

分类号

密 级

UDC

编 号

# 成都理工大学

## 硕士学位论文

题名和副题名 QEMU-KVM 设备虚拟化研究与改进

作者姓名 车 翔

指导教师姓名及职称 王华军 教授

申请学位级别 硕士 专业名称 计算机软件与理论

论文提交日期 2012.5 论文答辩日期 2012.5

学位授予单位和日期 成都理工大学 (2012 年 6 月)

答辩委员会主席 苗放

评阅人 苗放 李宏福

2012 年 5 月



分类号\_\_\_\_\_

学校代码: 10616

UDC \_\_\_\_\_

密级\_\_\_\_\_ 学号: 2009020550

## 成都理工大学硕士学位论文

# QEMU-KVM 设备虚拟化研究与改进

车翔

指导教师姓名及职称 \_\_\_\_\_ 王华军 教授

申请学位级别 \_\_\_\_\_ 硕士 \_\_\_\_\_ 专业名称 \_\_\_\_\_ 计算机软件与理论

论文提交日期 \_\_\_\_\_ 2012 年 5 月 \_\_\_\_\_ 论文答辩日期 \_\_\_\_\_ 2012 年 5 月

学位授予单位和日期 \_\_\_\_\_ 成 都 理 工 大 学 (2012 年 6 月)

答辩委员会主席 \_\_\_\_\_ 高放

评阅人 \_\_\_\_\_ 高放

2012 年 5 月

## 独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的  
研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其  
他人已经发表或撰写过的研究成果，也不包含为获得 成都理工大学 或其他教  
育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何  
贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：

车翔

2012年 6月 1日

## 学位论文版权使用授权书

本学位论文作者完全了解 成都理工大学 有关保留、使用学位论文的规定，  
有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和  
借阅。本人授权 成都理工大学 可以将学位论文的全部或部分内容编入有关数  
据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名：

车翔

学位论文作者导师签名：

车翔

2012年 6月 1日

# QEMU-KVM 设备虚拟化研究与改进

作者简介：车翔，男，出生 1986 年 03 月，师从成都理工大学王华军教授，2012 年 06 月毕业于成都理工大学计算机软件与理论专业，获得工学硕士学位。

## 摘 要

随着物联网、云计算等学科的迅速发展, 虚拟化技术又迎来了一个新的春天, 特别是在竞争激烈的今天, 实现资源的共享和高效利用已经成为了一个新的主题。虚拟机就是充分利用设备虚拟化来模拟出一套完整的硬件平台。通过分时处理的方法将机器硬件进行共享, 并且抽象出一个中间层将操作系统和硬件环境相分离, 因此提高了计算机硬件的使用效率, 增强了硬件的兼容性。但是随着计算机硬件技术的不断发展, Intel、AMD 等硬件厂商设计出了多种硬件虚拟化技术来支持虚拟机高效运行, 所以怎么充分使用这些硬件辅助虚拟化技术将成为一个新的挑战。

本文研究基于当今非常流行的开源虚拟机 KVM (Kernel-based virtual machine), 通过对 KVM 进行二次开发和改进, 实现了提出的虚拟化应用解决方案。通过将纯 QEMU 和经过修改的 QEMU-KVM 进行性能对比, 表明利用 Intel VT 硬件辅助虚拟化技术加速过的系统性能有明显的提升。在研究中, 本文具有以下两个创新点:

1. 提出了一个时钟设备的虚拟化改进方案。

本文通过对中断和时钟虚拟化的研究, 借助 Intel VT 技术, 通过保存-再触发方式解决时钟中断的注入丢失问题, 并且通过批量注入方式来提高实现时钟中断的注入效率, 减小系统损耗。

2. 提出了一个固定虚拟 PCI 设备 I/O 基地址的改进方法。

本文通过对虚拟 PCI 设备地址初始化进行拦截和修改, 实现给指定的设备固定分配 I/O 基地址。

基于以上的创新点研究以及虚拟化技术的基础研究, 本文取得了以下研究成果。

1. 实现了一个硬件管理器的虚拟化。

本文针对 Intel VT 硬件辅助虚拟化技术, 将 QEMU 中的硬件模拟部分转移到 KVM 中实现, 完成了一个硬件管理器的虚拟化实例, 同时提出了一个完整的设备虚拟化应用解决方案。

2. 完成了系统时钟和基于 VT-x 的虚拟中断研究, 提出了时钟设备改进方法。

提出了一个在虚拟化过程中减少虚拟时钟丢失的改进方法。通过将虚拟时钟丢失中断进行保存和再触发以及虚拟中断批量注入来解决时钟丢失问题。

### 3. 完成了对 QEMU-KVM 中设备 I/O 虚拟化的研究。

本文一方面对 QEMU 中的虚拟设备模拟进行研究,同时另一方面对 KVM 中的设备注册和 I/O 拦截进行研究,通过两个方面的结合,实现了将设备虚拟化 I/O 部分从 QEMU 中转移到 KVM。

目前国内外研究虚拟机技术的资料非常少,特别是针对当前硬件辅助虚拟化技术的资料更是稀少,本文作为这方面的研究,提出了一些自己的理论,弥补了设备虚拟化中理论的空白,为更深一步研究虚拟机技术提供了理论的依据。同时虚拟机技术也是云存储、物联网等技术的基础,它为系统共享硬件资源提供了有效保证,有着广阔的发展空间。

**关键字:** QEMU KVM Intel VT 技术 虚拟化 云计算 物联网

# **Research and Enhance of QEMU-KVM device virtualization**

Introduction of the author: Che Xiang, male, was born in March, 1986 whose tutor was Professor Wang Huajun . He graduated from Chengdu University of Technology in Computer Software and Theory, major and was granted the Master Degree in June, 2012.

## **Abstract**

With fast developed of the internet of the things and cloud computing technology, the virtualization encounter a new spring. To share and efficiently use the resource is becoming a new topic especially in the highly competitive today. Virtual machine is aim to make good use of virtualization technology to simulate a complete hardware platform. They share the hardware by time sharing and abstract a mesosphere to separate the operation system and the hardware. So it promoted the efficiency of usage of hardware and enhanced the compatibility of the hardware. However, the hardware factories such as Intel have provided all kinds of hardware virtualization technology to support the virtual machine follow the development of software virtualization technology. As a result, how to make good use of the new hardware virtualization technology will be a new challenge.

The research of this project is based on the open-source virtual machine (KVM) that was popular used in current day. It implement the methods which were raised in this paper by the secondary development and improvement of KVM, It also proved that the system performance of virtual machine with Intel VT support has a evident enhance by compare the original QEMU to modified QEMU-KVM. And this paper has the following Innovations.

1. Raised a virtualization improvement for timer device.

This paper focuses on the research of interrupt and timer device. With the Intel VT technology, it solved the timer lost problem by saving and retriggering the time interrupt and solved the low performance problem by inject all the rest time interrupt at the same time.

2. Raised an improvement to fix the I/O address of PCI device.

This paper implements the solution by capturing and modifying the initialization

operation of virtual PCI device

This paper got the follow achievements base on the research of the below innovations and basic virtualization knowledge.

1. Implement a virtualization sample of hardware manager.

This paper focuses on the Intel VT hardware virtualization technology, and moved the hardware simulation part from QEMU to KVM, it also achieved to implement a virtualization sample of hardware manager. At the same time, it raised a complete application solution of device virtualization.

2. Achieve the research of system timer and virtual interrupt which was based on VT-x, and raise a improvement for timer device.

This paper raised an improvement for reducing the lost time interrupt in the virtualization produce. And solved the problem by saving and retriggering the time interrupt and inject all the rest time interrupt at the same time.

3. Achieve the research of I/O virtualization in QEMU-KVM.

This paper researched the simulation of virtual device in QEMU on one hand, simultaneously, researched the device register and I/O capture in KVM on the other hand. At last, it combined the two sides to move the I/O part in device virtualization from QEMU to KVM.

There is little material on virtualization research on a global scale. And especially the material on Intel VT is rare. This paper focuses on the research of this field and raised some new theories. It also covered some blank and provide theory evidence for further research about virtual machine technology. Simultaneously virtualization technology is the base of cloud technology and internet of the things technology, it make sure the system can share the hardware resource efficiently and have a wide space to be extent.

**Keyword:** QEMU KVM Intel-VT virtualization internet-of-the-things cloud  
-computing

# 目 录

第 1 章 绪 论 .....	1
1.1 研究背景 .....	1
1.2 设备虚拟化的意义 .....	1
1.3 虚拟机研究现状.....	2
1.4 本课题的主要研究内容与成果 .....	4
1.4.1 主要研究内容 .....	4
1.4.2 主要成果.....	4
1.4.3 主要创新点.....	4
1.5 论文组织结构.....	5
第 2 章 虚拟化技术研究 .....	7
2.1 虚拟化研究 .....	7
2.1.1 虚拟化 .....	7
2.1.2 虚拟化的分类 .....	7
2.1.3 系统虚拟化 .....	8
2.1.4 系统虚拟化与多任务的区别 .....	9
2.2 纯软件虚拟化.....	9
2.2.1 QEMU 虚拟机.....	10
2.2.2 TCG - 动态翻译 .....	10
2.2.3 TB 链 .....	11
2.3 硬件辅助虚拟化研究 .....	11
2.3.1 硬件虚拟化技术 .....	11
2.3.2 Intel VT 技术.....	11
2.3.4 KVM 虚拟机 .....	15
第 3 章 基于 VT-X 的设备管理器虚拟化实现 .....	18
3.1 虚拟设备管理器的总体设计 .....	18
3.2 PCI 硬件设备的创建和初始化.....	20
3.2.1 PCI 总线结构虚拟 .....	20
3.2.2 PCI-PCI 桥的构造.....	21
3.2.3 PCI 设备初始化 .....	22



<b>3.3 设备中断虚拟</b>	<b>23</b>
3.3.1 物理中断	23
3.3.2 模拟中断源	24
3.3.3 物理真实中断拦截	25
3.3.4 中断的注入	25
<b>3.4 IO 虚拟</b>	<b>28</b>
3.4.1 设备虚拟中 I/O 的注册	28
3.4.2 KVM 中 I/O 操作拦截	29
3.4.3 KVM I/O 读写操作处理	30
3.4.4 硬件管理器 I/O 功能虚拟	30
<b>3.5 KVM 管理系统设计</b>	<b>32</b>
<b>第 4 章 虚拟时钟丢失优化</b>	<b>34</b>
<b>4.1 Linux 基本时钟</b>	<b>34</b>
<b>4.2 系统时间的丢失</b>	<b>35</b>
4.2.1 系统时间计算	35
4.2.2 时钟中断的丢失	36
4.2.3 时钟中断的耗时	36
<b>4.3 时钟虚拟化处理和改进</b>	<b>37</b>
4.3.1 虚拟时钟准确性改进	37
4.3.2 虚拟时钟性能上的提高	39
<b>第 5 章 虚拟 PCI 设备 I/O 访问起始地址修改</b>	<b>40</b>
<b>5.1 PCI 结构</b>	<b>40</b>
<b>5.2 QEMU-KVM 中 I/O 访问起始地址的指定分配</b>	<b>41</b>
5.2.1 I/O 起始地址的主动分配和统一刷新	41
5.2.2 PCI 虚拟设备 I/O 地址和虚拟桥 I/O 地址关系	42
5.2.3 虚拟设备跳过 I/O 重置直接刷新	43
5.2.4 虚拟 PCI 配置读写函数的重写	43
<b>第六章 实验测试与结果分析</b>	<b>45</b>
<b>6.1 基于 VT-X 的 QEMU-KVM 的性能对比测试</b>	<b>45</b>
6.1.1 测试环境配置条件	45

6.1.2 纯软件虚拟化和基于 VT 技术的 KVM 虚拟化对比 .....	46
6.1.3 纯软件虚拟化和基于 VT 技术的 KVM 虚拟化对比 .....	47
<b>6.2 设备管理器虚拟功能测试.....</b>	<b>48</b>
<b>6.3 时钟准确度测试.....</b>	<b>50</b>
<b>6.4 指定 PCI 硬件设备 IO 地址测试 .....</b>	<b>51</b>
<b>结论与展望 .....</b>	<b>55</b>
<b>致 谢.....</b>	<b>56</b>
<b>参考文献.....</b>	<b>57</b>

# 第 1 章 绪 论

## 1.1 研究背景

虚拟化技术不是一个新技术，它经历了很多年的不断发展和完善，早在十九世纪 60 年代在 IBM 的大型机里面就有所应用。虚拟化技术把物理资源和系统应用进行了隔离，提高了资源的使用效率，增强了硬件兼容性。在虚拟化中，看重的是如何去降低成本，提高 IT 资源的使用效率，然而使用纯软件方式提高资源使用效率已经遇到了瓶颈，从而具有虚拟化特性的处理器诞生了，这种处理器可以通过硬件辅助虚拟化技术实现多系统之间的隔离和切换，从而大幅度提高虚拟化运行效率和虚拟处理器的处理性能。

目前虚拟机技术是操作系统和系统结构领域的重要研究方向，特别随着虚拟化技术的不断发展，虚拟机技术也迅猛发展，因此出现了一批新的虚拟机。在这些新的虚拟机中 KVM 作为一个后起之秀，继承了之前多种虚拟机的优点，以卓越的性能被广大用户所青睐，所以越来越多的虚拟解决方案开始采用它。KVM 虚拟机因为它优越的性能，已经被加入到 Linux 最新内核源代码中，供用户使用，它精简独立，易于学习，里面涉及的各种最新虚拟化技术，包括 Intel 和 AMD 的硬件辅助虚拟化技术，非常的齐全，而且在不断更新和改进，这对研究虚拟技术非常有意义。

另外，虽然虚拟机技术在不断发展，但是它却没有得到很好的普及，特别是在国内，研究它的机构非常少，造成国内外虚拟化技术研究脱轨的现象出现。而虚拟机的应用又非常广泛，从小型的桌面系统，到大型服务器无处不在，它为减少资源浪费起到了至关重要的作用，而且现在推行的云存储中，底层实现中重要的元素也是系统级别的各种虚拟技术，所以怎么将资源共享使用成为了目前计算机世界的一个关注重点，研究虚拟机技术也迫在眉睫，意义重大。

