

國立清華大學資訊工程研究所
碩士論文

KVM-ARM之記憶體虛擬化設計

Memory Virtualization Design for KVM-ARM



學號： 9762592

姓名： 廖凡磊

Fan-Lei Liao

指導教授： 鍾葉青 教授

Dr. Yeh-Ching Chung

中華民國 一 百 年 七 月

摘要

系統虛擬化技術在大型電腦的應用已臻成熟,因此我們可以在同一個硬體上跑多個作業系統,然而這在嵌入式系統上還算是未成熟的領域,主要的原因是因為早期嵌入式系統的硬體設備較差,在自顧不暇的情況下,也就沒有辦法提供嵌入式系統運行多個作業系統的虛擬化平台,但就在嵌入式系統的處理器時脈逐漸拉高的同時,系統虛擬化技術在嵌入式系統上實現的可能性愈來愈高,我們希望在 Linux KVM 的原有架構下提供 ARM 系列的處理器一個系統虛擬化的平台,實現在嵌入式平台上運行多個作業系統的願望。在這個想法中,記憶體的虛擬化扮演一個重要的角色,目的有兩個(1)提供給虛擬機器記憶體空間,(2)在各個虛擬機器中有效的隔離,分配 以及共享記憶體資源。為了讓 guestOS 可以達成以上的目的,VMM(virtual machine monitor)會引進一個新的定址空間,稱為 GVA(guest virtual address)以及 GPA(guest physical address),然而這個定址空間不是真正的物理地址空間,需要再由 VMM 來 做轉換,所以有了兩層轉換,一層是 GVA 到 GPA 一層是 GPA 到 HPA,前者由 guest OS 來做,後者由 VMM 來做,為了降底這個轉換的負擔,我們將會採用 Shadow Page Table 的技術來達成。在這個技術下,VMM 可以在 guestOS 完全不知情的情況下,快速的通過 shadow page table 來進行地址轉換。在本篇論文中,我們將於 KVM 的基礎上實作在 ARMv6 的記憶體虛擬化,我們也會探討如何利用 rmap 來保持 shadow page table 與 guest page table 的一致性,最後利用數據來說明在使用了 rmap 機制之後,整體效能提升了多少。

Abstract

System virtualization is getting stable in servers recently, so we can run multiple operating systems on the same hardware in the same time. However, there are still some problems in embedded systems. With the increase of chip speed, the possibility of running system virtualization is getting higher. Before ARM provides hardware-assistance platform, we plan to propose a approach to implement system virtualization for ARM based on KVM. In system virtualization, memory virtualization plays a very important role. There are two goals we should achieve, one is to allocate memory resource during guest execution. The other one is to provide VM with separate and secure memory resources. In order to achieve those goals, VMM(virtual machine monitor) will introduce two new address spaces which are called guest virtual address and guest physical address. The translation from GVA(guest virtual address) to GPA(guest physical address) is done by guest, and the translation from GPA((guest physical address)) to HPA(host physical address) is done by VMM. To reduce the overhead of translation, we adopt a technique which is called shadow page table. Furthermore, we also propose a mechanism which is called rmap to keep the coherence between guest page table and shadow page table. In the thesis, we will also use LMBench and MiBench to evaluate the performance of KVM-ARM.

致謝

能夠完成這研究要感謝的人很多，首先我要感謝我的指導老師-鍾葉青教授，在我遇到研究困難的時候，適時地給予我指導，另外，我也要感謝在研究之路一直給予我協助的博班學長-俊宏，沒有他適時的拉我一把，我一定還是卡在無止盡的除錯輪迴中。

另外也要感謝跟我同期進到實驗室的祥葳，嘉宏及為超，藉由跟他們一起討論研究和課程，學習到了如何用不同角度來看事情，我相當珍惜與他們同窗的情誼。

在實驗室裡同計劃的士瑋是個相當有才華的學弟，感謝他在我遇到問題的時候撥空遇我討論，共同解決問題。家政，軒毅，世翔，Kenn，武君，伯其，敏娟，致穎，佳達，智傑，中昱，誌陞，朝柱，育穎，長融，雅婷感謝你們的共同砥礪，因為有你們的陪伴，讓我的研究生涯更顯豐富精彩。家豪以及弘斌是我高中時代的好同學，歡迎你們也加入了 SSLAB 的這個大家庭，雖然我先離開學校了，但我相信你們的研究之路一定也是一路順暢，沒有問題的。

再來我要感謝的是口試委員-金仲達教授以及徐慰中教授，感謝他們在百忙之中前來指導我的論文，而他們在我論文口試的時候所提供的寶貴意見更讓我受益良多。

最後，我更要感謝我的父母和女友-芊蕙，感謝你們一直在我背後的包容與支持，我謹以這研究成果來獻給你們。

西元 2011 辛卯年

凡磊于新竹清華

Content

1. Introduction.....	1
2. Related Work.....	3
2.1. Virtualization.....	3
2.2. Virtualization on X86.....	3
2.3. Intel VT-x	4
2.4. AMD-V	5
2.5. Virtualization on ARM.....	5
2.6. Xen on ARM	6
2.7. OKL4	6
2.8. Kernel-Based Virtual Machine.....	6
3. KVM on ARM	8
3.1. Environment.....	8
3.1.1 Hardware Environment	8
3.1.2 Software Environment	8
3.2. CPU Virtualization	9
3.2.1 Execution Model	9
3.2.2 Virtual Privilege Modes	10
3.2.3 Unpredictable sensitive instruction handling.....	11
3.2.4 Exception Handling	11
3.3. Memory Virtualization	12
3.4. IO Virtualization	12
3.4.1 IO emulation	13
4. Memory Virtualization Design Detail	14
4.1. Guest physical memory allocation.....	14
4.2. Address Translation	14
4.2.1 Shadow Page Table	14
4.2.2 Permission Model	18
4.2.3 Shadow Page Synchronization Model	21
4.3. MMU Fault Forwarding.....	22
4.4. Cache/TLB flush Model	23
4.5. Memory mapped IO	24
4.6. MMU-related functions for Guest OS	24
5. Experimental Results	26
5.1. Experimental Environment	26
5.1.1 Hardware Environment.....	26
5.1.2 Software Environment	26

5.2. Benchmark Introduction	26
5.2.1 Profiling counter	26
5.2.2 MiBench.....	28
5.2.3 LMBench	29
6. Conclusion and Future Work	30
6.1. Future work.....	30
6.2. Conclusion	30
7. Reference	31



