

Embedded VMM for Portable Virtual Machines

Naveen Kalla, Patrice Guelah and Scott Armstrong
(nkalla2, pguela2, sarmstr2 @uiuc.edu)

Abstract

Mobile phones running embedded operating systems and multiple applications on modern processors can provide functionality close to that of a desktop computer, but they can also be vulnerable to problems and threats similar to those of desktop computers. Virtual Machine Monitors (VMM) have been used to provide solutions for servers and desktop computers, so it follows that VMMs could be candidates for a variety of solutions in the current mobile phone market.

We present an implementation of an embedded VMM running on an ARM-based device, hosting the Android mobile phone software stack. This implementation will demonstrate the practical application of a VMM to run multiple software phones on one device, and to move virtual machines (VM) from one cell phone to another.

Categories and Subject Descriptors:

Primary Classification: D.4.7 [Operating Systems]: Organization and Design

Secondary Classification: D.4.9 [Operating Systems]: Systems Programs and Utilities

General Terms: Virtualization, Design, Security, Performance

Keywords: virtual machine monitors, hypervisors, paravirtualization, embedded systems

1. Introduction

As mobile phones become more feature rich, it is becoming tougher to shrink them into smaller and thinner form factors [1]. Most mobile phones used a different processor for running the communication stack (frequently called baseband processors) and a different processor for running the applications (frequently called application processor) [2], [3]. The real time requirements of the communication protocols and the limited processing power of the legacy application processors made it inevitable to have two processors. Extra hardware was needed to facilitate communication between the two processors thus resulting in bigger die sizes and higher bill of materials. Hence it was tougher to have high end phones in smaller and thinner form factors [1].

Even with the availability of high performance processors, there are hurdles keeping mobile phone manufacturers from using a single processor to run both the embedded Linux operating system along with its applications and the mobile communication stacks. One hurdle is the licensing issue. Linux is distributed under the GPL license and hence all derived code including the device drivers that are loaded into the kernel is subject to the same license and thus become open source [4-7]. But for chip vendors who consider device interfaces as valuable proprietary IP, this could be a potential problem. Another hurdle is dealing with security threats. If the operating system is compromised and the communication stack was running under this operating system, then there is a potential problem that the attacker can turn the phone into a jammer and thus disable communication in the entire cell. This could cause failure of even emergency calls for all mobile users in that cell [6]. We haven't seen any of these problems earlier because most of the single-chip mobile phones used proprietary operating systems and home grown applications. With the advent of smart phones which used operating systems such as Windows and Linux to support the killer applications, mobile phones were designed with dual processors [2, 3]

In the interest of miniaturization, a single processor could be used for both the communication stack and the applications with the help of VMMs. VMMs have been used to solve a variety of problems or challenges over the years [8], [9], [10] and they will likely continue to be applied as a tool to discover and solve new problems. Server Consolidation, Testing and development, Dynamic Load Balancing, Disaster Recovery, Virtual Desktops, Improved System Reliability and Security are some of the areas where virtualization has shown its advantages. Embedded VMMs allow us to run multiple operating systems

concurrently on the same hardware platform. As more processing power is being packed into a single processor, we could probably use a single processor for both the communication stack and the applications. We can solve the problems mentioned above by running the communication stack under a real-time operating system as a virtual machine and Linux as a separate virtual machine on the embedded VMM. This would provide isolation to the communication stack in addition to numerous other benefits provided by the embedded VMMs [6, 11].

Though VMMs are not new [12], [13] there has been very little work toward virtualizing ARM-based devices [14]. VMMs have been deployed for years by VMWare and Xen from the University of Cambridge, both aimed at the enterprise market. Recently virtualization solutions aiming at embedded designs have emerged, such as L4/Wombat from the University of New South Wales, and OK4 from Open Kernel Labs. We hope to take the current research further and demonstrate a viable, real-world application of VMMs on portable devices.

