Virtualization for embedded systems

M.Tech. Seminar Report

Sahil Singh
143050001

Seminar Guide
Prof. Purushottam Kulkarni

Department of Computer Science and engineering
Indian Institute of Technology Bombay
Mumbai, Maharashtra

Contents

1 Introduction

Silicon wafer fabrication technology has advanced by leaps and bounds in recent years. This trend, driven by Moore's law, has lead to emergence of new and powerful microprocessors. This, in turn, has fuelled the adoption of virtualization on various platforms, in an effort to make efficient use of these capable processors, and to support a variety of applications on the same platform.

The embedded systems world hasn't remained untouched by these developments, and is witnessing tremendous growth in complexity and variety of applications. Virtualization solutions, however, for embedded platforms differ in their use-cases, and constraints, from their x86 counterparts, and hence the solutions developed for x86 platforms cannot be directly ported.

Intel, with its low power x86 processors has a small and emerging portfolio of processors targeted towards embedded platforms. In the course of my study I mainly focused on x86 server, and desktop processors, therefore, the term x86 mentioned in this document refers to x86 processors for desktops and servers.

1.1 Virtualization

Virtualization, as an idea originated in the 1960s. Then it was used mainly to allow multiple users to access a common, and often expensive computing platform. With the advent of multi-user operating systems, and popularization of desktops, this was no longer needed, and virtualization lost it's significance. During the last few decades, however virtualization has risen again, albeit with a new set of use cases. The following are the main use cases of virtualization, and depending on the application domain, or hardware platform some may be more critical than others (For, eg. a provider of cloud services will be more concerned about the most efficient use of hardware he owns, to maximize profit, while ensuring least discomfort to his customers, whereas for a user running multiple VMs on his/her mobile phone, battery life might be a primary concern) .

- Server consolidation

  This refers to the ability to run multiple VMs on a single physical machine. Virtualization allows to flexibly multiplex hardware resources between the running VMs. This can be used to exploit statistical multiplexing. Statistical multiplexing is a way to overcommit resources, it relies on statistical nature of resource demand by various entities sharing it. For eg. a link of 2mbps can be shared between 10 users, each requiring 2mbps, if it is known that the times at which they try to access the link are mutually exclusive from one another.

- Provision of isolated application containers

  Virtualization can be used to create separate VMs, running their own operating systems, this ensures that a fatal fault in one VM doesn't affect the working of other VMs. This also enhances security, as a malicious applications running in one VM will not be able to harm the applications running in other VMs.

- Aid software development

  Using virtualization an enterprise can test its software on a variety of hardware platforms, under different resource allocations, without ever buying additional hardware. Virtualization allows record, replay, and check pointing which can be used for debugging. Porting legacy applications on new hardware, and providing support for multiple platforms can be done easily using virtualization.

- Desktop virtualization

  Desktop virtualization refers to running the entire desktop environment on a centralized server, while the terminals at the clients only serve as a means to access this desktop environment over a network. This aids maintenance, and saves power.

- Serviceability

  In enterprises running a cluster of servers, hardware failure is quite frequent. Virtualization helps in preventing hardware failure from having an impact on running services, by allowing VM migration. Migration can also be used to put a part of cluster offline for maintenance, without disrupting the services, running on them. This might decrease the performance of these services, as the load on remaining part of cluster will increase, but is still much better than completely shutting down a service.

1.2 Scope of this seminar

In the course of this seminar I studied the unique aspects of virtualization on embedded platforms, majorly focusing on the ARM architecture. I tried to explore the topic from three different viewpoints.

1. Architectural differences from x86

   Due to different application domains, and constraints under which the systems are expected to perform, the architectures for desktop computers and servers have evolved differently from that of embedded platforms. Highlighting these differences leads to a better understanding of embedded architectures. This is also crucial from the aspect of developing virtualization solutions for them.

