto de Aguiar. 2014. *On the Virtualization of Multiprocessed Embedded Systems*. Ph.D. Dissertation. Pontifícia Universidade Católica do Rio Grande do Sul.

Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab Hamid, Muhammad Shiraz, Abdullah Yousafzai, and Feng Xia. 2015a. A survey on virtual machine migration and server consolidation frameworks for cloud data centers. *Journal of Network and Computer Applications* 52, 11–25.

Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab. Hamid, Feng Xia, and Muhammad Shiraz. 2015b. A review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues. *Journal of Network and Computer Applications* 58, 42–59.

Ejaz Ahmed, Abdullah Gani, Mehdi Sookhak, Siti Hafizah Ab Hamid, and Feng Xia. 2015. Application optimization in mobile cloud computing: Motivation, taxonomies, and open challenges. *Journal of Network and Computer Applications* 52, 52–68.

Adnan Akhunzada, Ejaz Ahmed, Abdullah Gani, Muhammad Khan, Muhammad Imran, and Sghaier Guizani. 2015. Securing software defined networks: Taxonomy, requirements, and open issues. *IEEE Communications Magazine* 53, 4, 36–44.

Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. 2011. Cells: A virtual mobile smartphone architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 173–187.

ARM. 2015. Fast Models. Available at https://silver.arm.com/browse/FM000.

ARM Architecture Group. 2009. *ARM Security Technology: Building a Secure System Using TrustZone Technology*. Technical Report. ARM.

ARM Architecture Group. 2010a. *ARM Generic Interrupt Controller Architecture Specification 2.0*. Technical Report. ARM Limited. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0048b/index.html.

ARM Architecture Group. 2010b. *ARM Generic Timer Specification*. Technical Report. ARM Limited. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438c/BGBBIJCB.html.

ARM Architecture Group. 2014a. *ARM Architecture Reference Manual* (armv7-a and armv7-r ed.). ARM.

ARM Architecture Group. 2014b. TrustZone. Retrieved March 3, 2016, from http://www.arm.com/products/processors/technologies/trustzone/index.php.

ARM Limited. 2011. *Cortex-A15 Technical Reference Manual*. ARM Limited. http://www.arm.com/products/processors/cortex-a/cortex-a15.php.

François Armand and Michel Gien. 2009. A practical look at micro-kernels and virtual machine monitors. In *Proceedings of the 6th IEEE Consumer Communications and Networking Conference (CCNC'09)*. 1–7.

Rafael Vidal Aroca and Luiz Marcos Garcia Gonçalves. 2012. Towards green data centers: A comparison of X86 and arm architectures power efficiency. *Journal of Parallel and Distributed Computing* 72, 12, 1770–1780.

Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the 10th European Conference on Computer Systems*. ACM, New York, NY, 29.

Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. 2010. The VMware mobile virtualization platform: Is that a hypervisor in your pocket? *ACM SIGOPS Operating Systems Review* 44, 4, 124–135.

Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 41–46.

Cory Bialowas. 2010. *Achieving Business Goals with Wind River's Multicore Software Solution*. Technical Report.

Nilton Bila, Eric J. Wright, Eyal De Lara, Kaustubh Joshi, H. Andrés Lagar-Cavilla, Eunbyung Park, Ashvin Goel, Matti Hiltunen, and Mahadev Satyanarayanan. 2015. Energy-oriented partial desktop virtual machine migration. *ACM Transactions on Computer Systems* 33, 1, Article No. 2. DOI:http://dx.doi.org/10.1145/2699683

K. Bilal, S. U. R. Malik, S. U. Khan, and A. Y. Zomaya. 2014. Trends and challenges in cloud data centers. *IEEE Cloud Computing Magazine* 1, 1, 10–20.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2, 1–7.

Bernard Blackham and Gernot Heiser. 2012. Correct, fast, maintainable: Choose any three! In *Proceedings of the Asia-Pacific Workshop on Systems*. 13.

Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan, and Karthikeyan Sankaralingam. 2015. ISA wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures. *ACM Transactions on Computer Systems* 33, 1, 3.

Daniele Bortolotti, Christian Pinto, Andrea Marongiu, Martino Ruggiero, and Luca Benini. 2013. VirtualSoC: A full-system simulation environment for massively parallel heterogeneous system-on-chip. In *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. 2182–2187.

Jörg Brakensiek, Axel Dröge, Martin Botteck, Hermann Härtig, and Adam Lackorzynski. 2008. Virtualization as an enabler for security in mobile devices. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*. 17–22.

David Brash. 2010. Extensions to the ARMv7-A architecture. In *Hot Chips*, Vol. 22. ARM.

Brian Carlson. 2011. *Going Beyond a Faster Horse to Transform Mobile Devices*. Technical Report. Texas Instruments.

Wenzhi Chen, Lei Xu, Guoxi Li, and Yang Xiang. 2015. A lightweight virtualization solution for Android devices. *IEEE Transactions on Computers* 64, 10, 2741–2751. DOI:http://dx.doi.org/10.1109/TC.2015.2389791

Xiaoyi Chen. 2011. Smartphone virtualization: Status and challenges. In *Proceedings of the International Conference on Electronics, Communications, and Control (ICECC'11)*. IEEE, Los Alamitos, CA, 2834–2839.

Christoffer Dall, Jeremy Andrus, Alexander Vant Hof, Oren Laadan, and Jason Nieh. 2012. The design, implementation, and evaluation of cells: A virtual smartphone architecture. *ACM Transactions on Computer Systems* 30, 3, 9.

Christoffer Dall and Jason Nieh. 2010. KVM for ARM. In *Proceedings of the Ottawa Linux Symposium*. 45–56.

Christoffer Dall and Jason Nieh. 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*. 333–348.

Jiun-Hung Ding, Roger Chien, Shih-Hao Hung, Yi-Lan Lin, Che-Yang Kuo, Ching-Hsien Hsu, and Yeh-Ching Chung. 2014. A framework of cloud-based virtual phones for secure intelligent information management. *International Journal of Information Management* 34, 3, 329–335.

Jiun-Hung Ding, Chang-Jung Lin, Ping-Hao Chang, Chieh-Hao Tsang, Wei-Chung Hsu, and Yeh-Ching Chung. 2012. ARMvisor: System virtualization for ARM. In *Proceedings of the Ottawa Linux Symposium (OLS'12)*. 93–107.

Viktor Do. 2011. *Security Services on an Optimized Thin Hypervisor for Embedded Systems*. Ph.D. Dissertation. Lund University.

YaoZu Dong, JunJie Mao, HaiBing Guan, Jian Li, and Yu Chen. 2015. A virtualization solution for BYOD with dynamic platform context switching. *IEEE Micro* 35, 1, 34–43.

Heradon Douglas. 2010. *Thin Hypervisor-Based Security Architectures for Embedded Platforms*. Ph.D. Dissertation. Royal Institute of Technology.

Daniel R. Ferstay. 2006. *Fast Secure Virtualization for the Arm Platform*. Ph.D. Dissertation. University of British Columbia.

Justin M. Forbes. 2007. Why virtualization fragmentation sucks. In *Proceedings of the Linux Symposium*, Vol. 1. 125–130.

Thomas Gaska, Brian Werner, and David Flagg. 2010. Applying virtualization to avionics systems—the integration challenges. In *Proceedings of the IEEE/AIAA 29th Digital Avionics Systems Conference (DASC'10)*. 5.E.1-1–5.E.1-19.

Katharina Gilles, Stefan Groesbrink, Daniel Baldin, and Timo Kerstan. 2013. Proteus hypervisor: Full virtualization and paravirtualization for multi-core embedded systems. In *Embedded Systems: Design, Analysis and Verification*. Springer, 293–305.