We plan to develop a VMM for a mobile phone, bringing the benefits of a VMM to the handheld, portable device world. We will use the *Linux-based Android system* to demonstrate the benefits of the attributes of VMMs in the mobile phone environment. The Android system [15] architecture consists of an operating system layer running a modified Linux 2.6 kernel. This architecture is designed to run on the *ARM-based processor*. Development tools for the Android system include a hardware emulator to run and test applications developed for the Android system (Linux OS, libraries and applications) [15]. This emulator makes use of the QEMU ARM Instruction Set Architecture (ISA) [16] to emulate an ARM-based hardware telephone set. QEMU uses a dynamic translator [17]. The dynamic translator performs a runtime conversion of the target CPU instructions into the host instruction set. Our primary goal is not to emulate the hardware architecture, which QEMU already does. Since ARM7 is not fully virtualizable unlike ARM9 [13] we will have to handle sensitive instructions which could be done in a number of ways [18]. Apart from this our primary goals of this project are listed below.

2. Goals

2.1. Embedded VMM

Our VMM that will run between an ARM-based phone (or Emulator) and the Linux OS of the Android system.

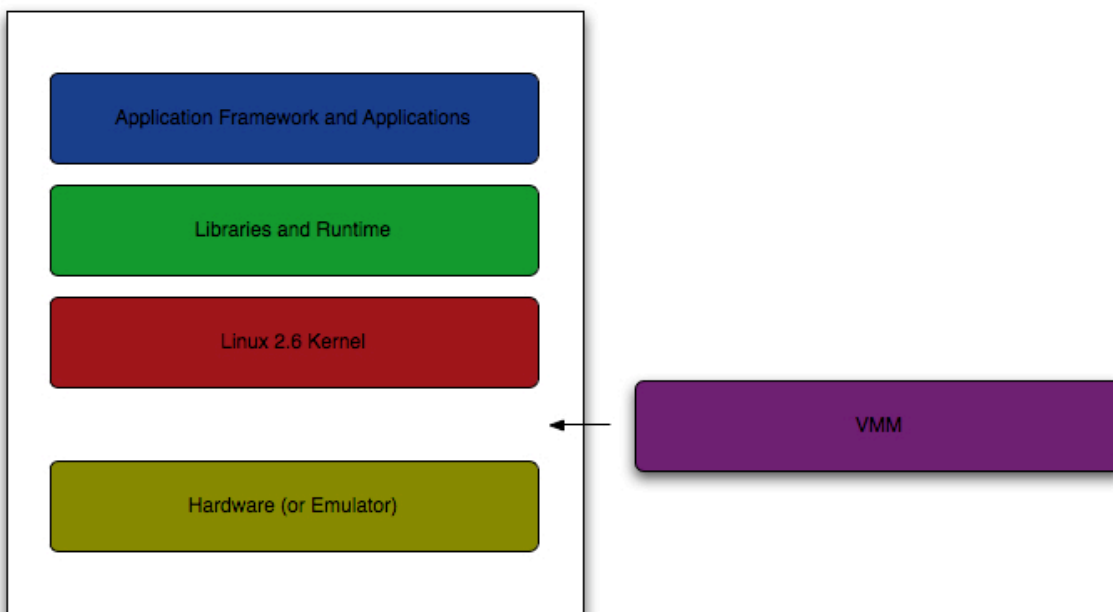


Figure 1: Location of the VMM in the complete Android mobile phone.

The main goal of this VMM development and enhancement effort is to demonstrate the VMM-enabled ability to run multiple software “phones” on the same device. The secondary goal is to develop a means to simplify the migration of the complete Android system software stack (OS with system state, libraries, and applications) between two hardware (or emulated hardware) environments, taking advantage of the check-pointing capabilities of VMMs.

2.2. Anticipated Results:

1. Boot a QEMU-based VMM, and from that VMM, boot the Linux OS of the Android system;
2. Boot a second instance of the Linux OS of the Android system;
3. Checkpoint one version of Android and restore the OS from a checkpointed system; and
4. We expect to incur performance overhead in our initial efforts to virtualize the Android system on the ARM ISA (Future efforts can be dedicated to performance enhancements);

3. Design

3.1. Preliminary Design Research

Our initial design efforts included a variety of approaches to virtualize the Android software stack on the ARM-based platform. We initially considered building on existing research at the University of Illinois using the Choices [16] operating system (OS) and the Choices-based VMM [12], [13]. Though the source code for the object-oriented Choices OS is relatively stable and is publicly available, we learned through communications with the research group [12] that the Choices-based VMM was a prototype and supported only a very minimal demonstration guest OS.