2. Motivations for virtualizing embedded platforms

   Embedded platforms are constrained in the amount of available resources, and therefore to make them run an additional piece of software (hypervisors in this case) must have some substantial basis. For any new capability that they are required to support, a solution which incurs least overhead is preferred. Thus running hypervisors on them must come with significant gains. This part tries to analyse the use cases which give rise to the need for virtualizing them.

3. Existing virtualization solutions for these platforms

   This part analyses the existing virtualization solutions which have been developed for embedded platforms. This is highly influenced by previous two points, as depending on the application, and the underlying architecture, the solutions can differ significantly. Analysing the existing solutions helps develop a better insight of the embedded platforms.

2 Virtualization on embedded platforms

Unlike x86 platform which generally runs multiple copies of same OS, the main focus of virtualization on embedded platforms is to support heterogenous set of applications. These different sets provide different kinds of functionalities. One operating system caters to one set of requirements, other caters to a different set. An example of such a device, is Motorola Evoke smartphone, in which a hypervisor was used to run an android OS, along with an RTOS supporting baseband stack [1] on a single processor. This resulted in a two fold advantage, one was that the chip count and hence the price of mobile phone decreased, second was that since both the OSes were on the same processor, which enabled low latency communication, and hence android was able to use real-time services of RTOS for rendering multimedia content. The following list summarizes the main use-cases which are the driving factors behind adoption of virtualization on embedded platforms.

[1] Baseband stack is a piece of software handling cellular communication

1. Co-existence of a feature rich OS, and an RTOS
   As complexity of embedded system grows, and it becomes important to support multiple connected components, a feature rich OS, which supports a large number of devices, and provides a big set of APIs becomes important. One such example is Android OS running on contemporary mobile phones. These feature rich OSes can be, for example used for creating nice graphical user interfaces, and networking. Embedded systems also are often required to support real time applications, which require a real time OS. These real time OSes provide good support for real time tasks, but often are not that feature rich to support multitude of connected hardware, and do not provide a good programming environment. Virtualization can help to reconcile these two divergent requirements, by allowing both kinds of operating systems to run in different VMs. RT-Xen is one such hypervisor which supports an RTOS as a guest VM.

2. Secure Communication devices
   Enterprises, to protect the assets of their company, do not allow employees to use personal mobile phones while connecting to the company intranet, or for carrying out business related communication. This is important for them, as consumer mobile phones do not have robust security policies, and can host a variety of malwares. This can lead to leaking out of company's confidential data. In such a scenario, employees are required to use a mobile phone provided by the company. This mobile phone is locked down by the company, and installing applications of an employee's choice is not supported. This causes discomfort to the employees, as they have to carry two different mobile phones, for work and for personal use. Using virtualization both kinds of needs can be supported on a single platform. The two can together be run as virtual phones inside two different VMs. The VM corresponding to work can be locked down by the company, and can have encrypted network connection to company intranet. The other VM can be freely operated by the employee. The hypervisor will gaurantee isolation between the two, and will ensure, any malfunction in one will not affect the other.

3. Consolidating multiprocessor systems
   As processing power of embedded systems increases, it becomes possible to support multiple applications on a single platform. This leads to overall reduction in cost, as multiple processors are replaced by a single, more powerful processor. For example an in-vehicle infotainment system, and vehicle control system, can be made to run on a single platform. An important constraint of such a setup is that malfunctioning of one application should not affect the other. This objective can be easily achieved through virtualization.

2.1 Architectural differences from x86

Due to the different application domains, x86 and ARM differ significantly in their architecture. The following list summarizes the main differences.

1. CPU

Arm has several CPU modes, typically more than 4. These modes generally have their own set of registers, so switching from one mode to other incurs less overhead as saving and restoring registers isn't required. For example FIQ (Fast interrupt request) mode is used to service interrupts that require low latency. Since FIQ can use its own set of registers for interrupt processing, save-restore isn't required. x86 comes with the standard 4-ring architecture and many registers are shared between the modes.

Following are the processor modes supported on most ARM architectures.

1. *User Mode* : Normal execution mode for programs, like ring 3 in x86 architexture.