Figure List

Figure 3.1: Execution Model in KVM-ARM	10
Figure 3.2: Relationship of KVM and QEMU.....	13
Figure 4.1: The Relationship of GVA and HPA	15
Figure 4.2: The default layout in shadow page table	16
Figure 4.3: The flow of filling in shadow page table.....	17
Figure 4.4: The page table entry layout in ARM-Linux	17
Figure 4.5: The Cache/TLB flush model in KVM-ARM	23
Figure 4.6: The flow of handling MMIO device in KVM-ARM.....	24



Table List

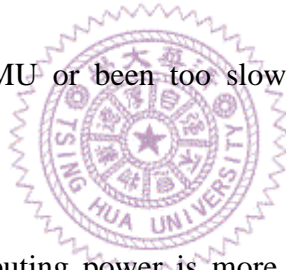
Table 4.1: The explanation of three Domain access	19
Table 4.2: The mapping table of Domain value in KVM-ARM.....	20
Table 4.3: The encoding of permission bits in ARMv6	20
Table 4.4 The shadow page table entry permission in KVM-ARM	21
Table 4.5: The classification of faults in KVM-ARM	23
Table 5.1: Profiling counter for fork + exit.....	27
Table 5.2 The experimental result of MiBench.....	28
Table 5.3 The experimental result of LMBench	29



1. Introduction

System virtualization is a technique to run guest virtual machines on a host physical machine. It has been applied in helping common operating system address problems such as kernel debugging [1], security-logging, honeypots, cluster-computing.

Since modern embedded system like mobile device contains much personal information. Thus embedded system can increase its security by leveraging virtualization's isolation characteristic. Any compromised guest OS cannot be propagated to other guest OS. In the past, this approaching is not practicable because embedded lacked support for dual-mode operation and an MMU or been too slow to accommodate the additional overhead of virtualization. [2]

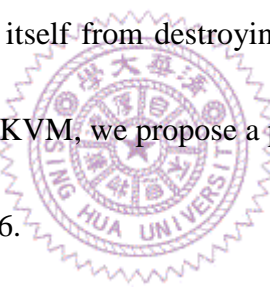


Due to the embedded computing power is more power, one physical machine is more capable of performing system virtualization. The TRANGO hypervisor [], Xen on ARM [3], L4 microkernel [4] and VLX [5] illustrate this trend. In order to investigate virtualization for embedded, we choose the most popular architecture-ARM and we choose KVM as our hypervisor.

Although KVM is not designed for ARM at begin, until in the middle of 2010, the ARMv7 extension starts to support hardware virtualization. However, currently the most widely used platforms are ARMv5 and ARMv6 that did not yet support virtualization. According to the virtualization requirements proposed by Popek and Goldberg in 1974,

ARM is not considered as a virtualizable architecture. Because there are a lot of non-privileged sensitive instructions which could not be trapped during guest execution. Traditionally, trap-and-emulation is adopted to handle those sensitive instructions.

For memory virtualization, the shadow page table implementation is essential when CPU utilizing virtual memory during guest execution. But because of the lack of hardware support, it is hard to be manipulated. First of all, the access permission and access attributes that are described in guest page table should be correctly emulated. Second, VMM should keep the coherences between guest page table and shadow page table. Third, VMM has to protect itself from destroying by guest OS. According to the above three requirements, base on KVM, we propose a practical approach to implement a virtual machine monitor on ARMv6.



In this thesis, we will focus on ARM virtualization technology and its implementation issues especially memory virtualization.

The rest of this thesis is organized as follows. Chapter 2 will describes the related work. Chapter 3 introduces KVM-ARM general porting issues. Chapter 4 presents the detail of memory virtualization we have done. Chapter 5 shows the experimental results of our work. Chapter 6 discusses the conclusions and future work.

2. Related Work

2.1. Virtualization

The term "virtualization" was coined in the 1960s, a term which is created from IBM's M44/44X project. The project is for the emerging time sharing concept. The other important term, virtual machine monitor(VMM), refers to software that virtualizes the system physical hardware and monitors virtual machine in which OS are executed. Sometimes, we call VMM a "hypervisor". The first hypervisor is Control Program 67(CP-67), which is originally used on IBM VM/370. It provides IBM machines with time-sharing functions.

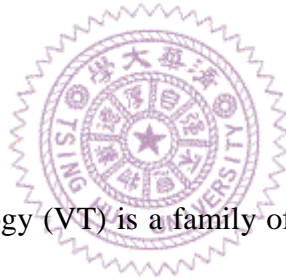


2.2. Virtualization on X86

At the begin, there is no hardware-assist in x86 virtualization. According to [6],x86 has some instruction which executed in user mode would not cause exception, so x86 is a non-virtualizable platform. Nevertheless, some companies still develop software- assist virtualization techniques. For example, the guest os should be run in user mode, so trap-and-emulate is the key technique to handle thoses instructions which can be skipped in user mode. Paravirtualization and Binary translation is also a classical technique to handle the same issue [7]. Although we can use software-assist to achieve virtualization, the drawbacks is that it brings about too much overhead.

Nowadays, Intel and AMD have provided additional hardware features to support full virtualization. AMD-V is AMD's virtualization technology trademark and it is first added in Athlon 64 family on May 23, 2006. Intel VT-x plays a good role on segmenting its market [8]. VMware is the most famous x86-based virtualization company, which was founded in 1998. It supplies enterprise and person kinds of virtualization services. The other competitor is Xen which is supported by many computer manufactures including Sun, HP and IBM. Microsoft also release Virtual PC series to fulfill custom virtualization requirement.

2.3. Intel VT-x



Intel Virtualization Technology (VT) is a family of technologies supporting virtualization on Intel IA-32, Xeon, and Itanium platforms. It includes elements of support for CPU, memory, and I/O virtualization, and guest migration. In intel VT-x, there is a new execution mode called VMX root operation mode. The hypervisor will run in VMX root mode, whereas virtual machines will not. When privileged instructions executed outside VMX root mode, the exception will be invoked. Since a guest OS can still run at ring 0, this will alleviate any problems arising from a guest OS running lower privilege but expecting to running at ring 0. Additionally, Extended Page Tables help x86 support memory virtualization. In software technique, VMM have to maintain a physical

mappings for each guest VM. Thus, VMM will suffer the overhead of synchronizing the guest page table and shadow page table. With EPTs, there are two page tables, one page table translates from "guest virtual address" to "guest physical address", the other one translates from "guest physical address" to "host physical address". Therefore, guest OS is free to access memory without needing to trap to the VMM and it will speed it up.