Though different in implementation than a VMM implemented as a layer, such as Xen or VMWare [25], user-mode Linux [24] provides features and benefits of a virtualized operating system, including isolation and the security that is associated with isolation. Using the Android-specific Linux source code, we were able to compile Linux to run only in user mode, executing no instructions that could access system resources. User-mode Linux requires a root file system unique to each running instance of Linux, and since there are a variety of pre-compiled root file systems available for use with user-mode Linux, we looked at different root file system options.

We successfully started two instances of user-mode Linux in one workstation environment using the Busy Box root file system [26]. This file system allowed user-level access to the host operating system’s file system, effectively allowing a user in one instance of Linux to create a file on the host that a user in the other instance of Linux could read and modify. This lack of isolation between running instances of user-mode Linux made the user-mode Linux system with Busy Box unattractive for our purposes.

We also started two instances of user-mode Linux in one workstation environment using the Slackware root file system [27]. This configuration provides an isolated environment much more like a traditional VMM hosting a VM; though, performance will likely be worse in this user-mode Linux environment than in a traditional VMM layer environment. As our work proceeds, we plan to compare the performance of our Android VMM system with that of the Android software stack running in this type of a user-mode Linux environment.

The L4-Wombat [18] virtualization environment is a research virtualization system that uses the L4 microkernel running Iguana, an ARM-based hypervisor and Wombat, a specialized version of Linux. Because the Wombat Linux code was tailored so extensively to run on the L4/Iguana hypervisor, basing our work on this system would result in a highly modified Linux VM, making it much less useful in the open-source environment supporting the Android system [15].

3.2. Design

The Android VMM is basically built with the following main components

- Booting and Initialization
- Guest OS loading module

- Guest OS scheduling and instruction emulation/translation module
- Memory Management module
- Interrupt / Exception / Fault handling

This section gives a high level overview of each of these components.

3.2.1. Booting and Initialization

The startup code performs the following functions

- Check the system to identify the CPU type it is running on and perform the necessary CPU initialization (such as setting up the stacks, zeroing the BSS, etc).
- Copies the Android VMM code to the high end of conventional memory
- Sets up the page tables and page directories needed for the android VMM
- Enable cache and MMU and jump to the main android function

3.2.2. Guest OS loading module

This function takes the guest OS image, ramdisk image and initialization parameters for each guest OS that it needs to load and performs the following functions

- Loads each OS at a separate physical address.
- Creates a per Virtual Machine structure
- Initializes the contents of the structure with interrupt status, shadow page table pointers, exception handlers of each guest OS, etc.

3.2.3. Guest OS scheduling and instruction emulation/translation module

Operating systems run in privileged mode and have access to privileged CPU instructions. To prevent interferences between guest operating systems in a virtualized environment, guest operating systems run in user mode. They are given the appearance of having privileged access, but control is passed to the VMM when a guest attempts to execute a privileged instruction.

Another class of instructions that cannot be executed by a guest OS in a virtualized environment are sensitive instructions. This is because their behavior depends on the current CPU privilege level. The guest is given the appearance of privileged access and will not get the desired result because it is in reality running in user mode. Only the VMM knows about real levels information to execute the instruction properly.

The MMU instructions, exceptions such as SWI, MSR and MRS instructions that read and modify the Program Status Register, and coprocessor access instructions like MRC and MCR, among others, are either privileged or sensitive instructions on the ARM ISA and cannot be executed by the guests running in user mode.

There are two main approaches to virtualization:

- Full virtualization fully emulates the underlying hardware in each virtual machine and allows the guest OS to run unmodified in the virtual machine. In fully virtualized systems, the guest OS is allowed to call all instructions, but privileged and sensitive instructions are trapped and handled by the VMM.
- Paravirtualization involves modifying the guest OS to allow direct calls to the VMM for privileged and sensitive instructions.