2. *Supervisor Mode* : Priviledged Mode for operating system, some registers of this mode are not shared with user mode. This mode is entered on system reset and on execution of svc(supervisor call) instruction.

3. *System Mode* : It is a priviledged mode for the OS, which uses the same set of registers as user mode.It is like ring 0 in x86. This is the only priviledged mode that cannot be directly entered on an interrupt.

4. *IRQ* : This mode is used for handling low priority interrupts.

5. *FIQ (Fast interrupt request)* : This mode supports handling higher priority interrupts, and has its own set of 7 banked registers (r8 - r14), and hence saving and restoring state is not required, while servicing IRQ.

6. *Abort* : When data or instruction fetch is aborted, for example in the case of demand paging.

7. *Undefined* : When an instruction, which is not recognised by processor, or any of its co-processors is encountered. This can be used to programatically extending the instruction set, by emulating undefined instructions, when Undefined mode is entered.

8. *Hyp mode* : This is a new mode introduced in ARM processors, as a part of the virtualization extensions. This mode is more privileged than system, or supervisor mode, and is intended to run a hypervisor. This mode has it's own set of banked registers, and can be used to configure various conditions on which a VM running in system mode can exit.

In addition to these difference, they also differ in the instruction set supported. ARM architectures support Thumb-2 instruction set, in which most instructions are of 16bits, to increase code density, while some are of 32 bits to provide more functionality. x86 instruction set uses variable length instructions, and the need for backward compatibility leads to a bloated, and a larger instruction set. Some ARM cores also support Jazelle technology, which allows execution of Java bytecode in hardware, to speed up execution of Java programs.

2. MMU

On ARM the TLB entries can be aggregated together to refer to a larger section of memory than a page. The are four type of entries and these can co-exist on the same TLB.

1. Supersections: Refer to 16MB blocks of memory.
2. Sections: Refer to 1MB blocks of memory.
3. Large pages: Refer to 64KB blocks of memory.
4. Small pages: Refer to 4KB memory.

This reduces TLB pressure. In x86 support for different page sizes, is supported by PSE (Page size extension), however this provides only two kinds of page sizes 4KiB sized normal pages, and larger pages of 4MiB, or 2MiB if the system has PAE enabled. The size of the larger page in case of systems with PAE is because, for supporting these PSE, the last level of page table is done away with, and when PAE is enabled the last level uses 9 bits instead of 10 (Non PAE virtual address: 10+10+12=32 bits, PAE virtual address: 2+9+9+12=32 bits), so these 9 bits plus the original 12 bits for offset give 21 bits for offset, which causes page size to be 2MiB. Thus this feature is richer in ARM.

3. Virtualization extensions

1. Hyp mode(CPU virtualization): This is a more privileged mode than system mode (in which OS kernels run). This mode can configure the conditions on which an OS will trap to Hyp mode. This mode is very different from the root mode on x86. A root has the same 4-ring architecture as non-root mode, thus a commodity OS can run in either mode without modification. Hyp mode doesn't offer the same view to an OS as visible on system and user modes on ARM. Moreover the register names and the page table format is different. Thus a commodity OS cannot run unmodified on Hyp mode.
2. x86 VM state can be saved and restored in a single instruction, on ARM the state save and restore has to be explicitly be programmed. This has its pros and cons. While it makes the hyp mode much simpler, and faster in some cases, it proves to be slower when context has to be saved and restored frequently.
3. Stage-2 page tables (MMU virtualization): These are used to translate Guest physical addresses to host physical addresses, like EPT/NPT on

x86. However there are subtle differences in the page table structure at this level from that used in system mode, as some additional bits are to be maintained.

4. Virtual timers and counters: ARM has virtual timers and counters in H/W, unlike in x86, where timer is virtualized by using VMCS data structure. These timers are for the use by the guest, which can configure them and read values from them. However these timers cannot directly issue a virtual interrupt to guest VM, and trap to Hyp mode instead.