2.4. AMD-V

AMD's version of virtualization support is entitled AMD-V[8], and offers comparable support for CPU, memory, and I/O virtualization, and migration. AMD-V has a new CPU mode called "guest mode" which is analogous to non-VMX root mode in Intel VT-x. Just like Intel's EPT, Nested Paging is AMD's version of hardware support for memory virtualization. With Nested Paging, VM will have dramatically performance increasing comparing to traditional software mechanism.

2.5. Virtualization on ARM

Nowadays, there are approximately 90% of all embedded 32-bit RISC processors designed by ARM-based CPU. And the computing ability of those embedded devices is more powerful than before. The need of security and fault-tolerance are increasing. Although ARM can't support full virtualization because of the HW assistance, there are still several para-virtualization softwares such as Xen-On-ARM, OKL4.

2.6. Xen on ARM

Originally, Xen is a virtual machine monitor for IA-32, Itanium and PowerPC 970. It provides a platform to run several guest OS on the same hardware concurrently. On most CPUs, if a guest OS wants to run on Xen, it should be modified. This kind of virtualization technique is called paravirtualization. Paravirtualization makes Xen achieve high performance even on its host architecture.

Since 2006, Xen community has started to port Xen to ARM. Samsung also wanted to include Xen-ARM in their products. They try to increase the security of ARM mobile devices by Xen-ARM

2.7. OKL4

[9] introduced the OKL4 hypervisor running on a ARM-based Windows CE devices and compared the performance with Xen-On-ARM. OKL4 is a branch of L4 microkernel family. In the paper, Heiser mentioned OKL4's main feature is real-time support and compact code size. Typically, the size of code implies how far from faults and errors [9]. Because of its real-time feature, L4-based hypervisor is suitable to meet the requirement of virtualization with real-time support.

2.8. Kernel-Based Virtual Machine

KVM is original a virtualization solution for Linux on X86 platform. It utilizes the virtualization technique on hardware, e.g. Intel VT and AMD-V. The system



architecture of KVM is shown in Figure 1. There are two components in KVM, the Linux kernel modules and the user-space modified QEMU. The KVM modules involving the core virtualization infrastructure and processor-specific features are included in the mainline Linux kernel from 2.6.20. When KVM modules are loaded into the Linux kernel, KVM makes Linux act as a hypervisor and has the capabilities to host guest virtual machines on it. In addition, KVM requires QEMU-KVM to provide various virtual I/O devices.



3. KVM on ARM

Because KVM is originally running on X86, we should modify the hardware specific code in the KVM, including CPU registers management, exception handlers and memory handling function. In this chapter, we will discuss the KVM porting issues we have done in KVM-ARM.

3.1. Environment

Before we start to discuss the porting issue, we have to define the development environment.

3.1.1 Hardware Environment

The following hardware environment is what we used:

CPU: Realview ARM11 MPCORE

Instruction Set: ARM v6

Platform: Realview Emulation Board

3.1.2 Software Environment

The following software environment is what we used

Host Kernel: linux 2.6.31

Guest Kernel: linux 2.6.31

KVM: KVM88

Root File System: Busybox 1.11.2

Tool-chain: Sourcery G++ Lite 2008q3-72

3.2. CPU Virtualization

We have to virtualize the CPU because we make guest OS run on a user mode for the security. When be in the user mode, the CPU can't and shouldn't execute some privilege instructions. How to emulate those privilege instruction is a important problem. Not only host but also guest have exceptions, so we must distinguish and dispatch those exceptions. Further more, ARMv6 has 7 execution modes including user mode, system mode, supervisor mode, abort mode, irq mode, fiq mode and undefined mode, so there are several banked registers in respective mode. When a guest OS which is always running in user mode wants to query a banked register, we must emulate this request.

3.2.1 Execution Model

On the whole, Guest OS native runs in the physical hardware most of the time instead of emulation. When any exception occurs from Guest OS, Guest OS will call exit guest function to backup its running state such as general purpose registers and PC. And then it will jump to KVM exception handlers. After KVM analyzes the cause of exceptions, it will take the relative steps such as injecting exceptions into guest or

emulating privileged instructions. There are two ways of leaving KVM for resuming guest. One of them is lightweight exit, and the other is heavyweight exit. Lightweight exit directly recovers Guest's running state for most of exception handling. Heavyweight exit returns to QEMU only for MMIO emulation. After MMIO emulation, QEMU will use IOCTL to inform KVM hypervisor. And then KVM resumes Guest via enter guest function.

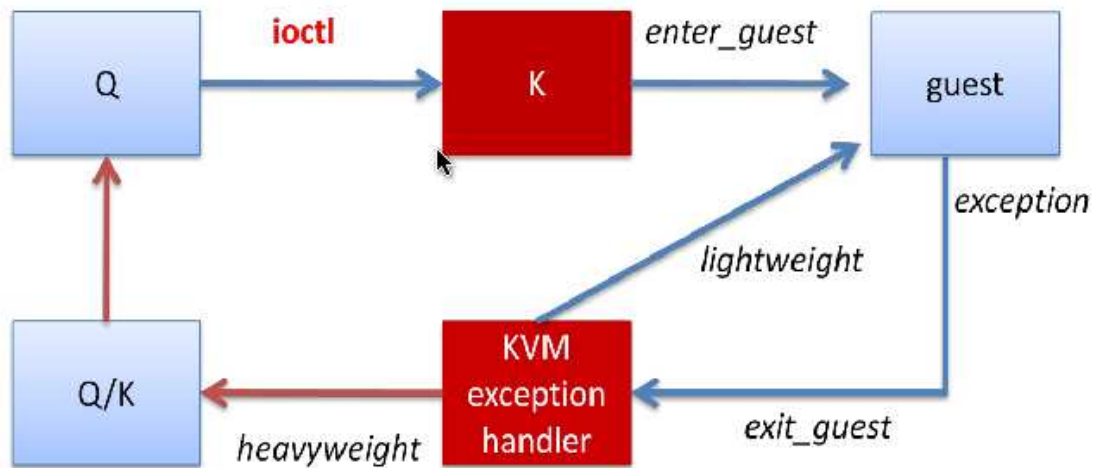


Figure 3.1: Execution Model in KVM-ARM

3.2.2 Virtual Privilege Modes

In order to deprive guest OS and allow full resource control to VMM, only VMM runs in the supervisor mode and guest OS runs in user mode. We provide an abstract supervisor mode to guest OS kernel. User mode is split into two logical modes (user process mode and kernel mode), and virtual banked registers for those virtual privilege modes. KVM-ARM will be in charge of switching between the user process mode and kernel modes.