## 1.2 设备虚拟化的意义

虚拟机技术主要使用系统虚拟化将一套物理计算机系统虚拟成为一套或者多套虚拟机系统[3]。而且虚拟客户机都拥有自己的虚拟硬件资源，如 CPU、内存和设备等，来提供一个独立的虚拟机执行环境。而在一个虚拟机里面 CPU 和内存的虚拟化基本上是固定的，不断扩展的就是支持源源不断的新虚拟设备，然而实质上 CPU 也只是一个特殊的虚拟化设备而已。所以怎么样高效的虚拟化一个计算机设备就具有非常普遍的意义。因此本文着重在设备虚拟化中进行阐述和分析，通过实例来验证研究成果，并且在这个基础上提出了自己的方案，对设备虚拟化

过程进行了优化。

### 1.3 虚拟机研究现状

从过去这些年来看，虚拟化技术确实不是一项全新领域的技术，但是它没有停止过发展，在十九世纪七十年代，著名公司 IBM 就已经做出了 VM/370 虚拟计算系统。当时用户可以在任何一个配置有 VM/370 的环境里面来运行任一款操作系统。人们研究虚拟化技术的目的大多是充分利用珍贵的硬件资源，通过虚拟机这种中间软件可以使更多的人通过统一平台接口接触和使用同一个计算机系统。而随着计算机本身硬件性能的不断提高，软硬件虚拟化技术再次成为计算机领域的研究热点，之所以这样，原因主要有两个方面：第一，经过多年的不断发展，计算机系统在不断变强大的同时，系统内部的管理问题也变得越来越难处理，并且越来越复杂，特别是随着计算机 CPU 多核化时代的来临，冗余计算的加入，使得这一矛盾将变得更加尖锐；第二，今天的计算模式，已经从以计算机为中心向以用户为中心的服务计算过渡，人们更关心的是计算系统将给用户提供怎样的接口和服务，而用户的需求往往是复杂和多样化的。在这样的情况下，虚拟化能够分离计算机硬件结构和软件接口之间存在的高耦合依赖关系，使得硬件充分有效的进行组合和使用，因此它重新获得了学术和工业上的重点关注。

随着计算机硬件技术的进一步发展，挑战也不断的出现，特别在 PC 计算机领域和服务器集群领域被普遍使用的 x86 计算机体系架构中，存在着虚拟上的缺陷，所以 x86 体系架构里的虚拟技术必须使用特殊的软硬件方法来弥补这些问题。虽然早起的完全软件虚拟化方法也可以实现系统虚拟化，但是仍然会造成性能大幅下降以及一些兼容性的问题。为了解决体系结构存在的缺陷，Intel 和 AMD 这两家硬件公司都开发了一套硬件辅助虚拟化技术，Intel 公司的 Virtualization Technology (VT) 技术和 AMD 公司的 Secure Virtual Machine (SVM) 技术。而本文主要针对 Intel 公司研发的 VT-X 技术进行深入的研究。

因为虚拟化效率需要进一步提升，计算机在每个层次中都在不断加入对虚拟化技术的硬件支持，这样能够很大幅度的提高虚拟化效率，如 Intel 在处理器里面除了加入 VT 技术外，芯片组里面还提供针对硬件 I/O 方面的 VT-D 虚拟化技术，这样大大提高了设备 I/O 读写操作的执行速度，而在一些特殊的如网卡设备中也开始提供特殊的网络虚拟化支持如 VMDQ 技术等。PCI 组织也正在制定 PCI 硬件的单根 PCI 桥 IOV (Single Root IOV, SR-IOV) 和多根 PCI 桥 IOV (Multi-Root IOV, MR\_IOV) 标准。

上面是最近虚拟技术方面的一些发展情况，下面介绍近年来出现的一些虚拟机软件：

BOCHS

BOCHS 是一个仿真 X86 系统架构的设备模拟器，能够模拟指定的型号的 CPU，可以在多种系统中运行使用，如 PowerPC, x86, SPARC, Alpha, Bochs, 它的优点是调试起来特别方便，有一个专门进行调试的版本，运行一个虚拟机镜像以后可以通过单步调试操作系统的每个地址，即可以指定物理地址，也可以指定线性地址，可以查看 CPU 的各个寄存器，也非常方便。

#### Wine

这种虚拟机叫做库级别虚拟机，因为它通过库来模拟操作系统里面的一些功能。

#### QEMU

QEMU 也是一种模拟器，和 BOCHS 很像，只是性能上比 BOCHS 更加好，支持的硬件也更加全面，不过没有 BOCHS 调试功能强大。它支持两种执行模式，其中一种模式需要附属在 Linux 内核上面运行，这种叫做用户模拟模式，在这种模式里可以运行不同平台的二进制文件；另外一种模式能够对整个计算机系统进行模拟，这种叫做全系统模式，它可以高效快速的动态翻译指令，并且模拟多种处理器架构，在下文中会进一步介绍动态翻译方面的一些知识。

#### VMWare

VMWare 是大家都非常了解的投入运营的完全虚拟化产品代表之一，它被 EMC 收购现在在国内也有研发，它以 Hypervisor 作为客户操作系统和硬件之间的纽带抽象出来，并且允许其它客户操作系统完全的运行在该主机操作系统上。

#### Xen

虚拟机 Xen 由开发小组 XenSource 进行维护，它也是当前一个相对成熟完整的商用虚拟机，它使用开源免费的操作系统级准虚拟技术，也就是说它是一个半虚拟化产品。Xen 需要与系统共同协作，需要加入特定的前端驱动到客户操作系统形成多级驱动，所以需要不断的修改客户机内核代码来进行改进和更新，因此这种虚拟化技术虽然快，但是需要针对特定产品做特定的修改，需要用户付出一定的开发成本。

#### KVM

KVM 作为本课题的主要研究对象，也是因为它既一个新诞生的虚拟机，也是一个优秀的虚拟机。它通过移植改进 Xen 里面的现有代码，并且充分利用硬件辅助虚拟化技术比如 Intel VT 技术，并结合 QEMU 来提供设备虚拟化。KVM 性能高效，使用起来也非常简单，它加快了 QEMU 的硬件模拟速度，另外 KVM 和 Linux 内核结合得很好，它使用了 Linux 的很多已有功能，因此可以很好的利用 Linux 内核不断改进和优化的技术优势来不断的提高性能。到 Linux 2.6 内核以后它已经作为模块加入到正式的内核之中。

## 1.4 本课题的主要研究内容与成果

### 1.4.1 主要研究内容

通过对当前国内外虚拟机的研究，本课题基于 Intel VT-x 虚拟化技术，深刻剖析了 QEMU-KVM 虚拟机设备虚拟化机制原理和过程以及硬件辅助虚拟化的用法和缺陷，并且通过一个特殊的设备管理器的虚拟化对其进行阐述，而且通过自己的理解提出一套设备虚拟化解决方案。本课题着重从虚拟设备的 I/O 模拟和中断虚拟两个方面来进行研究，最终还对时钟设备和特殊 PCI 硬件设备 I/O 访问起始地址模拟进行改进和优化。

本论文的主要研究内容有：

- (1) KVM 对设备 I/O 拦截和处理的研究。
- (2) QEMU-KVM 中断虚拟化和 VT-X 虚拟化技术的研究
- (3) 时钟设备虚拟化性能和准度的研究。
- (4) PCI 硬件设备虚拟 I/O 指定分配基地址的改进。
- (5) 模拟设备 I/O 处理从 QEMU 移植到 KVM，硬件管理器设备虚拟化实现。

### 1.4.2 主要成果

通过对 QEMU-KVM 虚拟机的研究，本文通过实验得到了预期的目标，并且完成了以下几点研究成果：

#### 1. 实现了一个硬件管理器的虚拟化。

本文针对 Intel VT 硬件辅助虚拟化技术，将 QEMU 中的硬件模拟部分转移到 KVM 中实现，完成了一个硬件管理器的虚拟化实例，同时提出了一个完整的设备虚拟化应用解决方案。

#### 2. 完成了系统时钟和基于 VT-x 的虚拟中断研究，提出了时钟设备改进方法。

提出了一个在虚拟化过程中减少虚拟时钟丢失的改进方法。通过将虚拟时钟丢失中断进行保存和再触发以及虚拟中断批量注入来解决时钟丢失问题。

#### 3. 完成了对 QEMU-KVM 中设备 I/O 虚拟化的研究。

本文一方面对 QEMU 中的虚拟设备模拟进行研究，同时另一方面对 KVM 中的设备注册和 I/O 拦截进行研究，通过两个方面的结合，实现了将设备虚拟化 I/O 部分从 QEMU 中转移到 KVM。

### 1.4.3 主要创新点

本文相比以往的虚拟化文献，在研究上主要有以下两点创新点。



1. 提出了一个时钟设备的虚拟化改进方案。

本文通过对中断和时钟虚拟化的研究，借助 Intel VT 技术通过保存-再触发方式解决时钟中断的注入丢失问题，并且通过批量注入方式来提高实现时钟中断的注入效率，减小系统损耗。

2. 提出了一个固定虚拟 PCI 设备 I/O 基地址的改进方法。

本文通过对虚拟 PCI 设备地址初始化进行拦截和修改，实现给指定的设备固定分配 I/O 基地址。

## 1.5 论文组织结构

本文组织结构安排如图 1-1 所示。

第 1 章 介绍了有关虚拟机技术的一些背景，发展现状，以及当前主要的虚拟机。通过对这些分析体现出本文研究的意义，另外本章节还列出了本文的研究的主要内容、主要成果和创新点。

第 2 章 对虚拟机技术以及一些基本概念进行了一个系统性的介绍，其中包括传统的纯软件虚拟化，以及加入了硬件辅助虚拟化技术的 VT-X 的虚拟化方法，还包括一些虚拟化的分类，完全虚拟化的特点等等，为后文提出解决方案提供理论依据。

第 3 章 通过对传统虚拟化和硬件辅助虚拟化技术进行分析和对比，在虚拟机 QEMU-KVM 的基础上提出一套针对 Intel VT 技术的硬件辅助虚拟化设备虚拟解决方案，并且实现对一个硬件设备管理器的虚拟化。