We decided to implement full virtualization for the android VMM because it has the advantage that we can run Android unmodified. Full virtualization presents some challenges on systems such as the x86 where privileged instructions do not trap when executed in a non-privileged mode. To be able to provide full virtualization in our VMM, we had the option of using binary translation. This means translating the guest OS instructions and replacing privileged and sensitive instructions with new instructions that emulate the desired effect on the virtual machine. This approach is quite complex and the performance penalty of

having the VMM virtualize each guest OS instruction is impractical for the mobile environment.

Many hardware vendors including Intel and AMD have started to provide hardware support for virtualization in their products by developing new features to simplify virtualization [20], [21]. One of the new features being provided is to allow privileged instruction calls to automatically trap to the VMM.

The current version of the Android software stack is compiled for ARMv5 which does not have hardware support for virtualization. Most recent versions of ARM processors like ARM Cortex™-A9 t processor provide hardware virtualization support through architecture extensions [22]. We anticipate that future versions of android will be compiled to run on newer ARM processors with hardware virtualization support. For that reason and to keep our design simple, we choose to make modification to the QEMU ARM CPU emulator to add some hardware virtualization support. Our virtualization extension forces all sensitive and privileged instructions called when the CPU is in user mode to trap to the VMM. The VMM handling routine takes control and virtualizes the instruction as required.

In order to run code compiled for ARM CPU on an x86 CPU, QEMU performs a runtime conversion of ARM CPU instructions to corresponding x86 CPU instruction and execute them on the x86 host CPU. QEMU performs the translation in blocks of ARM instructions translated together. A block is a set of instructions up to a jump or an instruction that modifies the known static CPU state in a way that cannot be determined at runtime [28]. After translating a block, it is executed on the x86 host CPU and then cached in a 16MB cache. This reduces the translation overhead as the block does not need to be retranslated each time it is needed.

To implement hardware virtualization support in QEMU, we provide a means to transfer control from a guest OS to the VMM when the guest OS attempts to execute a sensitive or privileged instruction. These instructions are executed directly only if the CPU is in privileged mode. In user mode, the VMM handling routine is called to emulate the instruction. The VMM keeps track of the state of each guest OS as well as relevant information needed to validate the privileged instructions before executing them.

The VMM manages the hardware clock and provides separate logical clocks to the guests. We have a simple scheduling scheme where equal CPU time is allocated for each guest OS. The VMM keeps a list of all VM on the system. Each one is allocated CPU time and a priority. When a guest runs until its time slice expires, the scheduler selects the next guest on the list or the VM with the highest priority to run. Basically, the scheduler is called periodically to pass control to the next VM.

3.2.4. Memory Management

In a virtual environment the guest OS does not maintain the hardware page tables. It thinks it does but in reality it does not. It maintains what we will refer to as the guest page table and the VMM maintains the real hardware page tables. The VMM reflects the changes made by the guest OS in the guest page tables in the hardware page tables.

Each guest OS has its own virtual view of the memory. The real memory contents are known only to the VMM. So the main job of the memory management module is to validate the guest physical page addresses in the guest page tables every time they are changed by the guest OS and map it to the real machine addresses.

The Memory Management module for Android VMM provides the following services:

- Managing the page tables for the VMM. Making sure it is not corrupted when the guest OS begins to run. Swapping the page tables when there is a context switch from one guest OS to another using shadow page tables.
- Managing the shadow page tables for the guest OS. Shadow page tables provide an extra level of indirection between the guest physical address and the real machine address. These may have to be updated every time there is a TLB flush or a page fault on the pages that contain the guest page tables as described below

The VMM detects when its shadow tables are out-of-sync with the guest tables and it re-syncs them. This

happens at two times: when handling page faults and when processing TLB flushes.

The guest can increase access to pages by marking 'invalid' guest page table entries as 'valid', or marking 'read only' entries as 'read write'. The VMM will only detect its shadow entries are out of sync when the guest faults while attempting to access or write those pages. At that time the VMM re-syncs the shadow page table entries and the guest retries the access.