3.2.3 Unpredictable sensitive instruction handling

Since guest OS is run in user mode, all sensitive instructions should be trap-ed and emulated. Unfortunately, there are 14 sensitive instructions in ARM v5 architecture, which fails executing quietly when in user mode. For example, when linux wants to create a critical section, it will use MRS/MSR instructions to modify the interrupt flag bits of CPSR. Those instructions fail without any error report. Thus, we should add a software interrupt instruction before those unpredictable sensitive instructions.

3.2.4 Exception Handling

When exception occurs, we should recognize the origin and dispatch it to the right handlers. In KVM-ARM, guest OS and VMM share the same exception stubs of VMM. When an exception occurs, the event will be intercepted by KVM and then dispatch the event according to a global variable which is called `vcpu->flag`. The flag will be updated every time when the system entry guest or host. Because we remap the exception stubs in dynamic time, KVM-ARM can handle all exception without modifying guest exception handling code by hand. It reduces the overhead to run a guest OS.

3.3. Memory Virtualization

Generally, the VMM should provide a virtual memory space for the guest virtual machines and protect itself from illegally accessing by any guest virtual machine. To achieve these goals, the implementation of memory virtualization is harder than CPU virtualization. First of all, a guest VM expects a continuous physical address space like in a real hardware. Therefore, an additional address translation layer should be introduced. Then the guest VM can convert a guest-virtual address to host-physical address by this layer which is also called shadow page table(SPT). Furthermore, the maintenance of shadow page table is also important. In the KVM-ARM, we adopt a simple synchronization mechanism to make shadow page table up-to-date. Traditionally, translation lookaside buffer (TLB) is used to speed up the process of virtual-address-to-physical-address. The introduction of shadow page table may confuse the translation of guest virtual address and the translation of host virtual address, so we will present a simple TLB flush model to avoid this kind of translation confusion problem. When a guest VM runs as a virtual machine, we want to protect guest kernel memory space from destroying by guest user-mode process. The ARM platform's domain mechanism will be adopted as a protection. The design detail of memory virtualization will be described in chapter 4.

3.4. IO Virtualization

3.4.1 IO emulation

The I/O emulation is left to QEMU. KVM will map MMIO address between guest OS and QEMU, so we can dispatch interrupts from hardware to QEMU. And the interrupt will be injected from QEMU to guest. The relationship between KVM and QEMU is shown in following figure.

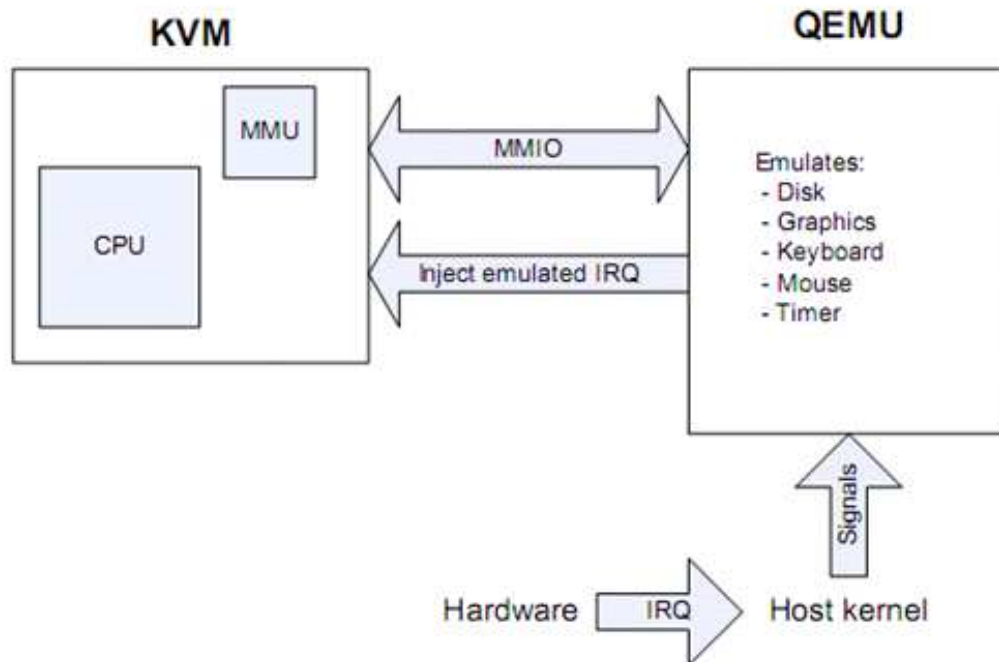


Figure 3.2: Relationship of KVM and QEMU

For example, if guest OS wants to read a timer register, it will invoke a MMIO trap because there is no mapping entry in our shadow page table. And then KVM- ARM will call QEMU to emulate those IO manipulation instructions and get timer value. After that, KVM-ARM will save the timer value into virtual register. Finally, the guest OS will get the timer value by restoring the virtual state.

4. Memory Virtualization Design Detail

4.1. Guest physical memory allocation

Generally, the guest physical memory can be allocated by static method or dynamical method. By static approach, VMM will reserve a continuous bunk of host physical memory space for a VM when a VM is enabled initially. Therefore, the size of memory resource will not be adjusted dynamically. In contrast, the VMM will gain benefit of flexibility in the dynamical method, because they will allocate host physical memory for a VM on demand. Currently, KVM-ARM adopts the original design philosophy from KVM-x86.



4.2. Address Translation

4.2.1 Shadow Page Table

In ARMv6, the TTBR(Translation Table Base Register) is used to indicate the head of a page table. Then a virtual address is transferred to a machine address by walking page table. Now we will introduce two new address spaces called guest virtual address space and guest physical address space in the system virtualization. And the guest page table is the translation table between guest virtual address and guest physical address. In the same word, the host page table is the translation table between host virtual address

and host physical virtual address.

As we mentioned before in chapter 3, currently, there is no hardware-assist virtualization in ARM, the transfer from guest virtual address to host physical address is not build in CPU, so we need to build a shadow page table to make guest OS use physical memory naturally. The shadow page table is the mapping table from guest virtual address to host physical address. Figure4.1 shows the translation relationship between guest virtual address and host physical address.