第 4 章 针对系统时钟进行改进，分析时钟出现丢失原因，加入丢失管理解决方案，提高时钟精准度。

第 5 章 通过对 PCI 硬件设备模拟进行改进，指定 PCI 硬件设备以及 PCI 桥的 IO 访问起始地址的分配。

第 6 章 通过实验结果证明本文提出的虚拟化解方案行之有效，并且处理实验结果，对比设备虚拟化性能数据，分析数据，证明改进效果。

最后一章是总结与展望。本章对本文实现的系统与进行的研究工作做一个总结，分析设计中的不足之处，并展望设备虚拟化的发展前景。

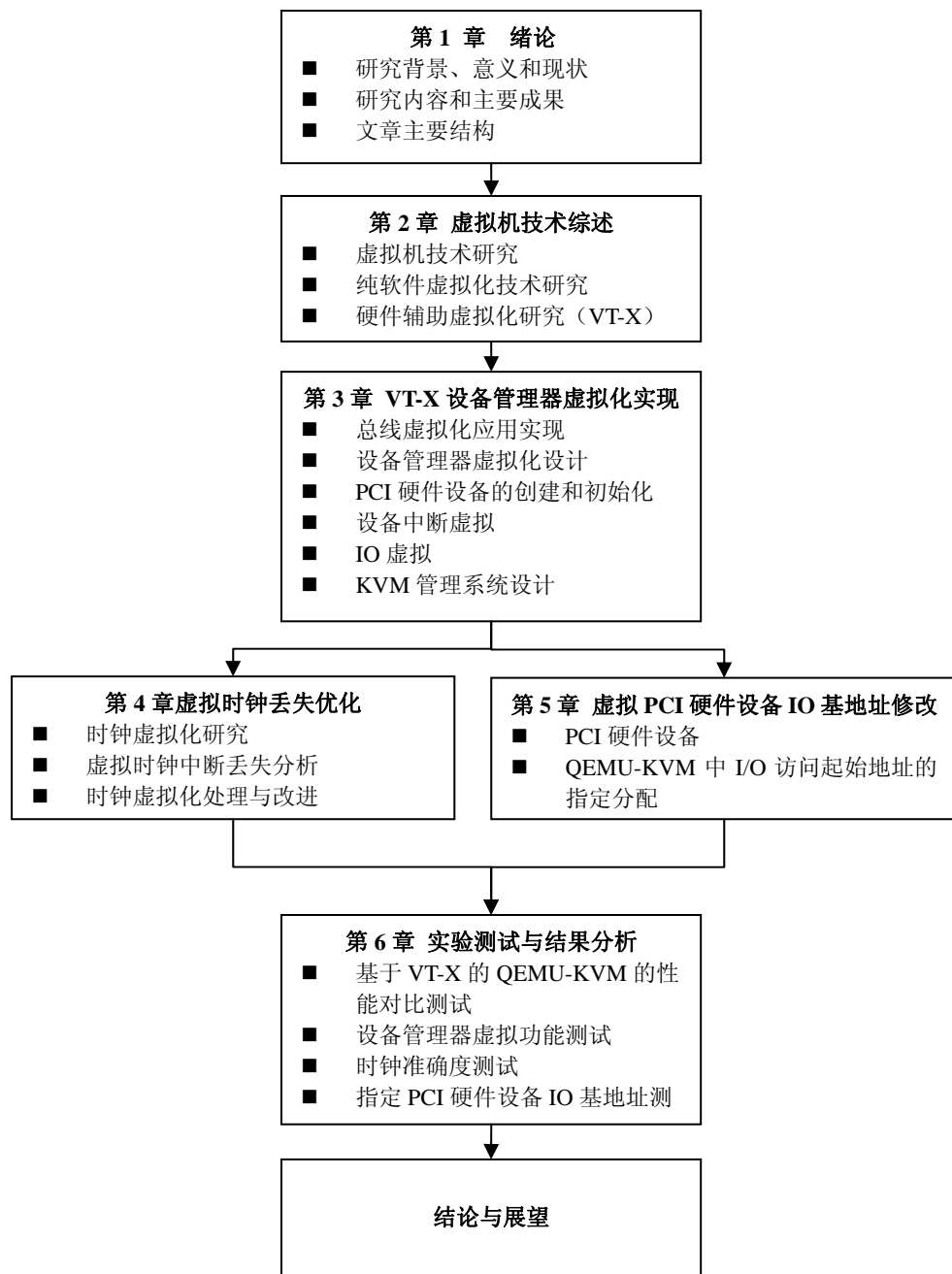


图 1-1 全文组织结构图

## 第 2 章 虚拟化技术研究

### 2.1 虚拟化研究

#### 2.1.1 虚拟化

虚拟化这个词语非常广泛，是指软件在非真实的环境上运行，而不是在本身存在的环境下运行。虚拟化技术可以扩大硬件的容量，增强软件兼容性。如图 2-1 所示。

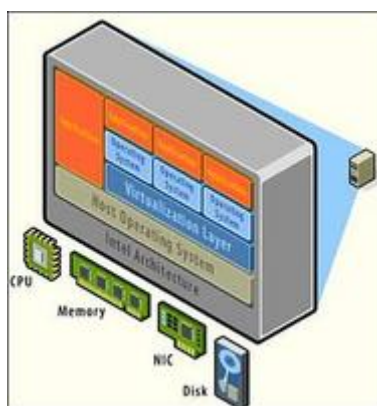


图 2-1 虚拟化技术图

#### 2.1.2 虚拟化的分类

现代计算机系统非常庞大，因此整个系统也非常复杂。所以计算机系统分成了很多层。其中每一层只需要知道下层的接口，而不需要知道它的内部实现机制，如果我们操作系统在使用一个硬件的时候只需要读写硬件接口而不需要了解硬件电路组成。而我们应用程序也只要直接调用系统提供的系统函数，而不需要去分析函数怎么实现，这在一定程度是一种面向对象思想的体现。如图 2-2 所示。

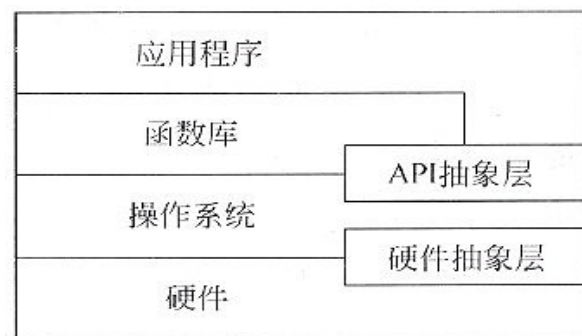


图 2-2 计算机系统分层结构图

从根本上来说，虚拟化就是一个中间层软件，不断满足上层软件需求的接口的技术方法。虚拟化可以发生在上述的各个层次之间，因此就引出了不同的虚拟化方法和分类。

在分类之前先了解两个重要概念。在虚拟机中，直接运行在物理硬件资源上的操作系统有一个词语叫做宿主机系统（Host OS），而通过虚拟出来的系统叫做客户机系统（Guest OS）。

### 1. 硬件设备层虚拟[3]

这种虚拟化技术是针对硬件层来实现虚拟功能的，通过虚拟以后客户机系统和真实物理硬件呈现相似或者一样的硬件特性，而在这两者之间就出现了一个硬件抽象层。因为客户系统能够看到的是硬件层，所以，客户操作系统的行为与物理平台上没有区别。通常客户机和宿主机的指令架构相同，客户机的指令基本上可以在宿主机上直接运行。只有需要虚拟化的指令才交给虚拟化软件处理，因此虚拟化软件做的工作不大。而在这种情况下，真实硬件设备如中断控制器等可以和客户机需求的虚拟环境完全不同，只需要虚拟化软件对这些请求进行拦截和模拟。这种虚拟化代表的产品有 VMware 和 Xen 等虚拟机。

### 2. 操作系统层虚拟[3]

这种虚拟化是指主机操作系统本身能够提供几个相互之间隔离的用户态进程。这些进程实例就分别模拟一台真实的计算机，每个计算机有自己的文件系统，网络系统以及库函数等。这是操作系统内核主动提供的虚拟化，因此非常高效，性能开销小，也不需要特殊硬件支持。但是它的致命缺点就是灵活性相对较小。它主要的代表有 Parallels 的 Virtuozzo。

### 3. 库函数层虚拟[3]

这种虚拟化主要是操作系统通过应用级的库函数的模拟来实现系统之间的需求转换。因为不同操作系统之间的服务接口是不同的，比如 Linux 和 Windows，那么在虚拟化的时候只需要转化模拟这些不同的服务接口，就能提供不同操作系统的服务需求，达到虚拟的目的。

### 4. 语言中间层虚拟[3]

这种虚拟化最上层，是给上层编程使用的，通过提供一个中间转化平台来解析用户编程语言。如 Java 的 JVM，Java 语言会先被虚拟机翻译成硬件支持的机器语言，然后进行执行，这个属于进程级的虚拟化。

## 2.1.3 系统虚拟化

系统虚拟化是虚拟化中非常重要的一种，也是本课题的研究重点，它指将一套计算机物理硬件资源系统的转化为一台或多台虚拟机系统。每个虚拟机系统拥

有自己独立虚拟硬件，来提供一个相对独立的虚拟客户机环境。不同虚拟客户机之间相对独立，这个介于真实硬件和虚拟客户机之间的部分叫做虚拟机监控器（VMM），如图 2-3 所示。

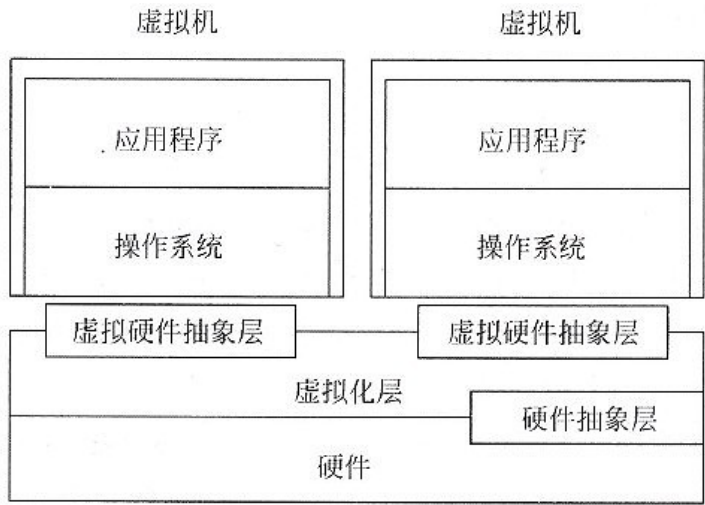


图 2-3 系统虚拟化结构图

### 2.1.4 系统虚拟化与多任务的区别

超线程技术和系统虚拟化技术属于两个完全不同的概念，而且实现技术完全不同，只是实现技术的思维上有点相近。多任务这个概念大家应该非常熟悉，它是在一个操作系统内部进行的，通过多个程序多个进程并发分时运行来实现的。而虚拟化技术在根本上也是通过分时共享硬件资源来实现，但是不是在进程级别来分时运行应用程序，而是分时来运行多个操作系统，这样对于任何一个操作系统里面都运行多个进程，虚拟操作系统与其对应都会分配一个虚拟 CPU 来支持运行。

## 2.2 纯软件虚拟化

纯软件虚拟化是指在不借助硬件虚拟化支持的情况下，完全使用软件方法来实现系统虚拟化的解决方案。它分为两种方法一种是模拟执行，另一种是直接改写底层二进制代码。其中模拟执行的方法就是我们所说的完全虚拟化技术，直接源代码改写则是对应的半虚拟化技术，也叫类虚拟化技术。

纯软件虚拟化包括 CPU 虚拟化，内存虚拟化以及 I/O 虚拟化。对于 CPU 虚拟化，虚拟机监控器会监控客户机运行的 CPU 指令，然后使用陷入再模拟的方式来翻译模拟执行这条指令；对于内存虚拟化虚拟机监控器使用多级页表，多层定位来区别不同客户机的内存区域，使得不同客户机的内存区域隔离开。而 I/O 虚拟

化是设备虚拟化的核心，在没有硬件辅助虚拟化的时候，所有 I/O 相关操作都要用纯软件的方法进行模拟。

纯软件虚拟化的方法来进行系统虚拟很有多方面的缺陷和限制。这种方式下运行的客户机系统大部分是通过虚拟机管理系统(简称 VMM)来与真实的硬件设备进行通讯。在纯软件方式进行虚拟化的情况下，VMM 就相当于传统计算机系统里面操作系统所处的位置，因为它要满足上层系统的应用需求，而此时客户机操作系统就相当于平时所说的应用程序，需要 VMM 提供支持才能运行。在这个中间层上进行一定的转化操作，能够翻译不同架构的 CPU 指令，并且调用各个真实物理资源的驱动程序来模拟出虚拟的硬件条件。这层转换大大的增加了整个系统的复杂性。另外，这种方式的模拟在环境兼容性上受到很大的限制，而且不容易适应新技术的发展，比如最近的操作系统，64 位客户机系统，等等。在纯软件的虚拟化方法里，软件实现的复杂性就意味着它将难于管理，进一步会让系统可靠性和安全性受到很大的威胁。

## 2.2.1 QEMU 虚拟机

如我们所知，QEMU 是一个模拟器，它能够动态模拟特定架构的 CPU 指令，如 X86，PPC，ARM 等等。QEMU 模拟的架构叫目标架构，运行 QEMU 的系统架构叫主机架构，QEMU 中有一个模块叫做微型代码生成器 (TCG)，它用来将目标代码翻译成主机代码。

我们也可以将运行在虚拟 CPU 上的代码叫做客户机代码，QEMU 的主要功能就是不断提取客户机代码并且转化成主机指定架构的代码。整个翻译的过程可以划分为两块：第一个部分是将做目标代码 (TB) 转化成 TCG 中间代码，然后再将中间代码转化成主机代码。

## 2.2.2 TCG - 动态翻译

QEMU 在 0.9.1 版本之前使用 DynGen 翻译 c 代码。当我们需要的时候 TCG 会动态的转变代码，这个想法的目的是用更多的时间去执行我们生成的代码。当新的代码从 TB 中生成以后，将会被保存到一个 cache 中，因为很多相同的 TB 会被反复的进行操作，所以这样类似于内存的 cache，能够提高使用效率。而 cache 的刷新使用 LRU 算法。

编译器在执行器会从源代码中产生目标代码，像 GCC 这种编译器，它为了产生像函数调用目标代码会产生一些特殊的汇编目标代码，他们能让编译器晓得在调用哪些函数。需要什么，以及函数调用以后需要返回什么，这些特殊的汇编代码产生过程就叫做函数的 Prologue 和 Epilogue，这里就叫前端和后端吧。至于



汇编里面函数调用过程中寄存器是如何变化的，在本文中就不再描述了。

函数的后端会恢复前端的状态，主要做下面 2 点：

1. 恢复堆栈的指针，包括栈顶和基地址。
2. 修改 cs 和 ip，程序回到之前的前端记录点。

TCG 就如编译器一样可以产生目标代码，代码会保存在缓冲区中，当进入前端和后端的时候就会将 TCG 生成的缓冲代码插入到目标代码中。

### 2.2.3 TB 链

在 QEMU 中，从代码 cache 到静态代码再回到代码 cache，这个过程比较耗时，所以在 QEMU 中涉及了一个 TB 链将所有 TB 连在一起，可以让一个 TB 执行完以后直接跳到下一个 TB，而不用每次都返回到静态代码部分。

## 2.3 硬件辅助虚拟化研究

### 2.3.1 硬件虚拟化技术

硬件辅助虚拟化技术是一项比较新的虚拟化技术，从名字可以看出，就是在硬件上加入了一些特殊的机制，能够对软件虚拟化有系统性的支持，使得系统软件里面能够非常容易非常高效的，比如在处理器、主板芯片组及设备 I/O 里面都加入专门针对虚拟化的特殊支持。

前面已经说了，纯软件虚拟化性能低下，而且存在很多问题，比如 X86 优先级切换问题，“影子页表”问题等等。因此 Intel 公司和 AMD 公司分别推出了支持虚拟化的 CPU 芯片，这样一来硬件辅助虚拟化开始飞速发展。本课题主要针对 Intel 的 VT 技术，因此这里主要介绍 Intel 推出的相关技术。

### 2.3.2 Intel VT 技术

讲到硬件辅助虚拟化技术，首先介绍一下处理器虚拟化，处理器虚拟化的本质是分时共享。实现虚拟化需要两个必要条件，第一是能够读取和恢复处理器的当前状态，第二是有某种机制防止虚拟机对系统全局状态进行修改。

第一个必要条件没有必要一定由硬件来实现，虽然硬件实现可能比软件实现更为简单。例如，x86 的处理器，对多任务系统接口提供了硬件上的支持，软件通常只需要执行一条指令，就可以实现任务切换，处理器硬件负责保存当前应用编程接口的状态，并为目标任务恢复应用编程接口的状态。但操作系统并不一定

要使用处理器提供的这种虚拟化机制，完全可以使用软件来完成应用接口状态的切换。例如，Linux 就没有使用 x86 的处理器提供多任务机制，完全依赖软件实现任务切换。

第二个必要条件一定要由硬件来实现，通常处理器采用多模式操作（multi-mode operation）来确保这一点。在传统 x86 处理器上，共有 4 种模式的操作，也就是常说的 4 个特权级。虚拟机（这里指进程/线程）通常运行在特权级 3 上，而虚拟机监控器（这里指操作系统）运行于特权级 0 上，进程/线程的所有访问全局的操作，如访问共享的操作系统所在的地址空间，访问 I/O 等等，均会导致异常的发生，被操作系统所截获并处理，使操作系统有机会向进程/线程提供一个虚拟的世界。