The guest can reduce access to pages by marking them 'invalid' or 'read only' or by changing the pages to which page table entries point. Before such changes become effective, the guest must flush the affected TLBs. This gives the VMM an opportunity to re-sync any modified page table entries.

But how does the VMM know which page tables have been modified? It is not practical to scan all guest page tables. The solution is to keep all guest page tables read-only normally. Then, when the guest tries to change a guest page table, a page fault occurs. The VMM can add the guest page table to an "out of sync" (OOS) list and allow write access. At TLB flush time, the VMM need only review the pages on the OOS list for changes. Once all changes in the guest page tables have been reflected in the shadow page tables, the OOS list is emptied and all guest page tables are made read-only again.

The following subsection describes the salient features of the ARM Memory Management Module (MMU) which will be used to provide the memory management functionality in the VMM.

ARM MMU

A single set of two-level page tables stored in main memory is used to control the address translation, permission checks, and memory region attributes for both data and instruction accesses. The MMU uses a single unified *Translation Lookaside Buffer* (TLB) to cache the information held in the page tables.

To support both sections and pages, there are two levels of address translation. These pages and sections are accessible through one or two stage table walking depending on page-mapped or section-mapped access. The MMU puts the translated physical addresses into the MMU TLB.

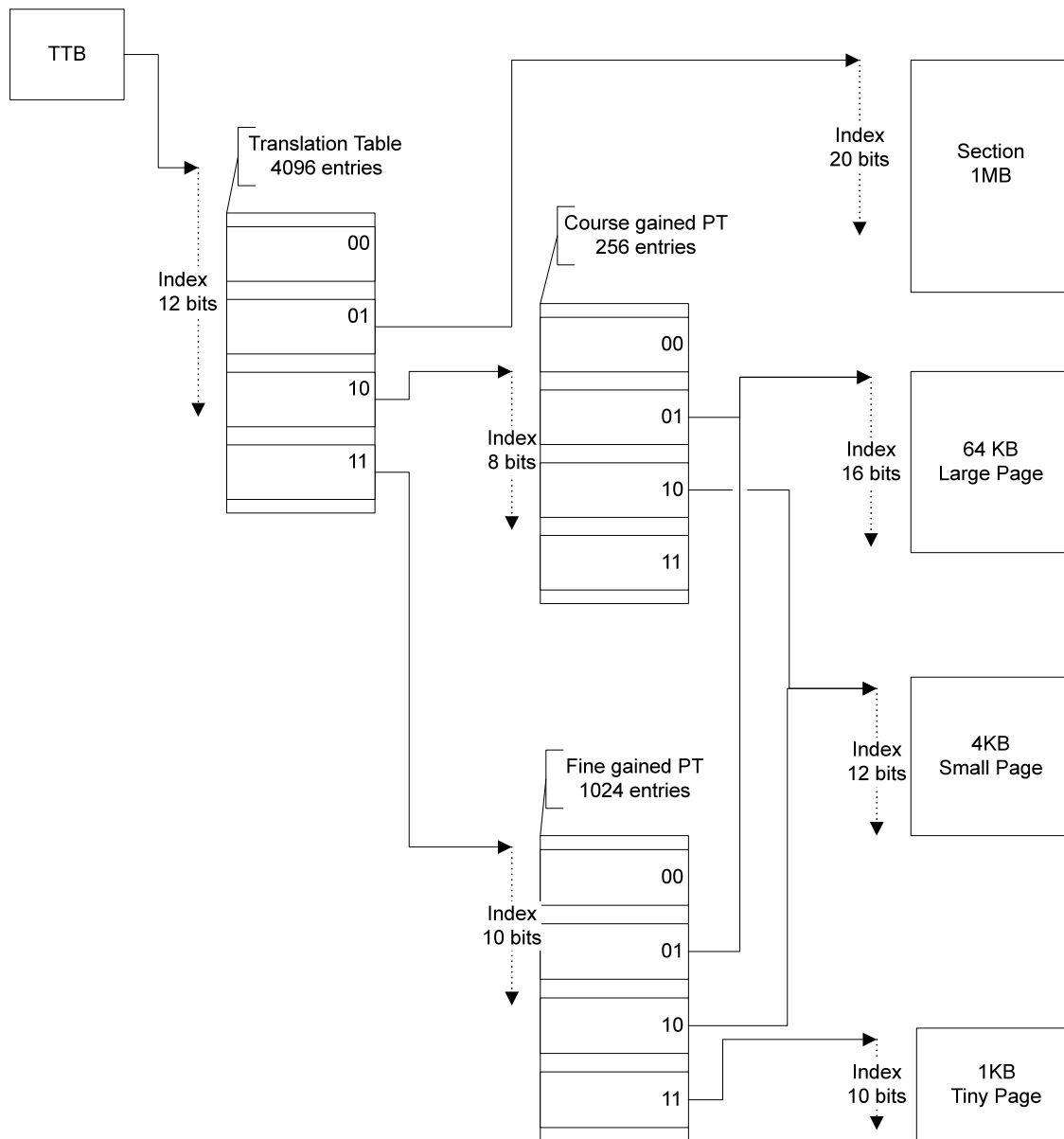
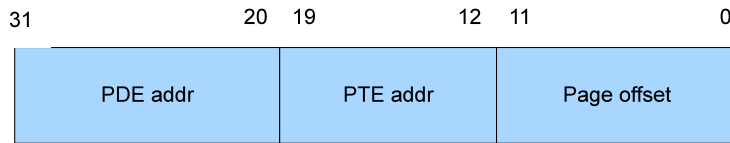