Sylvain Girbal, Miquel Moretó, Arnaud Grasset, Jaume Abella, Eduardo Quinones, Francisco J. Cazorla, and Sami Yehia. 2013. The next convergence: High-performance and mission-critical markets. In *Proceedings*

of the 8th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC'13). 1–11.

John Goodacre. 2011. Hardware accelerated virtualization in the ARM Cortex processors. In *Proceedings of XenSummit Asia*.

John Goodacre. 2013. The evolution of the ARM architecture towards big data and the data-centre (abstract only). In *Proceedings of the 8th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'13)*. ACM, New York, NY, Article No. 4. http://doi.acm.org/10.1145/2535800.2535921

John Goodacre and Andrew N. Sloss. 2005. Parallelism and the ARM instruction set architecture. *Computer* 38, 7, 42–50.

Green Hills Software. 2013. *Integrity Multivisor: Secure Virtualization for ARM*. Technical Report. Green Hills.

Yan Grunenberger, Ilenia Tinnirello, Pierluigi Gallo, Eduard Goma, and Giuseppe Bianchi. 2012. Wireless card virtualization: From virtual NICs to virtual MAC machines. In *Proceedings of the Future Network and Mobile Summit (FutureNetw'12)*. 1–10.

Zonghua Gu and Qingling Zhao. 2012. A state-of-the-art survey on real-time issues in embedded systems virtualization. *Journal of Software Engineering and Applications* 5, 4, 277–290.

Kevin Gudeth, Matthew Pirretti, Katrin Hoeper, and Ron Buskey. 2011. Delivering secure applications on commercial mobile devices: The case for bare metal hypervisors. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. 33–38.

Gernot Heiser. 2007. *Virtualization for Embedded Systems*. Technical Report. Open Kernel Labs.

Gernot Heiser. 2008. The role of virtualization in embedded systems. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*. 11–16.

Gernot Heiser. 2009a. Hypervisors for consumer electronics. In *Proceedings of the 6th IEEE Consumer Communications and Networking Conference (CCNC'09)*. 1–5.

Gernot Heiser. 2009b. *The Motorola Evoke QA4-A Case Study in Mobile Virtualization*. Technical Report. Open Kernel Labs.

Gernot Heiser. 2011. Virtualizing embedded systems: Why bother? In *Proceedings of the 48th Design Automation Conference*. 901–905.

Gernot Heiser and Ben Leslie. 2010. The OKL4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the 1st ACM Asia-Pacific Workshop on Workshop on Systems*. ACM, New York, NY, 19–24.

Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. 2008. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *Proceedings of the 5th IEEE Consumer Communications and Networking Conference*. 257–261.

Hiroaki Inoue, Junji Sakai, and Masato Edahiro. 2008. Processor virtualization for secure mobile terminals. *ACM Transactions on Design Automation of Electronic Systems* 13, 3, 48.

Yaser Jararweh, Lo'ai Tawalbeh, Fadi Ababneh, Abdallah Khreishah, and Fahd Dosari. 2014. Scalable cloudlet-based mobile computing model. *Procedia Computer Science* 34, 434–441.

M. Tim Jones. 2011. *Virtualization for Embedded Systems*. White Paper. International Business Machines Corporation, Armonk, NY.

A. Khan, M. Othman, S. Madani, and S. Khan. 2014. A survey of mobile cloud computing application models. *IEEE Communications Surveys and Tutorials* 16, 1, 393–413.

Travis Lanier. 2011. *Exploring the Design of the Cortex-a15 Processor*. Technical Report. ARM.

Kihong Lee, Dongwoo Lee, and Young Ik Eom. 2015. Power-efficient and high-performance block I/O framework for mobile virtualization systems. In *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*. ACM, New York, NY, 57.

Yuan-Cheng Lee and Chih-Wen Hsueh. 2013. An optimized page translation for mobile virtualization. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, New York, NY, 85.