Intel VT（Intel Virtualization Technology）包括 VT-X、VT-D 等技术，里面主要涉及 CPU 以及其他硬件设备的辅助虚拟化技术。其中最关键的就是 CPU 虚拟化技术（VT-X）。支持硬件虚拟化的 CPU 具有特别的一套虚拟指令集来优化虚拟过程中的各个操作，这样一来，VMM 的性能将会有很大程度的提高，经过提高以后的虚拟机性能和纯软件实现的虚拟机性能将发生数量级的变化，这个在最后一章里面将进行测试。硬件虚拟化技术还可以提供基于硬件芯片的功能，再加上对硬件虚拟化的软件支持部分，合起来就能够替代传统的纯软件虚拟解决方案。由于现代计算机系统中各种硬件提供了全新的系统架构，使得客户机操作系统能够在很大程度上直接在真实物理硬件上面进行指令执行，这样一来就不需要进行二进制转换，同时就大大减小了系统开销，而且不用像以前一样复杂的设计 VMM 系统。因此只需要设计一些硬件接口部分，因为硬件 CPU 的接口却相对比较统一，这样在很大程度上更加规范了现代 VMM 的设计与实现，趋于一种规范化。另外，在纯软件实现的 VMM 中，目前还没有考虑支持 64 位的处理器芯片，而现在就成为了一个非常迫切的问题，就因为 64 位处理器在大规模的普及，在这点上硬件辅助虚拟就有很大的优势，因为硬件的不断更新换代就增加了新的支持，使得软件实现部分可以保持不变。硬件虚拟化是一套系统的解决方案，其中包括 CPU 虚拟、主板芯片组虚拟、BIOS 虚拟和软件 VMM 设计的支持。但是各种虚拟化都在性能上有所提高，即使只使用 CPU 辅助技术，也在性能上有很大提高，本文主要就针对这项技术来实现设计。

VT-X 虚拟技术扩展了传统的 32 位处理器架构，为这种处理器的虚拟化实现增加了硬件上的支持。它的思路主要如图 2-4 所示。

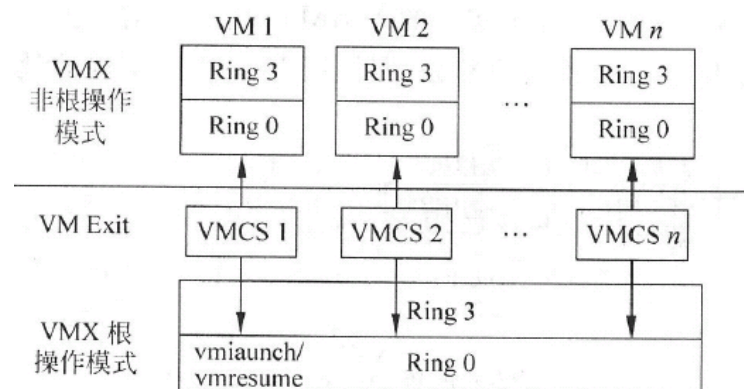


图 2-4 VT-X 技术架构图

首先，VT-X 技术引入了两种新的操作模式，统称为 VMX 操作模式，分别是根操作模式（简称根模式）和非根操作模式（简称非根模式）。这两种操作模式与 IA32 传统特权级 0-3 级呈正交。换句话说，也就是每个操作模式都对应具有特权级中 0-3 个运行级别。这样一来，在这种新的环境里面程序不仅要看是不是属于某个特权级，而且还需要知道当前是处于非根模式还是根模式状态。

引入这两种不同的操作模式是为了解决“陷入再模拟”虚拟模式漏洞的问题。在 IA32 系统架构下有十九条敏感指令不能沟通过简单的方法进行模拟，它需要段优先级的提升，所以引入两个模式能够通过改变模式来解决不同优先级段之间切换和保护的作用。在 VT-X 技术里，由于敏感指令引起的系统从非根模式下进入根模式的陷入过程就叫做 VM-Exit。当这种 VM-Exit 生效时，CPU 将进行模式之间的切换。相反的，VT-X 技术里也定义了对应的 VM-Entry，该指令是一个与进入相反的过程，是从 VMM 切换到某个具体客户机内部的过程。

另外，为了更好的支持 CPU 虚拟化，VT-X 技术引入了 VMCS (Virtual-Machine Control Structure，虚拟机控制结构)。VMCS 里面保存虚拟 CPU 切换需要保存的各种相关状态，每一个 VMCS 具体实例对应着一个虚拟 CPU 简称 VCPU，例如 CPU 在运行中需要用到的各种特权寄存器的值。VMCS 主要给 CPU 使用，物理 CPU 在相应系统里面发生进入和退出操作的时候就会改变读取或者写入 VMCS 的数据或者控制区域。而 VMM 有对应的特殊方法来读写 VMCS，进而改变 CPU 的行为。

最后 VT-X 还引入了一组新的指令，包括 VMLAUNCH/VMRESUME 和 VMREAD/VMWRITE，用来发起 VM-Entry 和 Exit 以及配置 VMCS。如图 2-5 所示。

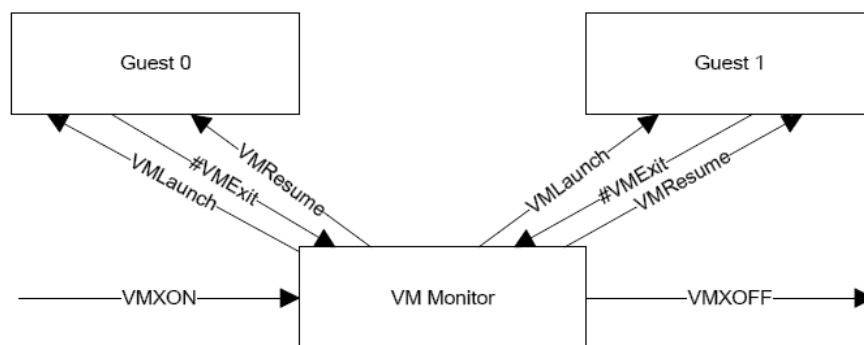


图 2-5 VMCS 指令关系图

虽然有 VT-X 技术的支持，但是目前的 I/O 设备虚拟化很容易成为虚拟性能的瓶颈，因此 Intel 就适时提出了 Intel Virtualization Technology for Directed I/O，简称为 Intel VT-d。虚拟机中的虚拟化技术从我们传统的纯软件方式逐渐发展成处理器级别的虚拟化，接着再发展成平台级别的虚拟化甚至最后包括输入/输出级别的虚拟化，这种针对 I/O 输入输出级别的虚拟化技术的代表性技术就是 Intel 的 VT-d。它解决的主要问题就是设备在 I/O 虚拟中出现的性能低下问题，比如 DMA 内存直接读取，中断 IRQ 请求等频繁操作造成的性能下降，因此它的关键点就在于解决真实 I/O 设备与虚拟客户机中数据交换的问题，只要系统能够解决好这方面的问题，并且做好分离和安全保护工作，就是成功的 I/O 虚拟化。

与处理器上的虚拟化技术类似，虚拟机 VT-d 技术主要是基于北桥芯片（或者说 MCH）的虚拟化辅助技术，因此需要支持 VT-d 就不仅是 CPU 的问题，需要主板和设备同时支持才行。在这些技术中通过北桥里提供 DMA 虚拟化功能和系统 IRQ 虚拟硬件支持，整体的打造了一套新型的 I/O 方式的虚拟化，这样，整个系统的 I/O 读写方面的性能就得到了非常大的提高，就像之前的 CPU 虚拟化一样，稳定性和兼容性也有一定程度的提高。

IOMMUs（I/O memory management units，I/O 内存管理单元）如大家所知是专门用来管理 I/O 内存的设备，它通过统一的方式来对 DMA 进行管理——不仅包括以前的一般 DMA，而且还加上 AGP TPT、GART、RDMA over TCP/IP 等这些特别的 DMA 设备，不同的设备 I/O 映射到不同的内存地址，通过这个地址来区分各个设备，这样做很容易实现。然而有个很大的问题就是隔离问题，所以 VT-d 重新设计了 IOMMU 架构，是的 DMA 能够很好的并行，相互之间的数据区域也能很好的保护，这样使得虚拟化也非常完美。通常也称作 DMA 的重映射。

说到 VT-d 技术，不得不讲述其中的中断，因为对于一个设备而言，中断是非常重要的组成部分，每个一个设备都可以通过两种方式来读取数据，要不就是轮训，要不就是中断，然而中断是最常见的方式，因为节省系统资源，然而在虚拟化中，中断性能却成了 I/O 性能的一种瓶颈，因为中断 I/O 设备需要许多的中

断请求，I/O 设备的虚拟就需要将每个设备的中断通过虚拟中断控制器分发到对应的虚拟设备上，这样才能实现中断虚拟化。传统硬件设备如果需要使用中断，可以使用两种方法：其中一种就是前面所说的虚拟中断控制器路由转发，还有一种就是通过 DMA 直接转发 MSI 消息，这里的 MSI 就是平时说的中断信号消息的简称。因为中断需要隔离，所以直接虚拟 MSI 的方式非常困难，一般不容易实现，所以大多使用中断分发的方式实现。

VT-d 技术主要有两种虚拟化方式来实现的，具体如下：

直接进行设备 I/O 分配：在这种情况下，虚拟机直接进行 I/O 地址分配，所以虚拟机里的程序可以访问真实的物理设备的 I/O 接口，因此速度快很多，在中间访问的过程中，经过 VMM 处理的部分非常少，然而对主板和真实硬件的虚拟化要求就非常高，需要自己已经实现部分虚拟化功能。

I/O 共享设备：在这种情况下，对硬件的需求就更高，因为需要设备已经支持多个虚拟接口，这样可以将每个接口直接分配给虚拟机内部的驱动程序，但是这种情况下虚拟设备的个数取决于真实物理硬件支持的个数，而且需要配置以后才能使用。

总之运用 VT-d，可以很大的提高 I/O 的访问速度，提高硬件虚拟化性能，然而这种情况对硬件要求非常高。

### 2.3.4 KVM 虚拟机

KVM 虚拟机 (Kernel-based Virtual Machine) 是一个年轻的虚拟机，从开发以来到现在一直以它优异的性能受到广大用户的喜爱。KVM 顾名思义，它是一个依靠 Linux 内核来运行的虚拟机，现在已经作为 Linux 内核的一部分加入到最新的内核之中，所以 Linux 用户使用 KVM 非常便捷。另外 KVM 是一个开源的虚拟机，由一个开源小组来进行维护，现在红帽子可开始维护 KVM 虚拟机，并且把 KVM 作为它 Linux 中的一个重要部分。最后再从技术角度来说，KVM 是从高性能虚拟机 Xen 改编过来的完全虚拟化虚拟机，继承很多虚拟机的优点，并且发展迅速，它本身的代码量很小，一方面依赖 QEMU 模拟器来进行设备模拟，另外一个方面加入大量硬件辅助虚拟化技术的支持，因此，无论从稳定性来说，还是从性能上来说，都是非常优秀的。特别是加入硬件辅助虚拟化技术使得它比很多传统虚拟机的性能都好，再加上和 Linux 的内核的紧密联系，随着 Linux 内核的不断优化，KVM 虚拟机也跟着提高性能，不断的被改进。

KVM 加上 QEMU 合起来就叫做 QEMU-KVM 这才算是完整的虚拟机，结合 Linux 内核就构成了前文所说的 VMM，因此 KVM 由两部分组成，一个作为 Linux 内核模块和内核一同运行在内核态，另外一部分就是 QEMU 部分运行在用户态和用户进行交互。在这里所说的内核态和用户态就是指 X86 体系结构里面 0-3 不同的特权

级，而我们的内核态的 KVM 又有两种运行状态一种是前面所说的根模式，在这里也叫 Kernel 模式，还有一种非根模式，在这里叫 Guest 模式，具体如图 2-6 所示。

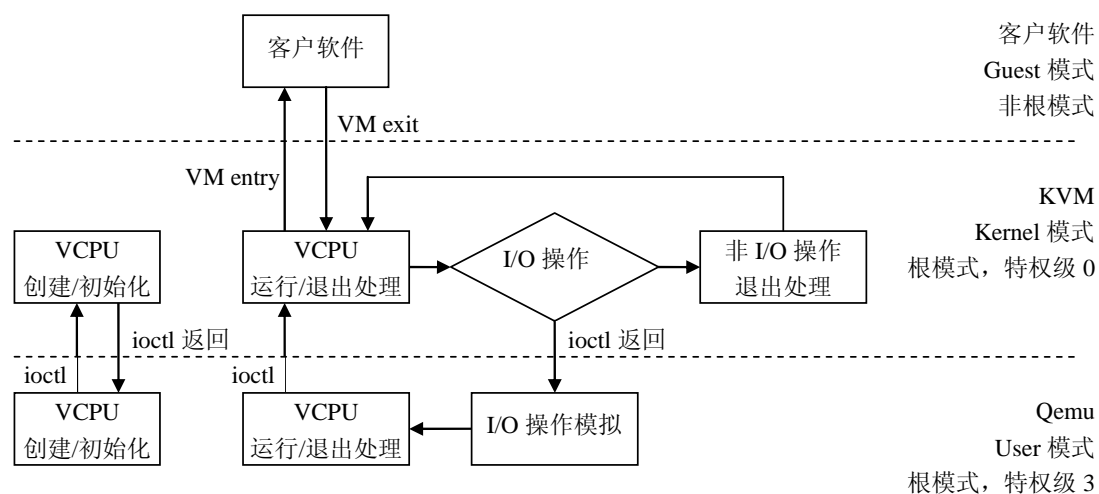


图 2-6 VCPU 工作模型

KVM 主要使用 VT-x 技术来进行 CPU 虚拟化，这个是通过 QEMU 和 KVM 的配合实现的，在虚拟机中，对于每个虚拟处理器都有一个对应的 VCPU 结构体，而每一个 VCPU 在 QEMU 里面会对应一个处理线程，进行循环处理 CPU 事件，最终分时到真实 CPU 上。在 KVM 内核内部，负责创建 VCPU、初始化 VCPU、运行 VCPU 以及各种退出处理等，这条线程贯穿 QEMU 和 KVM，而且需要 Kernel、User 和 Guest 三种模式配合进行，主要如图 2-6 所示。QEMU 的用户态与 KVM 的内核态主要通过 IOCTL 方式进行通讯，然而 KVM 的根模式和非根模式，主要通过 VM Exit 和 VM entry 进行切换。