Figure 2: MMU Access Diagram

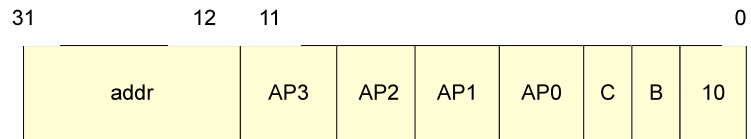
The MMU features are:

- mapping sizes are 1MB (sections), 64KB (large pages), 4KB (small pages), and 1KB (tiny pages)
- access permissions for large pages and small pages can be specified separately for each quarter of the page (sub-page permissions)
- hardware page table walks

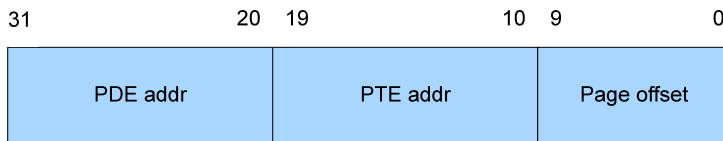
A virtual address consists of three parts: a page directory index, a page table index, and a page offset. The page directory index and page table index are offsets into first- and second-level tables of addresses. The page offset indexes into a memory page itself.



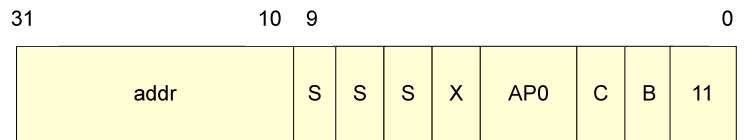
4KB Virtual address layout



4KB Page table entry layout



1KB Virtual address layout



1KB Page table entry layout

Figure 3: Virtual addresses and page table entries. The PTE bits are as follows : AP = access permissions, C = cacheable, B = bufferable, S = free for software use, X = software Copy-on-Write

ARM gives programmers a choice for the layout of their page tables. ARM allows for large (64 KB), small (4 KB), or tiny (1 KB) pages. ARM's 4 KB pages are divided as 12:8:12. This ARM division stores more data in page directories than the x86 (4×2^{12} or 16384 bytes) and less data in page tables (4×2^8 or 1024 bytes). This means page directories require four aligned, contiguous memory pages, and page tables will only use up one fourth of a page. Furthermore, page table entries in this MMU configuration do not contain any bit space for implementation-specific (software) flags, should they be needed.