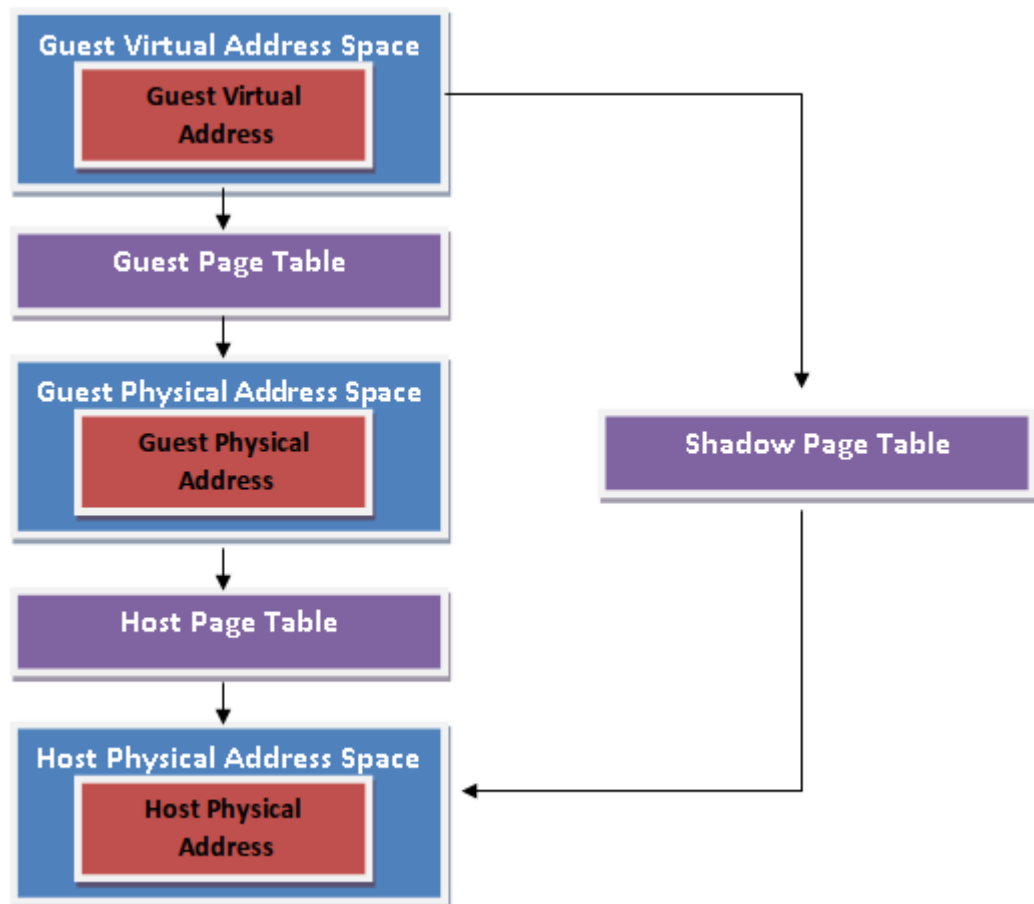


Figure 4.1: The Relationship of GVA and HPA

In our design, when a shadow page table is build, there are two entries will be

assigned automatically. One is for exception vector table, and the other one is for share-variable page such as `vcpu->vcpu_flag` which is used to recognized guest side or host side. The rest of shadow page table will be blank. When guest OS try to access a memory region which is not yet mapped in shadow page table, a translation fault will be trapped by KVM-ARM. The KVM-ARM will fill host physical address in the corresponding shadow page table entry after looks up guest page table and host page table. After filling up shadow page table entry, the guest OS can access the memory region successfully.

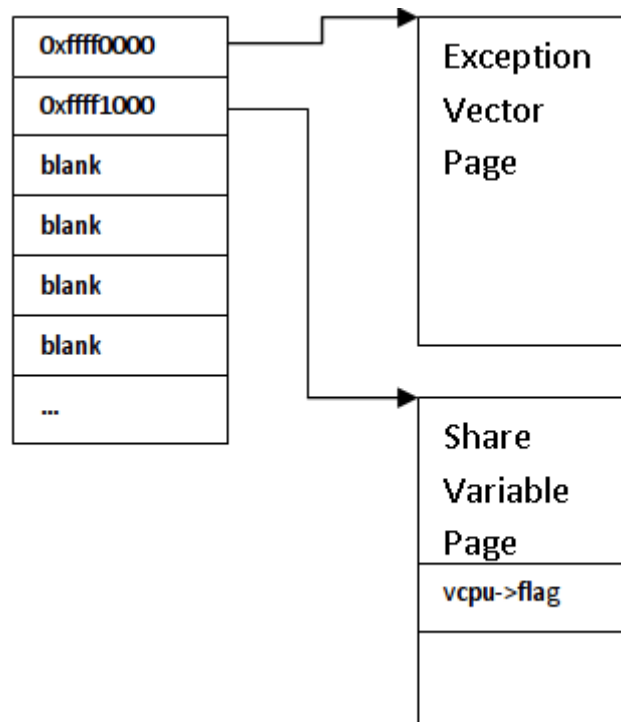


Figure 4.2: The default layout in shadow page table

From the view of linux, a page table entry has more information than an ARM page table entry has. Further more, in ARM architecture, the first level contains 4096 entries and the second level consists of 256 entries. Thus, linux kernel will prepare 16KB size to store all first level page table descriptor and store two second level linux page table

entries and two second ARM page table entries in a 4K page. The relation is described in Figure4.4 In our design, we store 16KB first level descriptors in four 4KB shadow pages and store one second level descriptor in a 4K shadow page. Hence, it will waste 3KB shadow page size for a second level descriptor.