上文说的 QEMU 主线程通过 IOCTL 然后进入 KVM 内核态，对各种 VCPU 状态进行操作，通过共享内存的方式传递 VCPU 的各种信息，而这些信息在 KVM 中也通过保存在 VMCS 中，它是一个关键性的数据结构。

QEMU 在启动以后会优先初始化虚拟 CPU 和各种硬件设备，然后通过 IOCTL 进入 KVM 内核模块，此时，VCPU 的信息会写入到 VMCS，紧接着当前 VCPU 得到真正的调度，KVM 虚拟机触发 VM entry 操作，这样，虚拟 CPU 就完成了一次调度，这个时候从图中可以看到 KVM 虚拟机从 Kernel 模式切换到了 Guest 模式，来运行客户机里面的应用软件。当这个调度受到其他因素，如敏感指令或者外部中断等影响时，虚拟机将会触发 VM exit 操作，这样，虚拟机会退回到 Kernel 模式，完成需要的操作。然后选择是否进入 QEMU，是否处理 IO，是否处理中断，等等。对于 I/O 处理来说，这个时候一旦退出，就会触发相应的总线读写事件，进而转到真实的物理设备，最终完成 I/O 操作，然而再等待下一次调度。



## 本章小结

本章对课题研究中涉及的最基本概念与技术进行了分析、研究与总结，首先对一般虚拟化技术进行了分析和分类，并且把虚拟化和多线程进行了一个对比。然后着重分析了本课题主要研究的系统虚拟化，接着对传统纯软件虚拟化技术进行分析讨论，最后在前面的基础上提出了硬件辅助虚拟技术并且对其设计到的主要相关技术进行研究，并且通过这个技术引出了本课题要用到的 QEMU-KVM 虚拟机。

# 第 3 章 基于 VT-X 的设备管理器虚拟化实现

系统虚拟化中一个完整设备的虚拟化过程需要模拟它的各个方面，其中包括设备的安装启动，设备的中断虚拟化以及 IO 读写处理。设备的安装一般在 QEMU 中完成，传统纯软件虚拟化的设备处理都在 QEMU 中，然而加入 VT-X 技术以后，整个虚拟机多了一个 KVM 模块，所以可以将部分设置工作放在 KVM 中进行。

在设备的虚拟化里面，非常关键的两个部分就是前文所说的设备的中断和 I/O 读写。在硬件辅助虚拟化技术里面他们涉及到 VT-X 和 VT-D 技术，然而 VT-D 技术和设备具体硬件支持有关，不具有普遍性，所以这里主要针对 VT-X 进行应用研究。

本章一方面实现了一个设备管理器的虚拟化，另一方面通过实例总结了一个基于 VT-X 技术的设备虚拟化解决方案。

## 3.1 虚拟设备管理器的总体设计

设备管理器是一种特殊的硬件设备，用于服务器群组中机器之间的简单消息通讯以及统一管理，本章节针对这种通讯和管理机制设计了一套虚拟化方案。

一般设备管理器硬件结构分布如图 3-1 所示：

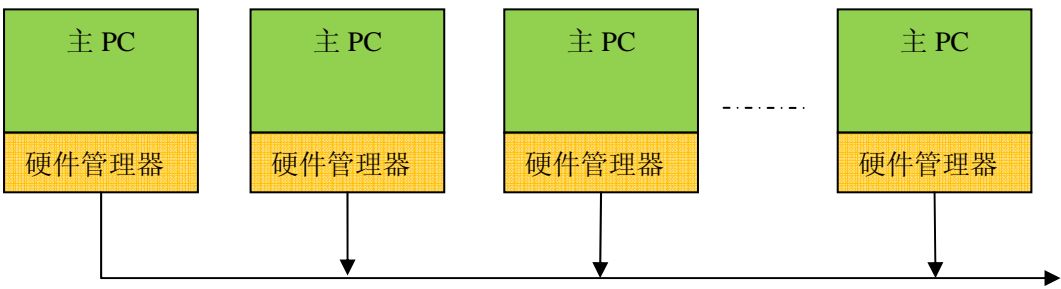


图 3-1 设备管理器分布图

在上图中，各个 PC 机通过硬件管理设备相互之间连成一个统一的网络，通过特殊的通讯模式相互之间实现通讯，而每个硬件管理器作为一个特殊的设备，负责对通讯数据的读和写，已经对 PC 机板子进行控制，比如开关机等。

在这种情况下如果我们需要对每台 PC 机中的硬件实现虚拟化，那么必将实现硬件管理器的虚拟化，而硬件管理器的虚拟化就需要对所有硬件管理器功能进行虚拟化，包括开关机功能，所以通过总结得到我们需要解决一系列问题如下所

示：

- 1) 硬件管理器 IO 读写的分发
- 2) 硬件管理器中断的分离
- 3) 硬件管理器对操作系统开关机操作
- 4) 硬件管理器驱动与虚拟机通讯问题
- 5) 硬件管理器实时性问题

为了解决这些问题，本课题设计了一个适用于此种情况的解决方案。此方案的整体架构图如图 3-2 所示。在图中为了实现 IO 和中断的分发和隔离，需要设计一套拦截系统，拦截真实和虚拟驱动之间的数据，并且实现转换，这个过程在驱动中有一层拦截层，在虚拟机中有一层虚拟设备模拟层。

针对操作系统开关机等操作，本课题设计了一个专门为 KVM 虚拟机进行管理操作的管理系统，它负责对虚拟机进行上电、断电等操作。这个管理系统同时接收从硬件来的信号和系统内部来的信号以及人工信号，来进行对应的操作，通过系统命令来实现对虚拟机的开关等操作。

最后在硬件管理器实时性问题上，本课题设计了将虚拟 IO 操作移植到 KVM 内核内部来实现，并且在虚拟 I/O 和中断时采用 VT-X 技术，大大减小内核和客户机之间的切换，加大了这个设备虚拟的实时性。

虚拟设备管理器的总体架构如图 3-2 所示。

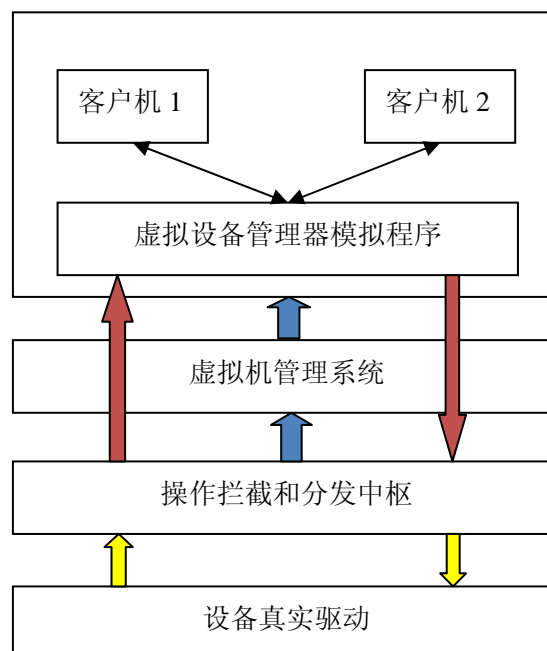


图 3-2 硬件管理器虚拟解决方案架构图

这样，整个系统虚拟化架构已经搭建完成，接下来研究具体的虚拟化解决方案。

## 3.2 PCI 硬件设备的创建和初始化

### 3.2.1 PCI 总线结构虚拟

对于计算机来说，一套最基本的硬件环境的基础就是一块主板，要在一个主板上虚拟出一个真实的硬件设备来就需要虚拟出一套真实的总线结构，然后再在想要的总线上挂载需要的设备，最后在每个设备上映射好固定的 IO 端口地址，来操作这个设备，在本节中主要研究 PCI 总线设备的构造，而 ISA 总线设备更加容易，类似于 PCI，本节就不多叙述。

在任何一個虚拟机中，都需要虚拟这些硬件环境，而在 QEMU-KVM 虚拟机中，QEMU 负责所有基础硬件环境的搭建和模拟，在传统 X86 架构下的初始化中，QEMU 会先跳入 PC 初始化函数 `pc_init1` 进行基础硬件结构的初始化。在这里会一次初始化需要用到的虚拟 CPU，虚拟端口，虚拟总线以及虚拟中断控制器等等。在这里还会查看客户机是否需要支持 PCI 总线，如果有这个需求则会初始化 PCI 总线，默认情况下主总线下挂载一根 PCI 总线。这个时候会调用 `i440fx_init` 这个函数来初始化所有总线结构。

`i440fx_init` 这个函数主要是将之前初始化的一些中断控制器，子总线等汇集到一起，然后创建主总线，主桥，然后连接到各个总线，这里是所有设备的根节点，以后设备就都挂在这个上面，而且通过索引能够找到所有的总线。在默认情况下 QEMU 会模拟一套最简单的 PCI 总线结构，所有的设备和 ISA 总线都挂载在根总线上面，如果我们需要模拟真实复杂总线结构就要重新构造总线。

在 QEMU 中，可以将根总线保存为全局数据结构，其它总线通过桥连接到根总线，通过查找和遍历可以访问根总线下面的所有设备。这样我们就通过软件的方式模拟出了一套自己的硬件总线结构。

在本系统中，设备控制器作为一个虚拟 PCI 硬件设备挂在从主 PCI 总线上引出的子 PCI 总线上，总线结构具体如图 3-3 所示，系统中默认从根总线中引出 PCI 总线，PCI 总线一边通过 HOST 总线与根总线相连，另外一边通过 PCI-ISA 桥与 ISA 总线相连，这里挂载遗留的 ISA 设备，同时通过 PCI-PCI 桥与子 PCI 总线相连。在默认 QEMU 系统里面把所有 PCI 硬件设备都挂载在主 PCI 总线上，然而本系统需要将设备挂载在子 PCI 设备总线上，所以，需要按照结构连接图来虚拟构造整个总线结构，而在下文中会详细说明 PCI-PCI 虚拟桥以及虚拟设备的构造过程。

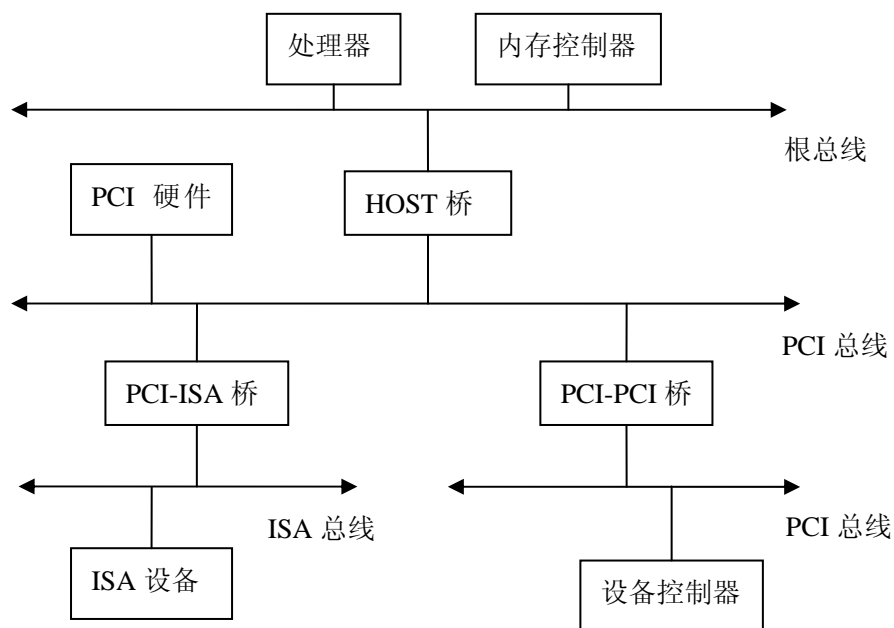


图 3-3 虚拟客户机模拟总线结果设计图

### 3.2.2 PCI-PCI 桥的构造

在前文已经说明了整个总线的虚拟化结构，然而在这个结构里面 PCI-PCI 虚拟桥是一个非常重要的地方，在这里进行详细阐述一下。在传统 QEMU 里面，一般的设备都有一个设备结构体，而所有的普通设备，桥设备，甚至总线都是用这个结构体来进行存储。然后通过复杂链表的结构将这些结构体串联到一起，这里主要是依靠 register 这个函数，设备一旦通过这个函数注册，在将来初始化的时候就会统一进行初始化。QEMU 和 Linux 模块的初始化很像，QEMU 在启动以后，QEMU 主线程会依次初始化所有设备，其中包括 PCI 硬件设备，这种设备初始化以后在它的内存地址里面就有了默认的值，但是接着会统一进行 RESET 重置。这也就是说所有的 bar 上地址会强制清除，接着把所有 bar 上的起始地址重新进行分配安排。

如前文所说 QEMU 在 X86 架构里面的总线虚拟默认将所有的虚拟设备挂在从主桥引出的第一条 PCI 虚拟总线上面，而且默认没有配备 PCI-PCI 虚拟桥结构，设备管理器是一个 PCI 硬件设备，需要挂载多根 PCI 总线，所以需要自己来构造 PCI-PCI 虚拟桥。从桥的字面就能理解，桥是两条总线之间的连接纽带，当然也需要配置相应的中断以及 IO 的区间，用来路由具体下游总线上的设备。因此，本系统可以将 PCI-ISA 虚拟桥的属性改成 PCI-PCI 虚拟桥，将桥的两端数据结构改成 PCI 虚拟总线。另外在 PPC 架构里面有一种 DEC 型号的桥，可以通过修改在 X86 中使用，主要修改 DEC 的头文件，将初始化函数共享，然后要用的文件中包