5. VGIC (Virtual GIC (Generic Interrupt Controller)): VGIC on ARM extends GIC to enable guests to ACK, and EOI interrupts. This speeds up interrupt processing. VGIC however doesn't allow the guest to configure the interrupt controller, and to send IPI, this must be emulated by the hypervisor.

4. Interrupt handling

ARM processors are required to EOI and ACK interrupts, while x86 only has to EOI interrupt. At the start of interrupt, address of interrupt is read from ACK register, which is a read only register. When the interrupt ends, the same address is written to EOI register, which is a write only register. Both of these registers are present in the Generic interrupt controller. In x86 the interrupt itself indicates the address, and hence no separate ACK is needed.

2.2 Challenges on x86 and ARM platforms

To develop applications for any platform, a developer has to acquaint himself/herself with the challenges which are a characteristic of that platform. Next two sections enlist the challenges unique to the x86, and ARM platforms.

2.2.1 Unique challenges on ARM platform

1. ARM processor market is fragmented with varied combinations of hardware. The instruction set and architecture are provided by ARM ,however ARM doesn't fabricate chips itself and the licensees may choose to mix and match different hardware components around that core. ARM also lacks hardware discovery features like BIOS and PCI.
   These factors make porting of OS, across all ARM platforms difficult.

2. ARM devices are resource constrained in terms of power and execution speed, and generally have to support a wide variety of sensors at once, like GPS, camera, etc.

3. Virtualizing the cellular network is a tough challenge, as is runs on a separate processor and has proprietary implementation, with only a user space library which runs on general purpose processor and exposes an interface to it. Everything beyond the library in that subsystem is complete black box.

4. As ARM has smaller instruction set the code designed for an ARM tends to be larger on an average.

5. Software being developed for ARM is intended to run on a resource constrained device, and hence optimization is one of the goals while development. Optimization is not a critical aspect of software designed on x86 platform, as they are lesser constrained on resources, and thus can afford the luxury (Which leads to some cost saving as significant investment is not required for optimization).

2.2.2 Unique challenges on x86 platform

1. x86 has supports a larger instruction set, and in in many cases has to support legacy instructions as well, these instructions need translation to newer instruction set, creating some overhead. ARM too in some cases needs translation but the problem is not as severe as in x86, and these translations are not as complex.

2. x86 has deep pipelines, often going beyond 20 or 30. This leads to more overhead on a failed branch prediction as more stages need to be flushed.

3. x86 comes with a lot of exotic features, like out of order execution, superscalar execution, speculative execution which adds lot of transistors to die, and complicated on-die logic.

4. x86 has advantage of superior fabrication techniques, which lead to better efficiency, and more powerful processors than ARM, for eg. Intel recently introduced it's tri-gate transistors which reduce power consumption. ARM fabrication technology is still lagging behind, and is in general 2 or 3 generations behind than Intel.

3 Virtualization solutions

Embedded systems encompass a big variety of processor architectures. Unlike desktop, and server platforms, there is no dominant architecture. Therefore a hypervisor which caters to all possible architectures is impractical to make, as subtle differences in architectures greatly impact VMM design. The definition of efficiently virtualizable platform, as given by Popek and Goldberg[7] leads to two broad categories of processors: ones that satisfy the constraint for efficient virtualization, i.e. for them the set of sensitive instructions is a subset of priviledged instructions, and the ones that do not satisfy this constraint. The ones that are efficiently virtualizable (eg. PowerArchitecture), are not without any limitations, and therefore a VMM must try to overcome these to run with lowest overheads. Processors that cannot be efficiently virtualized either rely on, either hardware extensions, or on some software techniques. The categories of virtualization solutions studied by me, in the course of this seminar, and the relation between them can be seen in figure 1.