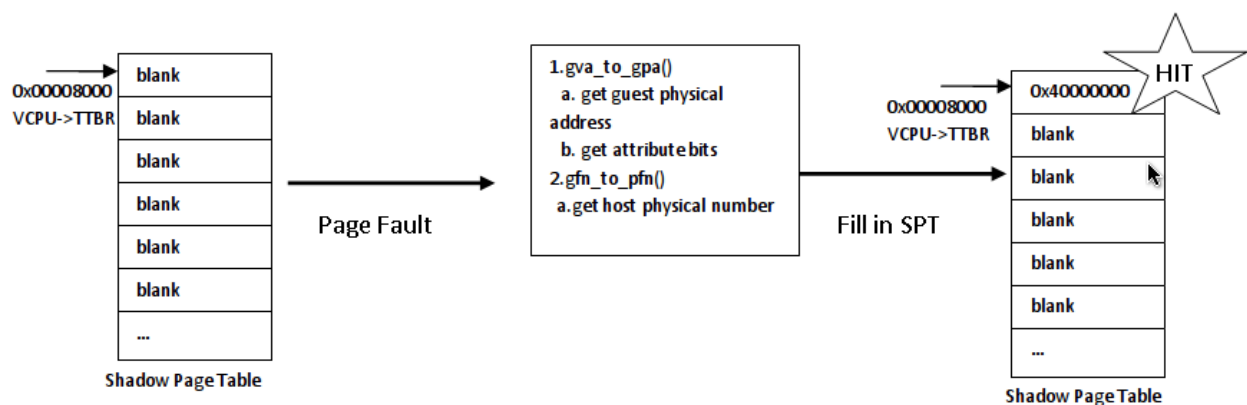


Figure 4.3: The flow of filling in shadow page table

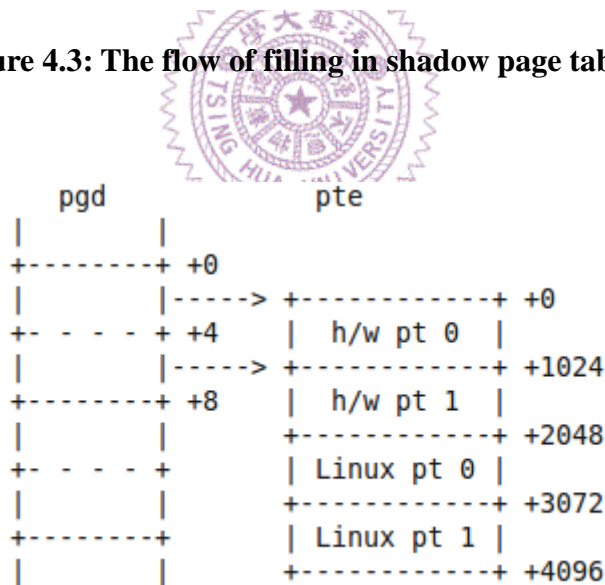


Figure 4.4: The page table entry layout in ARM-Linux

From the view of ARMv6, a first-level descriptor can describe different size mem-
memory region, for example, a section type can describe a 1MB continuous physical size and
a supersection type can describe a 16MB continuous physical size. But it is not
guaranteed for KVM-ARM to get a continuouse physical space under current memory

allocation system. The temporary way is to force first-level descriptor in shadow page table to be coarse page type. Then a second-level descriptor will describe a 4KB physical page. Therefore, it costs 256 entries to describe a section type first-level descriptor and 4096 entries describe a supersection type.

4.2.2 Permission Model

There are three ways which are used in accessing to a memory region. They are domain, access permissions and execute never bits. A domain is a collection of memory regions. The ARM architecture supports 16 domains. Domains provide support for multi-user operating systems. All regions of memory have an associated domain. Each page table entry and TLB entry contains a field that specifies which domain the entry is in. Access to each domain is controlled by a 2-bit field in the Domain Access Control Register, CP15 c3. Each field enables very quick access to be achieved to an entire domain, such that whole memory areas can be efficiently swapped in and out of virtual memory. Three kinds of domain access are described in table 4.1:

In our design, we will leverage ARM domain mechanism to make system safe. We can assign guest kernel memory as DOMAIN GUEST KERNEL, and when we switch the control to guest user process, we can set DOMAIN GUEST KERNEL value as DOMAIN NOACCESS. While the control is translated to guest kernel, we can reset DOMAIN GUEST KERNEL value as DOMAIN MANAGER. The mapping domain value is in

table 4.2.

Table 4.1: The explanation of three Domain access

Clients	<p>Clients are users of domains in that they execute programs and access data. They are guarded by the access permissions of the TLB entries for that domain. A client is a domain user, and each access must be checked against the access permission settings for each memory block and the system protection bit, the S bit, and the ROM protection bit, the R bit, in CP15 Control Register c1.</p>
Managers	<p>Managers control the behavior of the domain, the current sections and pages in the domain, and the domain access. They are not guarded by the access permissions for TLB entries in that domain. Because a manager controls the domain behavior, each access requires only to be checked to</p>

Noaccess	Noaccess
----------	----------

Table 4.2: The mapping table of Domain value in KVM-ARM

Current context in	Domain Value
Guest user process	GUEST KENEL SPACE:DOMAIN CLIENT GUEST USER SPACE:DOMAIN CLIENT
Guest kernel	GUEST KEREL SPACE: DOMAIN MANAGER GUEST USER SPACE: DOMAIN CLIENT

Table 4.3: The encoding of permission bits in ARMv6

NO	APX:AP	Privilege Mode	User Mode
1	0:00	No access	No access
2	0:01	RW	No access
3	0:10	RW	RO
4	0:11	RW	RW
5	1:01	RO	No access
6	1:10	RO	RO

The access permission bits control access to the corresponding memory region. If an access is made to an area of memory without the required permissions, then a permission fault is raised. Table 4.3 shows the encoding of permission bits in ARMv6.

In our design, because we want to make sure VMM can get complete control in privilege mode, the permission bits in shadow page table is different from original guest page table. Figure 4.5 presents the permission bits we use in shadow page table.

Table 4.4 The shadow page table entry permission in KVM-ARM

Guest page table			Shadow page table		
priv.	user	APX:AP	priv.	user	APX:AP
NA	NA	0:00	RW	NA	0:01
RW	NA	0:01	RW	NA	0:01
RW	RO	0:10	RW	RO	0:10
RW	RW	0:11	RW	RW	0:11

4.2.3 Shadow Page Synchronization Model

When guest OS wants to modify their own guest page table entry, we have to update the corresponding shadow page table entry automatically. For example, linux kernel will modify the page table entry after a program exiting to show the program no longer exists. Or by modifying the page table entry to implement COW(copy on write) mechanism.

The basic design is that we will set all shadow page table entry of those guest page table as read only, so the system will invoke a trap when those guest page table entries are modified, and then KVM-ARM will aware and destroy all shadow page table. Then the

new shadow page table entry could be rebuild.

To reduce the overhead of synchronization, we want to introduce a memory trace technique that is called RMAP. First of all, when a shadow page table entry is build, a data structure which is called RMAP(reverse map) is also created. Then if the shadow page table entry directs to a page which consists of guest page table. KVM-ARM will use the RMAP to trace back the shadow page table entry and set shadow page table entry as READ OLNY in privilege mode and READ ONLY in user mode(APX:AP=1:10). When guest system try to modify guest page table entry, for example forking a new process, a permission fault will be trapped and then KVM-ARM will trace guest page table entry back to shadow page table entry and then modify SPTE in the same time. That is, it will assure shadow page table of update-date.