含其头文件，最后在目录 x86\_64-softhmmu\config-devices.mak 下面修改对应的宏变量，修改操作如图 3-4 所示。

```
22 CONFIG_VIRTIO_PCI=y
23 CONFIG_VIRTIO=y
24 CONFIG_USB_UHCI=y
25 CONFIG_USB_OHCI=y
26 CONFIG_NE2000_PCI=y
27 CONFIG_EEPROM100_PCI=y
28 CONFIG_PCNET_PCI=y
29 CONFIG_PCNET_COMMON=y
30 CONFIG_LSI_SCSI_PCI=y
31 CONFIG_RTL8139_PCI=y
32 CONFIG_E1000_PCI=y
33 CONFIG_IDE_CORE=y
34 CONFIG_IDE_QDEV=y
35 CONFIG_IDE_PCI=y
36 CONFIG_AHCI=y
37 CONFIG_DEC_PCI=y
```

图 3-4 DEC 桥修改配置文件后截图

将 DEC 强行的配置下去以后，然后在调用 i440FX 总线初始化的时候直接初始化 DEC 桥，具体总线和桥初始化修改代码如图 3-5 所示。

```
dev = qdev_create(NULL, "i440FX-pcihost"); /*创建 PCI 主总线设备*/
s = FROM_SYSBUS(I440FXState, sysbus_from_qdev(dev));
b = pci_bus_new(&s->busdev.qdev, NULL, 0); /*创建我们真正的 PCI 总线*/
s->bus = b;
qdev_init_nofail(dev); /*初始化主总线设备*/
d = pci_create_simple(b, 0, "i440FX"); /*创建主桥*/
*pi440fx_state = DO_UPCAST(PCII440FXState, dev, d);
piix3 = DO_UPCAST(PIIX3State, dev, /*创建 ISA 桥*/
pci_create_simple_multifunction(b, -1,true,"PIIX3"));

piix3->pic = pic;

/*接着连接 8259，IOAPIC 方式也同，只是扩展中断个数*/
pci_bus_irqs(b, piix3_set_irq, pci_slot_get_pirq, piix3, 4);
*pi440fx_state->piix3 = piix3;
pci_sub_bus= pci_dec_21154_init(b,-1); /*需要的 PCI 桥*/
dev = hardmanager_init(pci_sub_bus); /*挂载管理器设备*/
```

图 3-5 总线和桥初始化修改代码

### 3.2.3 PCI 设备初始化

PCI 硬件设备的虚拟化过程具有通用性，虚拟桥属于特殊的设备，和一般的虚拟设备主要的区别就是 PCI 设备内存的属性值不同，比如 class 和 bar 地址不一样。一般设备的 class 属性就是对应设备，而桥为连接设备。在桥内总线上分



配的设备的 IO 地址必须在桥所分配 IO 地址范围内，不然在 PCI 硬件设备地址刷新的时候将会强行重置该设备的 IO 地址，这个也是我们在虚拟一个设备的时候要严格遵守硬件的分配规则，不能够随意的虚拟。因此一般 PCI 硬件设备安装分为三步，一是初始化设备结构体，二是将设备挂到虚拟总线，三是注册虚拟 IO 地址。具体的设备功能虚拟化在后文中阐述。

硬件初始化关键代码如图 3-6 所示。其中主要是构造了一个硬件管理系统设备的结构体，其中包含初始化函数，设备名称，和读写配置文件函数，其中的 init 的初始化函数在 QEMU 开启的时候会被统一调用。

```
static PCIDeviceInfo hardware_manager_info={
    .qdev.name = "hardware manager",
    .qdev.size = sizeof(HMANAState),
    .init      = pci_hardware_manager_init,
    .config_write = pci_hardware_manager_write_config,
    .config_read = pci_hardware_manager_read_config,};
```

图 3-6 设备管理器虚拟设备结构体代码

### 3.3 设备中断虚拟

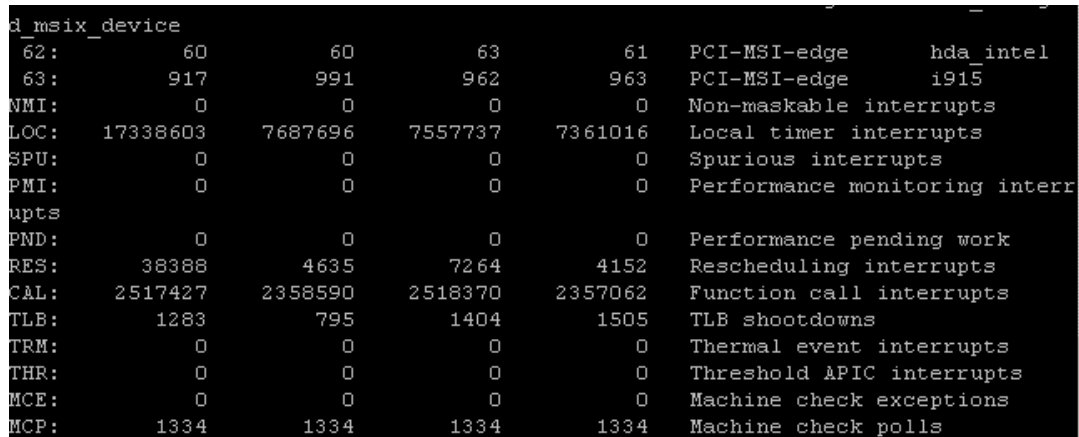
本课题研究的操作系统是针对 Linux 当时最新的内核 2.6.36 来实现的，所以 Linux 发展到现在，中断系统已经非常复杂，首先简单概括一下当前的中断环境，然后对中断进行分析和最后结合 VT-X 技术对其进行虚拟。

本课题的虚拟中断系统分为两个部分，虚拟机管理器部分和客户机部分。虚拟机管理器运行在 SMP 环境下，中断控制器主要使用的是 APIC，如时钟中断就使用的基于总线周期的本地时钟源。客户机部分由于运行在非 SMP 环境下，内部中断使用 PIC 控制器，如时钟中断就使用传统的类似 8254 时钟中断源。而本系统主要将虚拟机管理器感应到的中断投递到虚拟 PIC 设备上，然后再通过 VMX 指定的中断信息域注入到客户机操作系统。

#### 3.3.1 物理中断

随着硬件技术的发展，中断控制芯片已经不再是传统的 ISA 总线连着的简单 PIC 了，APIC，MSI，MSIX 等等的词语大家已经非常的熟悉。同时，Linux 内核也在不断发展，它在中断上的实现也越来越复杂。在 Linux 之中对于每一个中断有两个重要的数据结构与之对应，他们分别是中断门描述符 gate\_desc 和中断请求描述符 irq\_desc。这两个数据结构分别保存了中断处理过程和中断请求过程，

CPU 的作用就是得到脉冲以后将这两个数据结构进行对应。由于现在的中断更加复杂了，对于中断的模拟也会更加复杂，比如 PIC 中断占据物理中断前十六位，一般固定，所以在 QEMU 中虚拟需要提前进行保留，而对于其他中断就要分开处理，可以通过 `cat /proc/interrupts` 也可以看到这些中断与众不同，如图 3-7。这些中断左边显示的不是中断请求号，而是一个标识，右边显示的也不是中断控制芯片的信息。



	60	60	63	61		
62:	60	60	63	61	PCI-MSI-edge	hda_intel
63:	917	991	962	963	PCI-MSI-edge	i915
NMI:	0	0	0	0	Non-maskable interrupts	
LOC:	17338603	7687696	7557737	7361016	Local timer interrupts	
SPU:	0	0	0	0	Spurious interrupts	
PMI:	0	0	0	0	Performance monitoring interr	
upts						
PND:	0	0	0	0	Performance pending work	
RES:	38388	4635	7264	4152	Rescheduling interrupts	
CAL:	2517427	2358590	2518370	2357062	Function call interrupts	
TLB:	1283	795	1404	1505	TLB shutdowns	
TRM:	0	0	0	0	Thermal event interrupts	
THR:	0	0	0	0	Threshold APIC interrupts	
MCE:	0	0	0	0	Machine check exceptions	
MCP:	1334	1334	1334	1334	Machine check polls	

图 3-7 特殊中断详细列表

### 3.3.2 模拟中断源

无论设备处于什么中断环境，中断的实质都是一样的，中断分为硬件部分和软件部分，硬件部分负责物理上的连接和信号的触发，软件部分就是前面介绍的操作系统内部如何来组织中断请求和中断处理，而连接这两个部分的中枢就是 CPU。由于有 Intel VT-X 的支持，客户机需要的所有中断都可以通过 VMCS 数据结构传入到客户机。但是这些虚拟中断到底怎么产生，这个就需要根据不同的情况进行不同的分析，本文对这些中断源进行了下面的分类和补充。

**QEMU 中断源：**很多设备如键盘、鼠标、硬盘等都是靠 QEMU 来进行管理，然后通过 IOCTL 指令将中断产生的信息递交给 KVM，KVM 将中断投递到虚拟中断控制器，最后注入到客户机操作系统。因为 QEMU 是运行在用户态，所以可以在用户层检测用户的行为和中断需求，然后将需要的中断收集，然后放到虚拟系统中进行管理。

**辅助驱动程序中断源：**这类中断可以存在于 Linux 设备本身驱动中，这种驱动本身就兼容虚拟化操作，因此在它的真实驱动程序中有虚拟管理和触发虚拟中断功能，当要触发中断的时候我们可以通过内核模块之间的通讯来通知 KVM 模块有中断到来，然而 KVM 模块就可以进行接收和处理，这种方法在本课题中通过实验验证可行。

**KVM 模拟中断源：**这些主要是一些不需要 QEMU 干涉或者在性能上要求比较高的中断源，比如 8254 时钟中断，在 KVM 中 8254 时钟中断通过定时器来定时触发，然后进行中断的注入。

而本文中设备管理器的中断来自真实设备中断分发而来，因此需要对真实中断进行拦截，接下来讨论如何进行中断拦截。

### 3.3.3 物理真实中断拦截

在中断虚拟化过程中，我们为了得到某个设备的真实中断源，可以在 VMM 内核中的很多地方得到，这个过程也就是把物理设备的硬件中断信号进行拦截的过程，然后再通过一系列处理注入到客户机操作系统。

**PIC 设备中断：**不同的中断门指向对应 interrupt 函数地址，而这些 interrupt 函数又指向同一个中断处理入口 do\_irq 函数，传入参数为当前寄存器值，ax 中保存中断向量号，然后再通过这个中断向量号调用对应的中断处理函数，所以中断拦截可以在 do\_irq 中，也可以在分别中断处理函数中。

**APIC 设备中断：**这种中断一旦触发，则经过中断门直接跳转到各自对应的中断处理函数，中断拦截必须在中断处理函数中进行处理。

**修改驱动实现中断拦截：**在特殊情况下，为了实现某些特殊设备的虚拟化，需要修改设备在 Linux 中的真实驱动，增加一个通讯模块将中断信息有效的和其他模块进行交换和传输，实现中断的拦截。在这里我们直接在真实驱动中将中断信息发送到拦截模块，是的虚拟拦截过程更加简单高效。

### 3.3.4 中断的注入

虚拟机中断注入是先根据客户机的中断类型把中断投递到 VPIC 或者 VLAPIC 中，接着在 VM-enter 之前将中断写入 vmcs 的中断信息位。

KVM 通过函数 kvm\_set\_irq() 设置投递中断，然后分别调用 PIC 和 APIC 的投递函数，向两个控制器发送信号，最终有一种会成功接收到中断信号，为下次注入中断配置好环境。在重新进入客户机之前，会检查中断控制器，将悬而未决的中断事件写入到 VMCS 的中断信息域中。

中断的注入可以直接在 KVM 中也可以间接从 QEMU 里进行收集。QEMU 线程以 IOCTL 的方式通过 KVM\_RUN 向 KVM 内核模块发出运行 VCPU 的指示，后者执行 VM-Entry 操作，这个时候虚拟机就从内核模式进入到客户模式，然后进行用户的应用软件程序。这个时候如果发生外部中断事件或者 I/O 操作等，就会产生 VM-Exit，这个时候虚拟机又会退出到内核模式，这个时候就会从 VMCS 中读出退出原因，并且分析退出原因，包括中断相关信息，比如上次中断有没有成功等等，

然后开始考虑是否需要通过 IOCTL 退出到 QEMU 进行处理，如果处理完以后，然后再将新的中断等信息写入到 VMCS，重新进入客户机操作系统，具体如图 3-8 里以及 3-9 里所示。