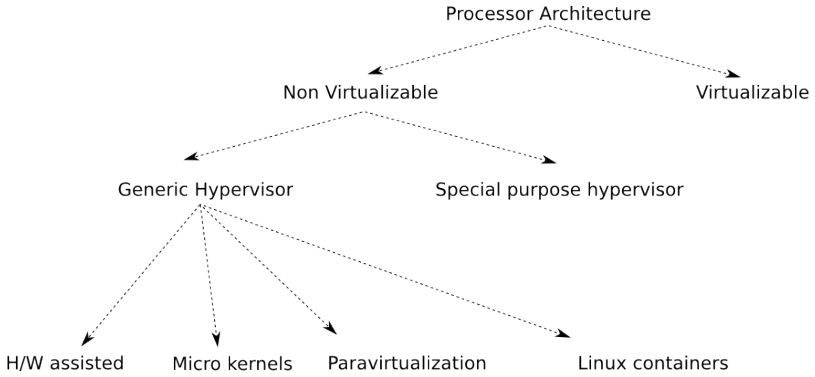

Figure 1: categories of virtualization solutions for different kinds of embedded systems. virtualizable: processors for which set of sensitive instructions is a subset of priviledged instructions

3.1 Virtualizable platform

These platforms meet the Popek and Goldberg virtualization requirement. For them a trap and emulate style VMM is easy to construct. Power Architecture is one such processor. The performance overhead on such a platform remains high, as all semsitive instructions trap to hypervisor. This overhead is significant, specially in processor bound applications. To reduce overhead a dynamic binary translation scheme can be used, as presented in [6].

The authors of the paper observe that in Linux running on Power Architecture, most of the traps from sensitive instructions occur from a small set of these instructions, moreover the locations where these instructions are present in kernel code, are also small in number. These locations can be monitored for the number of traps occurring at them, and if the number is high, they can be patched to improve performance. To prevent interference with rest of the code, a single sensitive instruction is replaced by a only one instruction while patching. This instruction can either be a single instruction which emulates the sensitive instruction (for eg. replacing mfmsr by load instruction [2]), or by a jump instruction to a translation cache (if the sensitive instruction cannot be emulated by a single instruction). In case the second instruction cannot be emulated by a single instruction, it becomes important to figure out two questions:

How will the jump take place ?, and
Where will the translation cache be located?

To make a jump to translation cache, either a `bl`[3] instruction can be used, or by `branch` instruction. The former can lead to corruption of link register[4]. For example, when the code to be patched is already a part of a function call,

[2] mfmsr:move from machine state register, this instruction copies contents of Machine State Register(MSR) into a general purpose register (GPR). This can be replaced by a load instruction which loads the GPR with the memory location emulating MSR for the VM.

[3] a `bl` instruction, along with `blr` instruction is used in Power Architecture for calling a function, and returning from it

[4] link register is used to store the return address during a function call

a new function call without saving the original content of the link register, will overwrite the original value. Also, it will not be possible to store the contents of the link register because of our constraint to use only one instruction at the patch site. The `branch` instruction seems as the obvious choice now, and is the one which is used to jump to translation cache. This instruction puts a constraint on the location of the translation cache, as it only allows to jump within $\pm 32mb$ of the program counter. To overcome this limitation translation cache is placed in data section of a process, by stealing a data page from the virtual address range of a process. For this, a page containing data of the process is selected, its contents are backed up, and translation cache is placed their. To protect this stolen page, and to emulate access to this page, in case the process tries to read or write to it, read and write permissions are revoked, and execute permission is set. When program tries to access this data location, a fault occurs, which is termed tracing fault. This binary patching scheme is only viable till these tracing faults are less in number, than the faults occurring due to the sensitive instruction.

This discussion, highlights that even if a platform satisfies Popek and Goldberg's virtualization requirement, trap and emulate mechanism of virtualization may not be very efficient, and even for such platforms, significant thought must be given while virtualizing.

3.2 Non Virtualizable platforms

For these platforms, the set of sensitive instructions are not a subset of privileged instructions, and therefore successful emulation of instructions becomes a challenge. ARM architecture, which is the dominant architecture being used in mobile phones, is an example of such an architecture. For these kind of architectures, the virtualization solutions present are of two kinds: Generic , and special purpose.
Generic hypervisors run multiple copies of the same operating system. Special purpose hypervisors are made to satisfy a single set of requirements. As special purpose hypervisors are not needed to support all possible scenarios, they can be much simpler.