4.3. MMU Fault Forwarding

Since we introduce shadow page table layer into our design, when a virtual machine runs on KVM-ARM, there are several faults which will be generated in different situations. Every time, when a fault is caused by a system operation, KVM-ARM will check the source of this fault and then handle it appropriately. For example, a translation fault will be triggered because of two reasons, one is a true translation fault because there is no corresponding entry in current guest page table, we call it as true page fault. The other reason of a translation fault is caused by the current shadow page table, we call it as

hidden page fault.

Table 4.5: The classification of faults in KVM-ARM

Fault types	Reasons	Handling part
True translation faults	No entry in guest page table	Guest kernel
True permission faults	Copy on write for creating new process	Guest kernel
Hidden translation faults	No entry in shadow page table	KVM-ARM
Hidden permission faults	Protection for updating SPT	KVM-ARM

4.4. Cache/TLB flush Model

During the guest execution, the traps or interrupts will make guest suspend and then VMM in host will decide how to handle them. In these situations, the cache and TLB should be invalidated before entering host VMM or guest. Figure 4.7 presents the flush model during the interleave of host and guest.

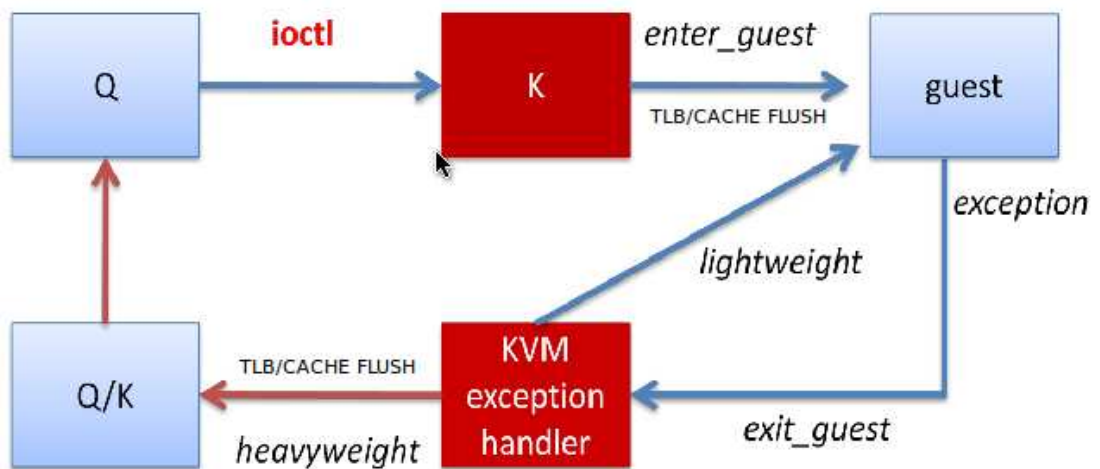


Figure 4.5: The Cache/TLB flush model in KVM-ARM

4.5. Memory mapped IO

For Memory mapped IO device, what we do is to keep the shadow page table entry of those corresponding MMIO address blank. When a guest read or write a MMIO address, a translation fault will be intercepted by KVM-ARM and then KVM-ARM will switch to QEMU for I/O emulation. Finally, the guest can get the value from the emulated devices in QEMU. The figure 4.8 illustrates the flow of handling memory mapped IO device.

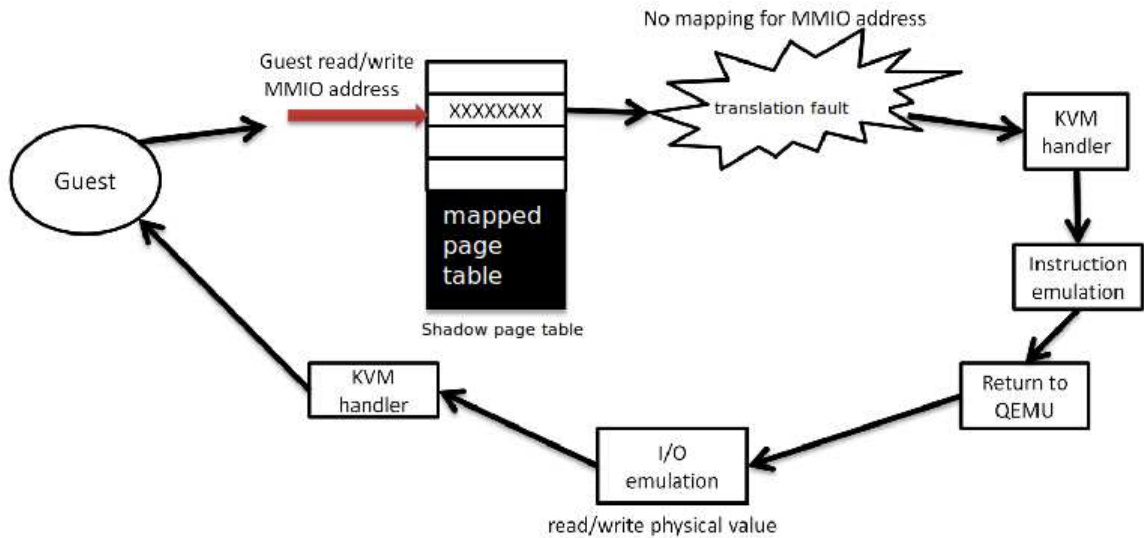


Figure 4.6: The flow of handling MMIO device in KVM-ARM

4.6. MMU-related functions for Guest OS

Except the guest physical memory allocation and shadow paging mechanism, there are some other MMU-related functions such as TLB operations, cache operations, enabling/disabling MMU, resetting Domain register, reading FSR/FAR, changing ASID and loading TTBR during guest execution. Those operations are controlled by coprocessor in ARM platform. In KVM-ARM, we should emulate those operations correctly.



5. Experimental Results

5.1. Experimental Environment

5.1.1 Hardware Environment

The following hardware environment is what we used:

CPU: Realview ARM11 MPCORE

Instruction Set: ARM v6

Platform: Realview Emulation Board

5.1.2 Software Environment

The following software environment is what we used

Host Kernel: linux 2.6.31

Guest Kernel: linux 2.6.31

KVM: KVM88

Root File System: Busybox 1.11.2

Tool-chain: Sourcery G++ Lite 2008q3-72

5.2. Benchmark Introduction

5.2.1 Profiling counter

Though the understanding of KVM, we define some counters for profiling. In the

table 5.1, we can see the items including the True Guest dabt which means the data abort occurred because of the miss of guest page entry, the Hidden dabt exits which means the data abort occurred because of the miss of shadow page entry. Though the profiling counters, we cannot only find the bottleneck of our design, but we can also find how much it improves when we apply a new improvement. For example, in the table 5.1, the one without RMAP have twice times then the one with RMAP in hidden pabt and hidden dabt.