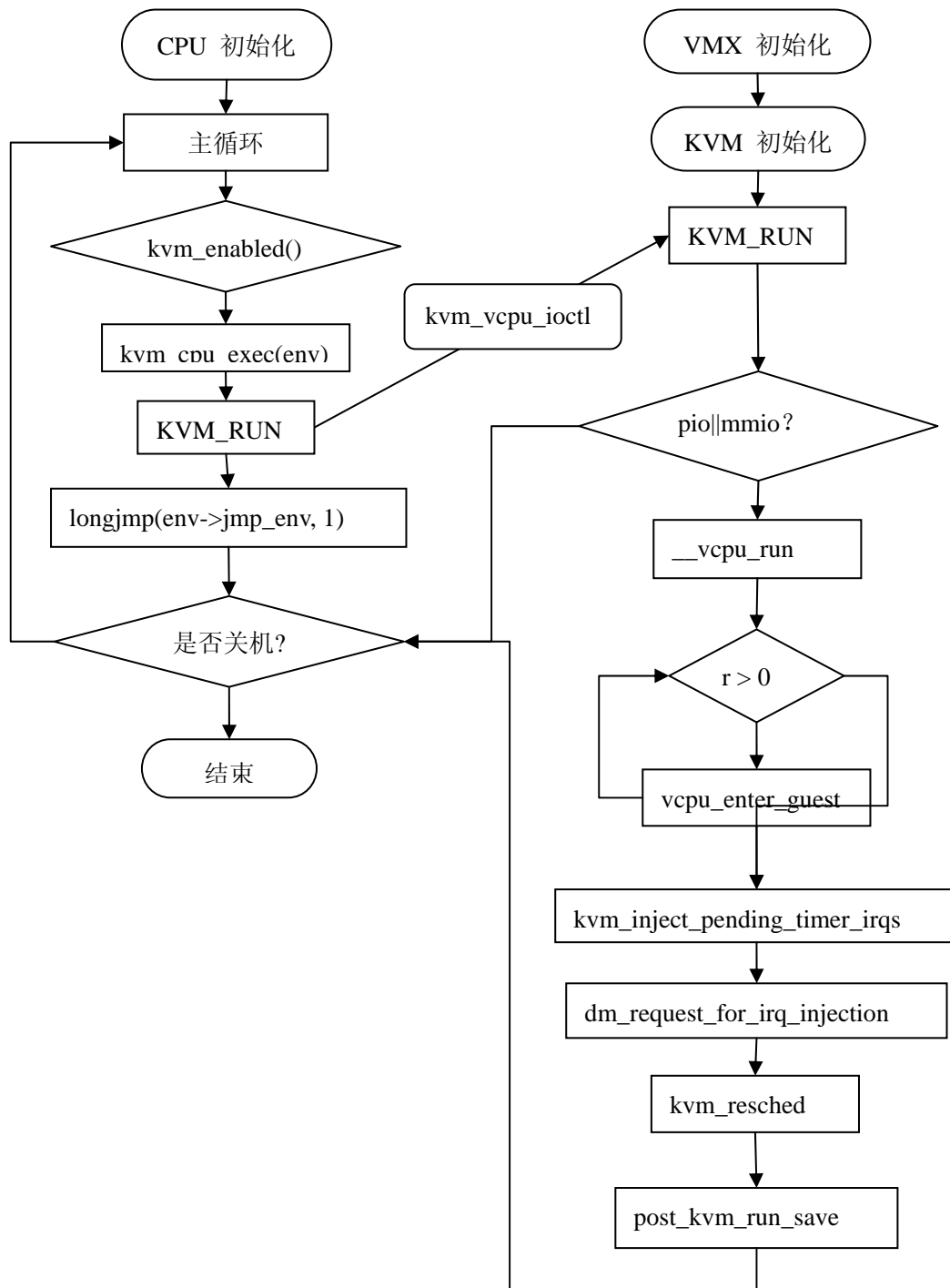


图 3-8 QEMU-KVM 客户机调度流程图

在上图中，左边为 QEMU 的 CPU 主循环，右边为 KVM 的主循环，两者通过 `ioctl` 相互联系到一起。这个大循环关系到整个虚拟机的运行状态，左边主要负责 CPU

指令的模拟和状态的控制，和各种硬件设备的模拟。右边则是 KVM 部分，是对左边部分的加速改进，这里虚拟机不断的进入客户机和退出客户机，通过 VT-X 技术得到退出客户机的原因，然后分析退出原因，比如图中的判断 IO 的方式，中断的情况以及调度的情况，判断完以后决定是直接处理还是其他处理，这样就完成了客户机 VCPU 的整个循环，而下面是中断在这个循环中注入相关的流程图。

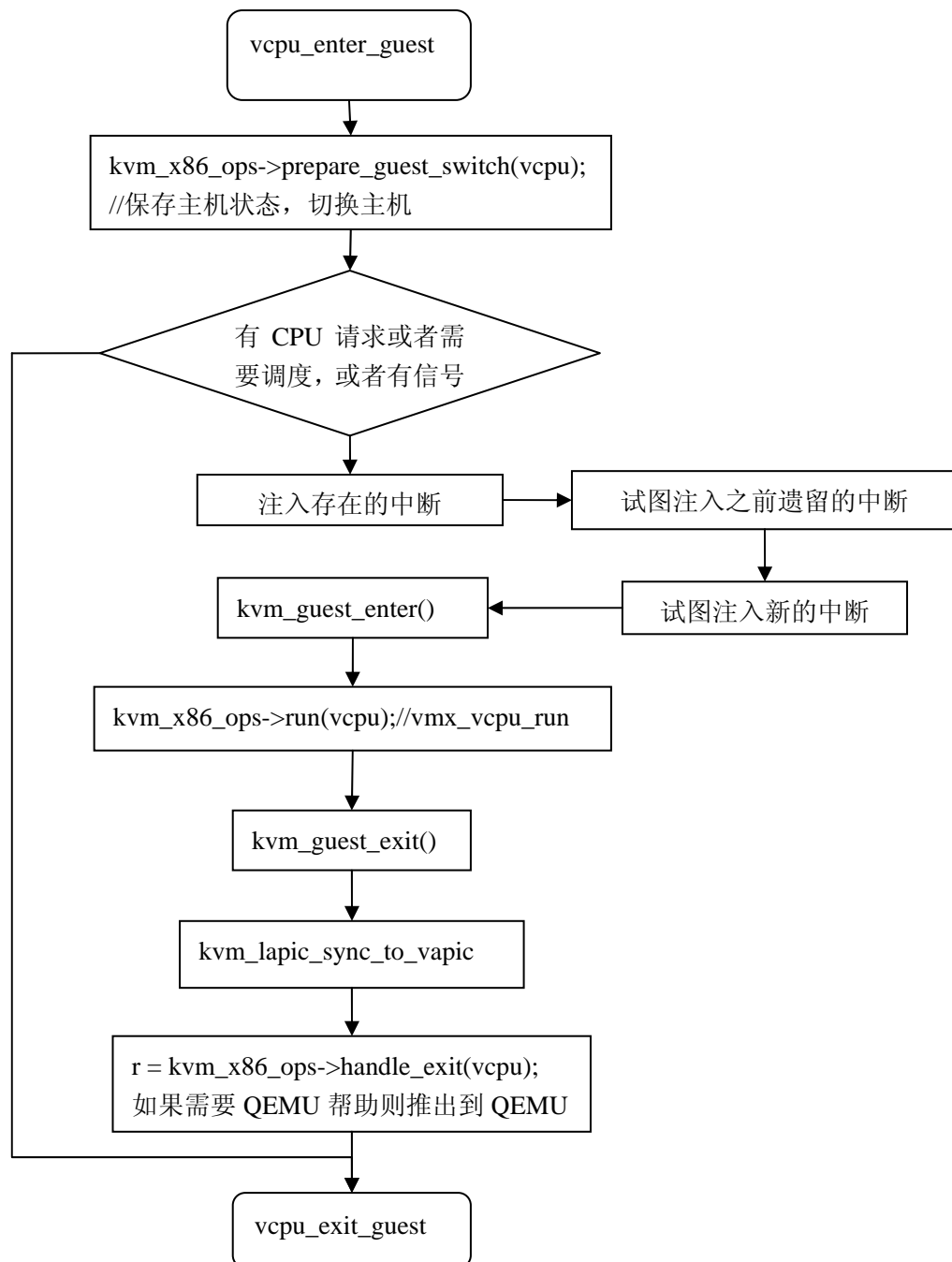


图 3-9 QEMU-KVM 中断注入流程图

在图 3-9 中，主要是虚拟机注入中断的流程图，其中 `kvm_guest_enter` 函数作用就是进入客户机，然后退出客户机，接着就会将系统中需要的中断同步到虚

拟中断控制器，最后通过 `handle_exit` 来处理退出的中断事件，选择是否需要返回到 QEMU，或者继续。

## 3.4 IO 虚拟

### 3.4.1 设备虚拟中 I/O 的注册

QEMU-KVM 虚拟机里面的模拟设备都是在 QEMU 中进行注册的，但是同时在 KVM 中也有些硬件与之对应，并且在这里进行拦截。所以设备的 I/O 注册就要从 QEMU 中进行分析。

与一般的 I/O 类似，QEMU 中注册 I/O 也分为两种，分别是端口 I/O (PIO) 和内存映射 I/O (MMIO)。

本课题主要的研究是使用 PIO 的方式来实现，这个也是最原始的 IO 实现方法。物理设备在主板上会被分配指定的 IO 端口段，用来对这个设备进行 IO 读写，所以，从驱动层需要做的是怎么来得到这段 IO 地址，怎么来使用这段 IO 段。然而反过来在硬件层就是需要怎么来注册这段 IO 地址，怎么来满足系统驱动的读写 IO 需求。

主要实现步骤如下例：

本研究中需要注册的硬件管理器需要分配 PCI 的 IO 地址。空间 0x800，映射函数 `hard_manager_ioport_map`。只需要通过图 3-10 中代码实现。

```
static int pci_hard_manager_init(PCIDevice *d)
{
    FPGAState *s = DO_UPCAST(FPGAState, dev, d);
    pci_config_set_vendor_id(d->config, PCI_VENDOR_ID_HDM);
    pci_config_set_device_id(d->config, PCI_DEVICE_ID_HDM);
    pci_set_byte(d->config + PCI_REVISION_ID, 0x02);
    pci_config_set_class(d->config, PCI_CLASS_BRIDGE_OTHER);
    pci_set_word(d->config + PCI_BASE_ADDRESS_1, 0x2001);
    pci_set_word(d->config + PCI_COMMAND, 0x1);
    pci_register_bar(&s->dev, 1, 0x800, 1, hard_manager_ioport_map);
    return 0;
}
```

图 3-10 QEMU-KVM 中断注入流程图

上面这个函数是在 QEMU 开始时被动触发的，其中设置了我们虚拟管理器设备的各种属性值，包括 class id，device id，第二条 bar 上的基地址，以及一个被动触发的 I/O 注册函数 `pci_register_bar`。从代码中还可以看出，实际上真正的注册函数是最后一个函数，其中指定了设备类型，使用哪个 bar，以及一

个地址触发函数的函数指针。在这里没有基地址参数，所以在 QEMU 里面设备基地址是动态分配的，任何设备都一样。然而前面我们设置的 0x2001 就是为了强行给 bar 上放入我们想要的基地址。

这样，本课题的硬件管理器的虚拟的 I/O 空间就成功的注册了。

### 3.4.2 KVM 中 I/O 操作拦截

从上文中我们已经知道了 QEMU-KVM 有内核和用户两个运行空间，QEMU 中主要负责设备的注册和部分处理，KVM 内核部分负责对 I/O 进行拦截处理，因为当客户机从非根模式退回到根模式的时候，先要到达 KVM 内核部分，所以，这个也是为什么可以进行拦截的原因。所以硬件设备如果要想实现拦截，就要在 KVM 内核中也进行一次设备注册，将设备放到 KVM 设备总线上，然而 KVM 中设备的注册就需要从 QEMU 中来发起。

(1) 在 QEMU 中调用 IOCTL 进行注册过程如图 3-11 里代码所示：

```
static int kvm_create_hard_manager(kvm_context_t kvm)
{
    r = kvm_vm_ioctl(kvm_state, KVM_CREATE_HDM, cpuid);
    if (r >= 0) {
        kvm_state->hard_manager_in_kernel = 1;
    }
}
```

图 3-11 QEMU-KVM 中断注入流程图

(2) 注册 KVM 设备，这里主要是对 KVM 总线设备进行添加，以及设备内存分配。

经过以上的两点，就已经将虚拟设备在 KVM 中进行了注册，也就是将 I/O 读写函数映射到了 KVM 里 PIO 数据总线上的某一段，以后只要因为 I/O 发生了虚拟机 Exit 时间，那么系统就会遍历整个 PIO 数据总线，这样在这根总线上的全部设备都会被搜索一遍，一旦查找到发生 I/O 事件的地址是属于某个具体的设备的 I/O 段，那么就会跳转到设备的 I/O 处理函数进行特殊的事件处理，这样就实现了拦截过程。从上面也可以看出来，如果发生 I/O 事件的地址不属于某个设备，那么系统将通过 IOCTL 重新回到 QEMU 中进行其他的处理，如果还没有系统将会出现异常，具体如图 3-12 所示。

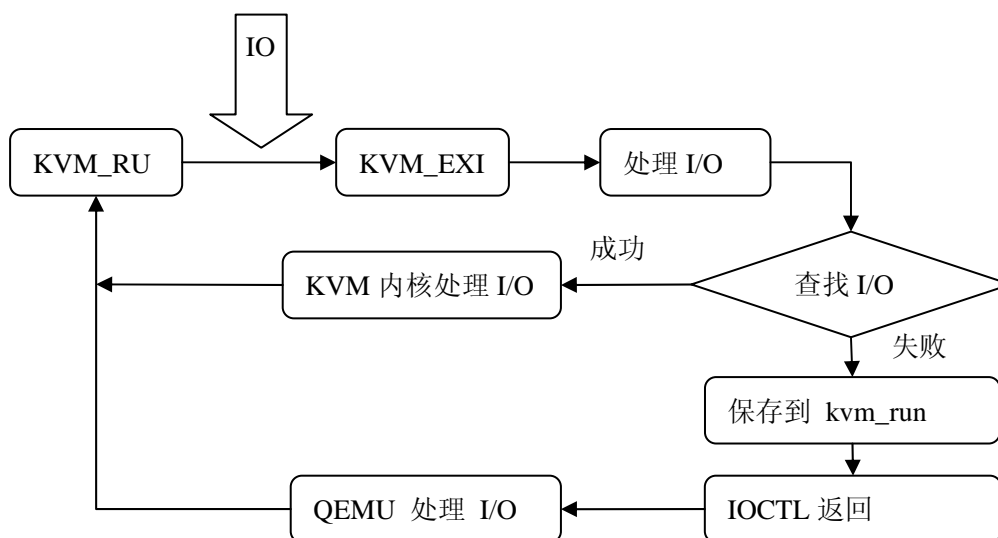


图 3-12 KVMIO 拦截

### 3.4.3 KVM I/O 读写操作处理

在 KVM 中我们在注册设备的 I/O 地址的时候会关联两个处理函数，它们分别负责设备 I/O 拦截以后的读和写处理：hardware\_manager\_ioport\_read 和 hardware\_manager\_ioport\_write。在这两个函数里面要充分发挥虚拟和真实中断的作用，配合 I/O 读写，满足客户机驱动的需求，而且这个和真实设备的功能密切相关。