3.2.1 Generic hypervisors

3.2.1.1 H/W supported virtualization ARM has come up with several hardware extensions to support virtualization. These include a new CPU mode, called the HYP mode for hypervisor execution, a newer interrupt controller to accelerate handling of interrupts in guest, called VGIC (Virtual Generic Interrupt Controller), stage 2 page tables (which are like EPT/NPT on x86[5]), and virtual timers and counters. These extensions can be leveraged to provide an efficient virtualization solution. Unlike x86, these extensions are not amenable to commodity OS, as the HYP mode is significantly different from SVC[6](supervisor) mode. The HYP mode is more suited to a bare metal hypervisor. As ARM processors come with a diverse set of specifications, supporting

[5] EPT: Extended Page Table, NPT: Nested Page Tables (aka Rapid Virtualization Indexing)

[6] mode in which kernel runs on ARM

all of them would require writing a bare metal hypervisor for each. A novel solution suggested by Christoffer, et al. [3] is to use hosted hypervisor, in which a Linux OS acts as a host. This Solves the problem of compatibility , as Linux is already supported on most ARM platforms. The HYP mode can then be used to place a small routine which will facilitate interaction between the guest OSes and the Linux host. Every interaction between guest and host will require two world switches, one from guest to HYP mode, and then from HYP mode to host OS, and same number of switches while going from host to guest. The role of HYP mode in this case, when going from guest to host will be to save the state of guest and restore the state of host, while disabling second level address translation (as the host is not required to run in a virtual environment, thus it will not require second level page tables). similar will be the procedure when going from host to guest. The devices can be emulated either in the host kernel, or in the host user space. As the extensions include virtual timers and interrupts, these can be configured by the guest OSes directly, without the involvement of hypervisor, moreover VGIC allows ACK-ing and EOI-ing (explained earlier) without the involvement of hypervisor which significantly increases efficiency. Virtual timers and counters, cannot generate an interrupt directly to the guest OS, and that mechanism still has to emulated in the hypervisor. VGIC also cannot be used by guests to configure interrupts, and for sending IPI (inter processor interrupts), so these mechanisms too have to be emulated in hypervisor.

3.2.1.2 Paravirtualization Before ARM came with hardware extensions to support hypervisors, the virtualization solutions mainly relied on paravirtualization. As ARM came up with these extensions recently, the set of solutions is still dominated by the ones using paravirtualization. This technique is same as paravirtualization on x86 platform. This method suffers from one disadvantage that it requires significant developmental effort to paravirtualize an OS, and most of the times the paravirtualized OS is several versions behind the latest version. These efforts are often specific to an operating system, and require experienced developers. KVM for ARM [2] presents an efficient way to automate the process. Using pattern matching it replaces sensitive instructions in guest OS's kernel code with calls to hypervisor. These calls to hypervisor are either made using SWI instruction (which is used for making operating system calls on ARM) or by using co-processor access instructions. Both of these allow 24-bit space for operands, and together can be used to replace all sensitive instructions in guest code. Now, as in a paravirtualized OS, both the user programs, and OS kernel run in user mode, it is important to distinguish between the calls made by the patched versions of sensitive instructions, and by user space programs for a kernel function. The former must be emulated by the hypervisor, while the latter must be passed on to guest kernel. This distinction is made by maintaining state in virtual CPU(vCPU), that represents mode of operation, i.e. if guest kernel is executing on vCPU, the mode of vCPU will be set to supervisor mode, even though in reality the physical CPU executing the virtual CPU will still be in user mode. When a call reaches the hypervisor, the mode of vCPU can be checked to make the distinction.

For virtualizing memory shadow page tables can be used. As ARM uses a concept of domains for memory protection, the shadow page tables created by hypervisor must be aware of these, and the bits corresponding to domains must be set correctly.