Table 5.1: Profiling counter for fork + exit

	KVM-ARM with RMAP	KVM-ARM without RMAP
True Guest dabt:	370	291
True Guest pabt:	266	220
Hidden pabt exits:	5072	12966
Hidden dabt exits:	7060	16453
Guest PGT/PGD protection write fault	1508	598
Guest hidden translation fault	10572	28778

5.2.2 MiBench

MiBench [10] is a series of applications which are used to measure the performance of embedded system. We choose the Automotive and Industrial Control and Decode/Encode JPEG as our benchmarks to compare the performance between full emulation solution and virtualization solution. The result of experiment is following.

Table 5.2 The experimental result of MiBench

UNIT: second	QEMU Full emulation	KVM-ARM without RMAP	KVM-ARM with RMAP	Native-Run Linux
Basic math small	148.54	5.49	4.42	2.97
Basic math large	> 25 minutes	35.15	35.18	39.11
Bit count small	11.63	2.65	2.6	0.53
Bit count large	145.36	5.56	5.68	3.29
Quick sort small	11.2	4.67	4.55	0.61
Quick sort large	298.64	12.35	12.76	6.21
Susan small	13.38	5.1	4.96	0.72
Susan large	130.48	11.6	11.25	3.43
Encode JPEG small	13.9	1.98	2	0.53
Decode JPEG small	4.53	1.62	1.6	0.41
Encode JPEG large	36.18	4.19	4.15	1.47
Decode JPEG large	10.4	2.71	2.67	0.89

As shown above, those basic applications can all run successfully in our system virtualization solution. And we can see the speed of KVM-ARM is much faster than full emulation(QEMU). Furthermore, the improvement of RMAP is negligible, it is because those application is less about memory operation. In the next experiment, we can see the benefit of RMAP.

5.2.3 LMBench

LMBench is a set of benchmarks which is often used to evaluate the performance of systems. It consists of numerous system operations such as system call, IPC, process creation and signal handling. According to the paper [11], we choose a set of benchmarks to measure performances of virtual machines under KVM-ARM comparing to linux kernel in native environment.

Table 5.3 The experimental result of LMBench

UNIT:microseconds	KVM-ARM with RMAP	KVM-ARM without RMAP	Speedup
lat_syscall null	844.824	850.5775	0.676422783
lat_syscall read	878.8947	885.8539	0.785592297
lat_syscall write	876.4447	854.2291	-2.600660642
lat_syscall null	1477.3806	1434.8451	-2.964466339
lat_syscall stat /dev/zero	852.5986	869.2968	1.920885939
lat_syscall fstat /dev/zero	3282.6602	3290.0674	0.225138245
lat_sig install	1007.7303	1034.6014	2.597241798
lat_sig catch	7428.4847	7455.1133	0.357185718
lat_proc fork	600645	1020490	41.14151045
lat_proc exec	1448875	2503247	42.12017432
lat_proc shell	2579672	4402136	41.39953877
lat_pagefault /tmp/rootfs.bin	4409.8337	9900.3399	55.45775454

As shown above, the process-creation benchmark such as lat_proc fork, lat_proc exec and lat_proc shell can gain much improvement after we apply RMAP, it is because those applications are memory-intense. The design without RMAP causes a lot of hidden page faults, because KVM-ARM will destroy all shadow page tables when one entry is out-of-date. After we introduce RMAP, the overhead of synchronization can be reduced to up to about 50%.

6. Conclusion and Future Work

6.1. Future work

Currently, we can create a VM contained linux guest OS and could run some tasks on the guest OS. Because there are no hardware support in our virtualization implementation, the performance of guest execution is not good enough comparing to native linux. In the future, we will leverage the ARMv6's new feature to reduce the overhead of the switch cost of guest and host. It is Application Space Identifiers (ASIDs), which allows several TLB entries with the same virtual address, but belonging to different address spaces, to stay in the TLB at the same time. It is a method to lower the frequency of invalidating TLB.

The multiple processors is the trend in embedded system,

6.2. Conclusion

In this thesis, we described the design and implementation of KVM-ARM. We also discuss the issues when porting KVM on ARM architecture especially about memory virtualization. Furthermore, to increase the performance, we also introduce RMAP to speed synchronization up. Thus we use profiling counter, LMBench and MiBench to verify our work. According to the data, we got up to about 50% speed up by introducing RMAP structure.

7. Reference

- [1] S.T.K, “Debugging operating systems with time-traveling virtual machines,”
- [2] L. P. Cox and P. M. Chen, “Pocket hypervisors: Opportunities and challenges,” in HOTMOBILE '07: Proceedings of the Eighth IEEE Workshop on Mobile Computing Systems and Applications, (Washington, DC, USA), pp. 46–50, IEEE Computer Society, 2007.
- [3] H. Joo-Young, S. Sang-Bum, H. Sung-Kwan, P. Chan-Ju, R. Jae-Min, P. Seong-Yeol, and K. Chul-Ryun, “Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones,” in Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE, pp. 257–261, 2008.
- [4] D.-G. Kim, S.-M. Lee, and D.-R. Shin, “Design of the operating system virtualization on l4 microkernel,” in NCM '08: Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management, (Washington, DC, USA), pp. 307–310, IEEE Computer Society, 2008.
- [5] F. Armand and M. Gien, “A practical look at micro-kernels and virtual machine monitors,” in CCNC'09: Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference, (Piscataway, NJ, USA), pp. 395–401, IEEE Press, 2009.
- [6] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third

generation architectures,” Commun. ACM, vol. 17, no. 7, pp. 412–421, 1974.

[7] VMware, “Vmware and hardware assist technology(intel vt and amd-v),” 2006.

[8] J. Stokes, “Microsoft, intel goof up windows 7’s xp mode,” 2009.

[9] G. Heiser, “Hypervisors for consumer electronics,” in Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE, pp. 1 –5, 10-13 2009.

[10] D. E. T. M. A. T. M. R. B. B. Matthew R. Guthaus, Jeffrey S. Ringenberg [11] E. G.

S. T. K. M. A. B. Yang Xu, Felix Bruns