The alternative to 4 KB pages are ARM's tiny 1 KB pages. These do allow a few bits of space in the page table entries. Using the tiny layout, the 32 bits of a virtual address are divided as 12:10:10. This means that pages are 2^{10} or 1024 bytes in size. It also means that each page table entry stores 22 bits of a page's memory address, and 10 bits of meta data about a page. Page directories now take up sixteen aligned and contiguous pages, whereas page tables require four tiny pages of space.

On ARM, the MMU hardware does not perform read/write permission checks at the page directory entry level. There is an entity on ARM called a domain, which can completely restrict a page table from non-kernel access. This is useful for shielding kernel memory from user processes.

3.2.5. Interrupt / Exception Handling

The guest OS has its own set of interrupt and exception handlers; the function of these handlers ranges from simply passing the exception straight through to the virtual machine to emulating the instruction causing the exception. The VMM determines which I/O interrupts will be handled by the guest OS and which will be handled by the VMM itself [19]. The VMM may also need to monitor attempts by a guest OS to block and re-enable interrupts within its VM.

4. Implementation

4.1. Development Environment

We started with the Android SDK that is comprised of the tools and APIs necessary to begin developing applications on the Android platform. This SDK comes with an emulator which was basically a customized version of QEMU.

The operating system running Android is a Linux distribution based on the Linux kernel version 2.6 for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack.

We were able to obtain the configuration used to build the Linux kernel in the Android image that runs on the emulator. Looking at the configuration we concluded that the emulator basically emulates an ARMv5 based CPU.

We used the Linux 2.6.23 Android M5-RC14 source code and the Android Emulator M5-RC14 source code available for download from the Android source downloads page [29], along with the configuration extracted from a running Android emulator to successfully build an Android Linux kernel and run it on the emulator. This provided the base environment for our project development.

We used the following tool chains to compile the Android image

- ARM 2007q3 GNU/Linux and IA32 GNU/Linux tool chain from CodeSourcery [23]
- Android image compilation tool chain from the adt-m5-rc14 plug-in for Eclipse [29]

4.2. Implementation Details

We implemented boot and initialization code for our android VMM that initializes the CPU, MMU, cache and the UART, and modified the Makefiles to build a small Android VMM image. We also added a printf capability to the Android VMM to help us debug the VMM.

Our next task involved passing the guest OS information to the android VMM. There were a number of ways to do. One method consists of adding ramdisk support to the VMM, storing the guest OS images in the ramdisk and having the VMM read information from the ramdisk. But for the sake of simplicity, we decided to add the android kernel and the ramdisk image used by the android kernel as binary data that can be accessed at a labeled segment by the VMM. This would keep the VMM simpler and enable us to jump into the core VMM functionality faster.

Once we were able to access the guest OS from the VMM, we started to implement the guest OS loading module. Currently, the guest OS loading module is able to load the guest OS and the ramdisk to a certain location in memory and pass control to the image. The image prints the following string to the console:

```
Uncompressing Linux...
```

This string is printed from the bootloader in the Android image. We are currently exploring the memory

management module, the CPU instruction emulation which involves the changes needed in the QEMU-based Android emulator and the interrupt / fault handling functionalities in the android VMM.

5. Evaluation

(future)

6. Related Work

(future)

7. Conclusions

(future)

8. References

1. Fox, K., *Mobility Maximized: Intel Redefines What a Wireless Handheld Can Do*. Technology@Intel Magazine, 2004(May).
2. Pajak, D., *System Solutions for a Baseband SoC*. IQ (Information Quarterly) Magazine, Vol. 5 2006(Issue: Number 2).
3. *Qualcomm MSM7500 Convergence Platform Baseband Applications Processor*, http://www.semiconductor.com/resources/reports_database/view_device.asp?sinumber=17078. Semiconductor Insights.
4. *Free Software Foundation. Frequently asked questions about the GNU GPL*, <http://www.fsf.org/licensing/licenses/gpl-faq.html>. 2007.
5. *LKM's should not be used to evade the GPL*. <http://www.wasabisystems.com/LKM/summary/>, 2007.
6. Heiser, G., *Virtualizing Embedded Linux*, <http://www.embedded.com/design/opensource/205918954?pgno=1>. Embedded.com, 2008.
7. Paul, R., *Why Google chose the Apache Software License over GPLv2 for Android*. Ars Technica, (November 06, 2007).
8. Barham, P., et al., *Xen and the art of virtualization*. Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM, New York, NY.
9. Rosenblum, M. and T. Garfinkel, *Virtual Machine Monitors: Current Technology and Future Trends*. Computer 38, 2005(5): p. 39-47.
10. Chen, P.M. and B.D. Noble, *When Virtual Is Better Than Real*. Proceedings of the 2001 Workshop on Hot Topics in Operating Systems, 2001(May 2001).
11. Kulkarni, D., et al., *Intel® Virtualization Technology in Embedded and Communications Infrastructure Applications*. Intel Technology Journal, 10, 2006(03).
12. Philip A. Reames, et al., *A Hypervisor for Embedded Computing*. Illinois Journal of Undergraduate Research, 2007(April, 2007).
13. Bhardwaj, R., et al., *A Choices Hypervisor on the ARM architecture*. CS523 Course Project