3.2.1.3 Micro kernels

3.2.1.4 Linux containers (LXC)

Drawbacks:

1. Security is limited by the security of the kernel, as for all containers the kernel remains the same.
2. Cannot host applications from different operating systems simultaneously.

3.2.2 Special purpose hypervisors

In a BYOD like scenario employees of a company are allowed to use their personal mobile phones for work related communication. In such a scenario, it is important to safeguard assets of a company from any malicious application running on the mobile phone. Virtualization can be used to run two virtual mobile phones on a single physical phone. One virtual phone can be a less featured, locked down phone, to be used for work, while the other virtual phone, with no such restrictions can be used for personal use. VMware's MVP (Mobile Virtualization Platform)[1], and vNative[4] are two hypervisors for supporting such a use case. MVP runs on ARM architecture, while vNative runs on Intel Atom processor, which is Intel's offering for embedded platforms. Although their intended goal is same, their approach towards the solution is quite different, and is quite interesting to compare.

VMware MVP
This is a hosted (type 2) hypervisor. The host, also acts as the personal mobile phone. When MVP is installed on a mobile phone, it creates a secure and isolated VM for work related usage. The VM runs a paravirtualized version of Android operating system.
Secondary storage used by Guest VM is encrypted via block encryption. Most common secondary storage device in mobile phones is the SD card, hence keeping that in mind further optimization is done. SD card's non-sequential writes are orders of magnitude slower than sequential writes, therefore MVP buffers writes. Network traffic is tunnelled through VPN from work VM to the company network. Memory partitioning between guest and host is achieved using shadow page tables. Here it is assumed that the work VM will not require many features, for eg. graphics acceleration for games, thus the VM design is reduced in complexity.

vNative
vNative allows multiple VMs to exist simultaneously, however only one VM executes at a particular time, remaining are present in a reactive state, i.e. they don't consume CPU cycles, but are capable of reacting in case of an event. The VM in execution is called the FVM (Foreground VM), while others are called BVMs (Background VMs). Only FVM interacts with the user. Memory is statically partitioned between guests, using Intel's IMR (Isolated Memory Regions). The multiplexing of devices between VMs is done by FLR (Function

Level Resetting), which is a faster way to relinquish devices, and put them in a state which is suitable for another VM to use, than doing a conventional reset.

Comparison of the two techniques

MVP simplifies the use case, by assuming that the work phone will not be required to be feature rich, thus it doesn't support many devices, for example the need of good graphics might not be there for it, and therefore graphics acceleration can be left out. vNative on the other hand, uses FLR, to arbitrate device access. As, while using FLR, one VM completely gives away it's control of devices, before other VM starts using it, the VMs are able to use their own native drivers, which provides a big saving in development time, as there is no need to make special drivers, also all the devices can be given to each VM, unlike in MVP. This also allow each VM to do power management independent of others. This feature however is purely dependent on FLR functionality. For switching between VMs, both the techniques require explicit user intent, which is shown by pressing a button on home screen. Also, the VM(s) present in background remain reactive in both the cases.

4 Conclusion

References

[1] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The vmware mobile virtualization platform: Is that a hypervisor in your pocket? *SIGOPS Oper. Syst. Rev.*, 44(4):124–135, December 2010.

[2] Christoffer Dall and Jason Nieh. Kvm for arm. In *Proceedings of the Ottawa Linux Symposium*, pages 45–56, July 2010.

[3] Christoffer Dall and Jason Nieh. 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*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM.

[4] YaoZu Dong, JunJie Mao, HaiBing Guan, Jian Li, and Yu Chen. A virtualization solution for byod with dynamic platform context switching. *Micro, IEEE*, 35(1):34–43, Jan 2015.

[5] Gernot Heiser. Virtualizing embedded systems: Why bother? In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 901–905, New York, NY, USA, 2011. ACM.

[6] Aashish Mittal, Dushyant Bansal, Sorav Bansal, and Varun Sethi. Efficient virtualization on embedded power architecture&#174; platforms. *SIGPLAN Not.*, 48(4):445–458, March 2013.

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