Joshua LeVasseur, Volkmar Uhlig, Yaowei Yang, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. 2008. Pre-virtualization: Soft layering for virtual machines. In *Proceedings of the 13th Asia-Pacific Computer Systems Architecture Conference (ACSAC'08)*. 1–9.

Wenhao Li, Liang Liang, Mingyang Ma, Yubin Xia, and Haibo Chen. 2015. Poster: TVisor—a practical and lightweight mobile red-green dual-OS architecture. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, 485–485.

Chengchao Liang and F. Richard Yu. 2015. Wireless network virtualization: A survey, some research issues and challenges. *IEEE Communications Surveys and Tutorials* 17, 1, 358–380.

Marius Marcu and Dacian Tudor. 2012. Energy efficiency measurements of mobile virtualization systems. In *Security and Privacy in Mobile Information and Communication Systems*. Springer, 88–100.

Roberto Mijat and Andy Nightingale. 2011. *Virtualization Is Coming to a Platform Near You*. White Paper. ARM.

Aashish Mittal, Dushyant Bansal, Sorav Bansal, and Varun Sethi. 2013. Efficient virtualization on embedded power architecture® platforms. *ACM SIGPLAN Notices* 48, 4, 445–458.

Saad Mustafa, Babar Nazir, Amir Hayat, Atta ur Rehman Khan, and Sajjad A. Madani. 2015. Resource management in cloud computing: Taxonomy, prospects, and challenges. *Computers and Electrical Engineering*, 186–203.

Seyed Hossein Nikounia and Siamak Mohammadi. 2015. Gem5v: A modified gem5 for simulating virtualized systems. *Journal of Supercomputing* 71, 4, 1484–1504.

Soo-Cheol Oh, KangHo Kim, KwangWon Koh, and Chang-Won Ahn. 2010. ViMo (virtualization for mobile): A virtual machine monitor supporting full virtualization for ARM mobile systems. In *Proceedings of the 1st International Conference on Cloud Computing, GRIDs, and Virtualization*. 48–53.

OpenVZ. 2015. OpenVZ Wiki Main Page. Retrieved March 3, 2016, from http://wiki.openvz.org/Main_Page.

Sixto Ortiz. 2011. Chipmakers ARM for battle in traditional computing market. *Computer* 44, 4, 14–17.

Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. 2013. Formal virtualization requirements for the ARM architecture. *Journal of Systems Architecture* 59, 3, 144–154.

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

PowerPC. 2009. *Hardware and Software Assists in Virtualization*. Technical Report. Freescale Semiconductor.

Nikola Rajovic, Alejandro Rico, Nikola Puzovic, Chris Adeniyi-Jones, and Alex Ramirez. 2014. Tibidabo: Making the case for an ARM-based HPC system. *Future Generation Computer Systems* 36, 322–334.

Rahul Ramasubramanian. 2011. *Exploring Virtualization Platforms for ARM Based Mobile Android Devices*. Master's Thesis. North Carolina State University.

Manoj R. Rege, Vlado Handziski, and Adam Wolisz. 2013. Crowdmeter: An emulation platform for performance evaluation of crowd-sensing applications. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*. ACM, New York, NY, 1111–1122.

Daniel Rossier. 2012. *EmbeddedXEN: A Revisited Architecture of the XEN Hypervisor to Support ARM-Based Embedded Virtualization*. White Paper. Switzerland.

Rusty Russell. 2008. Virtio: Towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5, 95–103.

Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. 2012. MOSES: Supporting operation modes on smartphones. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*. 3–12.

Mahadev Satyanarayanan. 2015. A brief history of cloud offload: A personal journey from Odyssey through cyber foraging to cloudlets. *ACM SIGMOBILE Mobile Computing and Communications Review* 18, 4, 19–23.

Ryan Shea and Jiangchuan Liu. 2012. Network interface virtualization: Challenges and solutions. *IEEE Network* 26, 5, 28–34.