- Report, Department of Computer Science, University of Illinois at Urbana-Champaign, 2006.
14. David, F.M., et al., *Porting Choices to ARM based platforms*. Technical Report UIUCDCS-R-2007-2830, 2007(March, 2007).
 15. Alliance, A.-O.H., *Android – Open Handset Alliance Project Web pages*. <http://code.google.com/android>.
 16. Systems Research Group, D.o.C.S., University of Illinois at Urbana-Champaign, *Choices project Web Pages*. <http://choices.cs.uiuc.edu>.
 17. Bellard, F., *QEMU, a fast and portable dynamic translator* Proceedings of the 2005 USENIX Annual Technical Conference. April 10-15, 2005, 2005.
 18. Heiser, G., *Virtualization for Embedded Systems*. http://www.ok-labs.com/_assets/download_library/OK_Virtualization_WP.pdf, 2007.
 19. King, S. and Chen, P., *Operating System Extensions to Support Host Based Virtual Machines*, Technical Report CSE-TR-465-02, Department of Electrical Engineering and Computer Science, University of Michigan, 2002.
 20. Neiger, G., et al., *Intel® Virtualization Technology: Hardware Support for Efficient Processor Virtualization*, Intel Technology Journal, Volume 10, Issue 3, <http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art01.pdf>, August 10, 2006.
 21. *Virtual Server 2005 R2 SP1 and AMD Opteron™ Processors Increase the Buisness Value of Virtualization*, http://enterprise.amd.com/GetFile.aspx?aliaspath=%2FDownloads%2FSMB%2FVirtualization_whitepaper-pdf, January, 2007.
 22. Trango Virtual Processors™ announces Secure Virtualization on ARM Multicore Platforms, http://www.trango-vp.com/download/press/release/20071003_TRANGO_ARM_MPCORE_Announce, Trango Press Release, October 3, 2007.
 23. CodeSourcery Web pages. http://www.codesourcery.com/gnu_toolchains/arm/download.html.
 24. Dike, J. 2000. A user-mode port of the linux kernel. In *Proceedings of the 4th Conference on 4th Annual Linux Showcase & Conference, Atlanta - Volume 4* (Atlanta, Georgia, October 10 - 14, 2000). Atlanta Linux Showcase. USENIX Association, Berkeley, CA, 7-7.
 25. Rosenblum, M. and Garfinkel, T. 2005. *Virtual Machine Monitors: Current Technology and Future Trends*. Computer 38, 5 (May. 2005), 39-47. DOI=<http://dx.doi.org/10.1109/MC.2005.176>
 26. Wells, N. 2000. BusyBox: A Swiss Army Knife for Linux. *Linux J.* 2000, 78es (Oct. 2000), 10.
 27. Bauer, M. 2006. Paranoid penguin: Running network services under user-mode Linux, Part II. *Linux J.* 2006, 152 (Dec. 2006), 12.
 28. Bellard, F. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA, April 10 - 15, 2005). USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, 41-41.
 29. Alliance, A.-O.H., *Android – Open Handset Alliance Project source download Web pages*. <http://code.google.com/p/android/downloads/list>.