J. Shuja, K. Bilal, S. A. Madani, M. Othman, R. Ranjan, P. Balaji, and S. U. Khan. 2014b. Survey of techniques and architectures for designing energy-efficient data centers. *IEEE Systems Journal* PP, 99, 1–13. DOI: http://dx.doi.org/10.1109/JSYST.2014.2315823

Junaid Shuja, Kashif Bilal, Sajjad Ahmad Madani, and Samee U. Khan. 2014a. Data center energy efficient resource scheduling. *Cluster Computing* 17, 4, 1265–1277.

Alexey Smirnov, Mikhail Zhidko, Yingshiuan Pan, Po-Jui Tsao, Kuang-Chih Liu, and Tzi-Cker Chiueh. 2013. Evaluation of a server-grade software-only ARM hypervisor. In *Proceedings of the IEEE 6th International Conference on Cloud Computing*. 855–862.

Brad Smith. 2008. ARM and Intel battle over the mobile chip's future. *Computer* 41, 5, 15–18.

Shivani Sud, Roy Want, Trevor Pering, Kent Lyons, Barbara Rosario, and Michelle X. Gong. 2012. Dynamic migration of computation through virtualization of the mobile platform. *Mobile Networks and Applications* 17, 2, 206–215.

Sang-Bum Suh. 2007. Secure architecture and implementation of Xen on ARM for mobile devices. In *Proceedings of the4th Xen Summit.*

Akihiro Suzuki and Shuichi Oikawa. 2013. Analysis of the ARM architecture's ability to support a virtual machine monitor through a simple implementation. *International Journal of Networking and Computing* 3, 1, 153.

Ehsan Totoni, Babak Behzad, Swapnil Ghike, and Josep Torrellas. 2012. Comparing the power and performance of Intel's SCC to state-of-the-art CPUs and GPUs. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'12)*. IEEE, Los Alamitos, CA, 78–87.

Arash Vahidi and Patrik Ekdahl. 2013. *VETE: Virtualizing the Trusted Execution Environment*. Technical Report. SICS.

Prashant Varanasi. 2010. *Implementing Hardware-Supported Virtualization in OKL4 on ARM*. Ph.D. Dissertation. University of New South Wales.

Prashant Varanasi and Gernot Heiser. 2011. Hardware-supported virtualization on ARM. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*. 11.

vIrtical Project. 2013. *Hypervisor for ARM A15 and GPPA*. Technical Report.

Bi Wu. 2013. *Virtualization with Limited Hardware Support*. Ph.D. Dissertation. Duke University.

Lei Xia, Sanjay Kumar, Xue Yang, Praveen Gopalakrishnan, York Liu, Sebastian Schoenberg, and Xingang Guo. 2011. Virtual WiFi: Bring virtualization from wired to wireless. *ACM SIGPLAN Notices* 46, 7, 181–192.

Seehwan Yoo, Yunxin Liu, Cheol-Ho Hong, Chuck Yoo, and Yongguang Zhang. 2008. MobiVMM: A virtual machine monitor for mobile phones. In *Proceedings of the 1st Workshop on Virtualization in Mobile Computing*. 1–5.

Seehwan Yoo, Sung-Bae Yoo, and Chuck Yoo. 2013. Virtualizing ARM VFP (vector floating-point) with Xen-ARM. *Journal of Systems Architecture* 59, 10, 1266–1276.

Nairan Zhang, Parameswaran Ramanathan, Kyu-Han Kim, and Sujata Banerjee. 2012. Powervisor: A battery virtualization scheme for smartphones. In *Proceedings of the 3rd ACM Workshop on Mobile Cloud Computing and Services*. ACM, New York, NY, 37–44.

Baojing Zuo, Kai Chen, Alei Liang, Haibing Guan, Jun Zhang, Ruhui Ma, and Hongbo Yang. 2010. Performance tuning towards a KVM-based low latency virtualization system. In *Proceedings of the 2nd International Conference on Information Engineering and Computer Science (ICIECS'10)*. IEEE, Los Alamitos, CA, 1–4.

Received May 2015; revised November 2015; accepted January 2016