I/O 读写处理是发生在拦截以后，所以当 I/O 地址经过掩码过滤以后，跳转到对应的处理函数就开始处理，这里需要为 I/O 数据创建一组缓冲区，因为要实现 I/O 数据的虚拟化，那么从真实设备来的数据需要先准备好，才能给虚拟客户机，而虚拟客户机写下来的数据，需要按位组装以后统一交给虚拟分发器，这样才能实现数据的虚拟和分发，实现设备的分时。

在 I/O 事件处理函数中都是按位进行读写的，所以在这里要通过 switch 语句对每一位进行过滤和分情况讨论。在某位出现特定的状态的时候触发虚拟中断，或者进行特殊的操作，这就是 I/O 数据虚拟化的精髓，也就是模拟一个真实硬件功能的关键。

### 3.4.4 硬件管理器 I/O 功能虚拟

硬件管理器的具体功能如前面章节描述所示，然而为了实现其 IO 需求，就需要对这些 I/O 数据进行有效的读写，存储，分发，转交给虚拟硬件，这个过程就是



I/O 数据虚拟的过程。其中就需要虚拟中断和各个模块协调通讯，最终来实现整体的虚拟功能。

数据的处理就是分别在 Linux 驱动层和虚拟机虚拟层分别建立接收和发送队列，这两个队列分别通过链表来存储通讯的消息。在 Linux 驱动层主要是和真实设备进行交互，然后把数据通过内核模块间通讯和虚拟层的数据进行交换。虚拟层得到数据以后在通过客户机的虚拟，和客户机的读写 I/O 事件进行交互，这个就是虚拟化的整个过程，具体步骤如下流程图 3-13 所示。

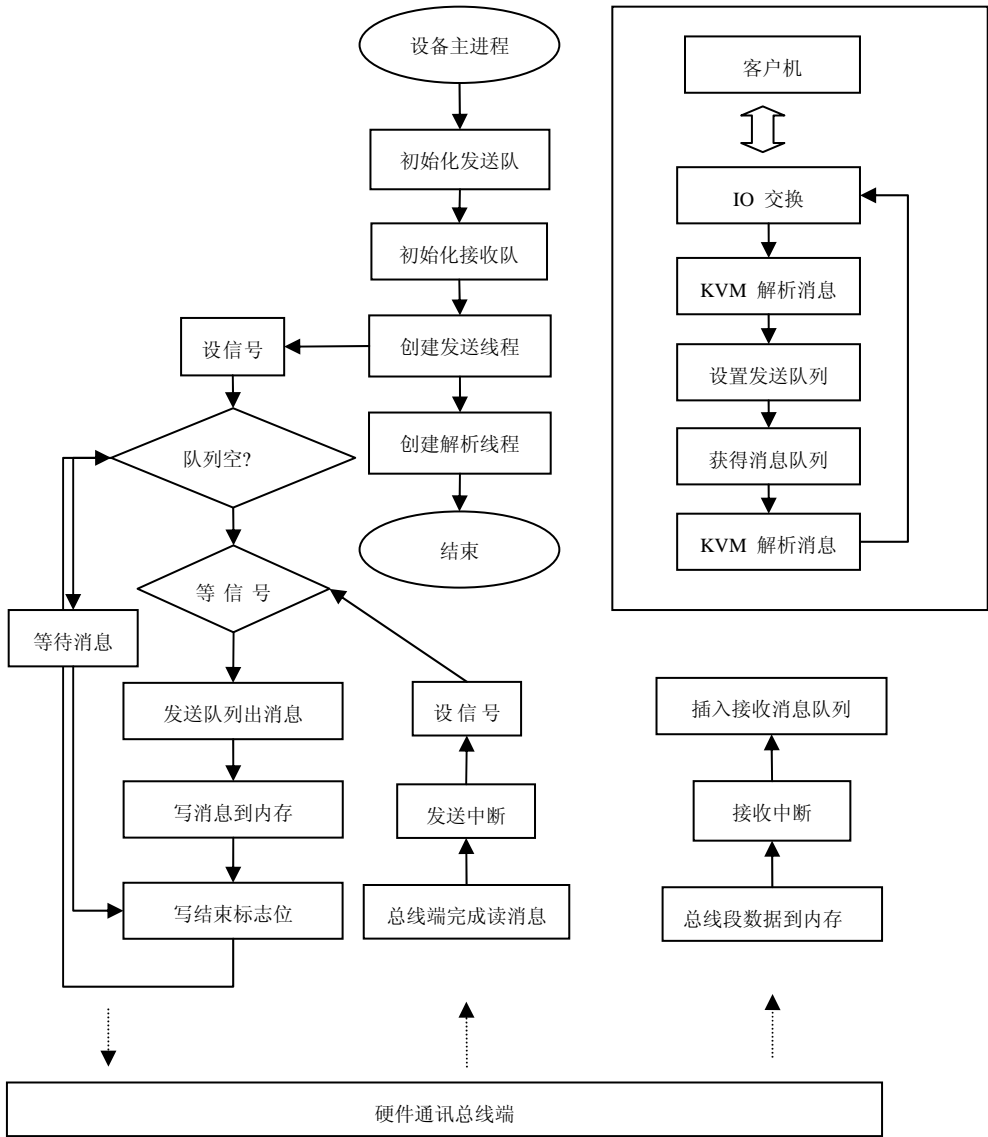


图 3-13 硬件管理器 IO 数据通讯流程图

在图 3-13 中，总共分为上下两个部分，上面的是硬件设备和本课题设计的系统，下面是连通各个主机 PC 的通讯总线。整个服务器集群通过硬件设备将各个节点计算机的硬件管理器连接到一起。上部分中，系统通过信号量来决定是否

读取内存中数据,而信号量又是依靠硬件中断来触发。下部分中,当数据到来时,通讯总线上的数据一旦发生改变将会同步到系统内存,并且在必要的时候触发中断。发送数据也类似,整体呈现有序的通讯循环。

### 3.5 KVM 管理系统设计

在实现了所有的硬件虚拟化以后，我们最后需要完成 KVM 管理系统的编写，这个系统的设计如前面总体设计中所说的，需要接收多个地方的消息输入，并且对应进行相应的操作。具体架构如图 3-14 中所示。

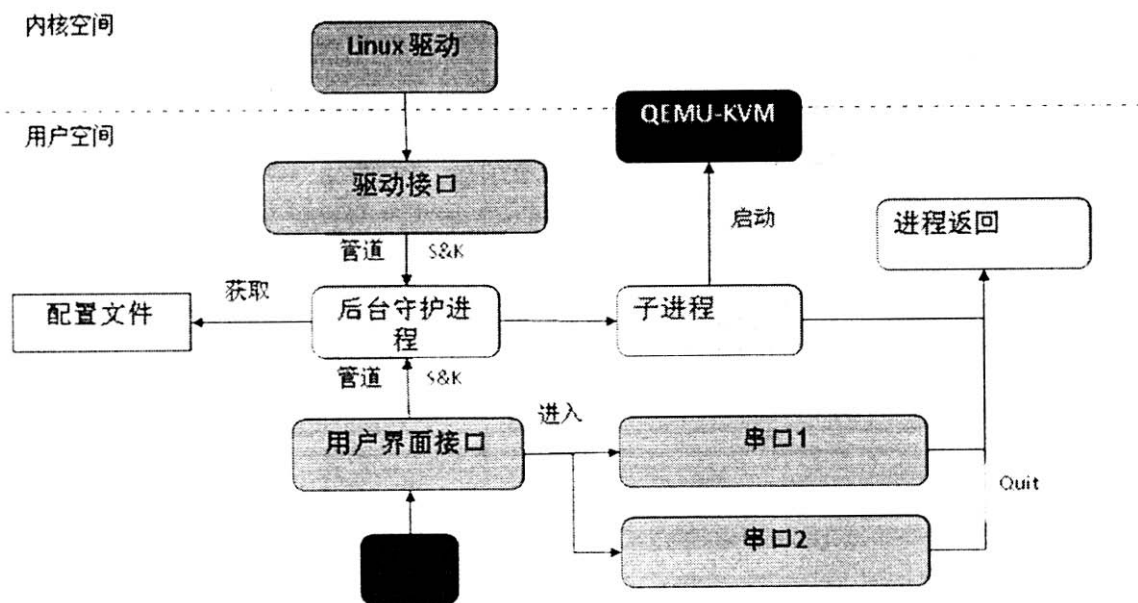


图 3-14 KVM 管理系统架构图

在图中，分为两个部分，上面为内核空间，下面为用户空间，内核空间主要与 Linux 真实驱动进行交互，因为我们的 Linux 的真实驱动和 KVM 模块都是内核空间，而管理程序和 QEMU 是在用户空间，所以我们管理程序有两个部分与 Linux 内核进行交互，一个就是 Linux 真实驱动数据接口，还有一个就是 QEMU 调用接口。然而之所以在管理系统中需要与真实驱动进行通讯，是因为很多硬件管理器的系统命令如开关机等直接发给 KVM 内核模块是没有用的，在 KVM 内核模块内部是无法控制整个 QEMU 的开关机状态，当启动多个 QEMU 以后只能通过外部命令来中断虚拟机的运行。所以本系统需要将这些命令发送到用户空间的集中控制系统来统一管理。

图 3-10 中间是是一个后台守护进程，通过管道接收驱动接口和用户界面接口来的命令信号，然后解析信号，最后启动子进程来对虚拟客户机进行强制操作，

如开关机。本系统将所有虚拟机的特殊操作都放在用户空间进行统一命令管理，这个管理就在这个守护进程中，用户只能通过菜单命令选择来对虚拟机进行控制，而不能直接使用 QEMU 系统命令，这样系统就能清楚每时每刻所有虚拟机的运行状态，并且能够正确解析硬件管理器发送过来的系统命令和用户输入的命令。后台守护进程当接收到开机命令的时候会创建子进程，并且将 QEMU 进程全部替换当前子进程，触发虚拟机的开启功能，在接收到关闭的时候主动通过系统调用杀掉虚拟机主进程，起到关闭系统的作用。

图 3-10 的最下面是与用户交互的部分，一方面直接与用户进行交互，接收系统命令，操作操作系统，另外一个方面可以将用户连接的串口直接导入 QEMU 的串口接口，进而连接到 QEMU 的输入输出接口，在 QEMU 中通过虚拟串口来连接到虚拟机内部，实现人机交互，所以这里主要针对串口虚拟化，而不是显示器的 VGA 虚拟化。

## 本章小结

本章主要深入研究了基于 VT-x 硬件辅助虚拟化技术的设备虚拟化方面的实现。总体上通过一个设备管理器的虚拟化实例来对整个过程进行详细的分析和研究，首先介绍了硬件管理器的主要功能，然后提出了硬件管理器的主要虚拟构架，接着针对虚拟总线、虚拟中断、虚拟 I/O 三个方面依次进行研究和实现，这里通过原理和代码进行了深入的分析，并且进行了理论阐述，最后设计了一个虚拟机管理系统来对整个虚拟机进行管理，配合来实现虚拟管理器的全部功能。

## 第 4 章 虚拟时钟丢失优化

时钟是一个操作系统的动力源泉，是各种操作的基准，如果没有时间整个系统都会混乱，然而在我们操作系统中有两种时钟，一种从开机到现在经历的时间也叫做墙上时间（Wall Time），另外一种是通过 RDTSC 指令得到的相对时间间隔。不同的操作系统内核中获取时间和计算时间的方法都不一样，然而通过时钟中断来驱动定时器计算时间的方式是都需要的一种方式，这种时钟中断在虚拟化的过程中就容易出现丢失情况，怎么来让时间更加准确，性能更加优异也成为了虚拟化中一个值得研究的问题。

### 4.1 Linux 基本时钟

#### （1）RTC 实时时钟

硬件时钟源，可以产生稳定的信号脉冲，频率一般在  $2\text{Hz} \sim 8192\text{Hz}$  范围内，可以初始化系统时钟等，起到时钟的驱动作用。

#### （2）PIT 可编程时钟

PIT 简称可编程时钟，可以通过设置参数来获得稳定的时钟中断源，这个也是比较传统的时钟，比较经典的时钟，以前的 PIT 都接在 8259PIC 中断控制器上，相应 0 号中断。因为比较简单，不具体详细说明。本文主要针对这种时钟进行理论说明和论证。

#### （3）HPET 高精度时钟

这种时钟源非常精准，在 Linux 内核中会优先使用配置有 HPET 的时钟源，它在 CPU 里面存在，和 PIT 功能类似，只是比 PIT 更加精准，同样响应 0 号中断，现在已经取代 PIT 了。

#### （4）TSC 时间戳计数器

TSC 是一个时钟计数器，也属于 CPU 的一个寄存器，随着 CPU 指令的执行不断计数。性能上相对比较精准，可以用来校准系统时钟。

#### （5）Local Timer 本地时钟

这种时钟是针对本地时钟，针对多处理器存在的，每个核上都有对应的时钟中断计数器，精度相比 PIT 也准确的多，Linux 内核在启动以后也会优先于 PIT 使用。

## 4.2 系统时间的丢失

### 4.2.1 系统时间计算

操作系统在启动时会读取 CMOS 的时钟，或者通过 NPT 协议，得到系统绝对时间。然后又可以通过时钟中断来得到系统相对经历的时间，这样就能够得到系统随时的绝对时间，同时系统还会通过每隔几秒钟读取 CMOS 时间来校正当前时间。

当前时间 = 系统启动时间 + 运行时间

在一个正常的系统里面，假设有两种时间概念，一种是系统运行的时间，一种是自然增长的时间，那么当有一个系统运行的时候，当前的系统时间应该等于自然的时间，它们正确的时间关系应该图 4-1 所示。其中假设当前时间为  $t_0$ ，经历的时间为  $(t_1 - t_0)$ ，那么当前的时间就为  $t_0 + (t_1 - t_0)$ 。

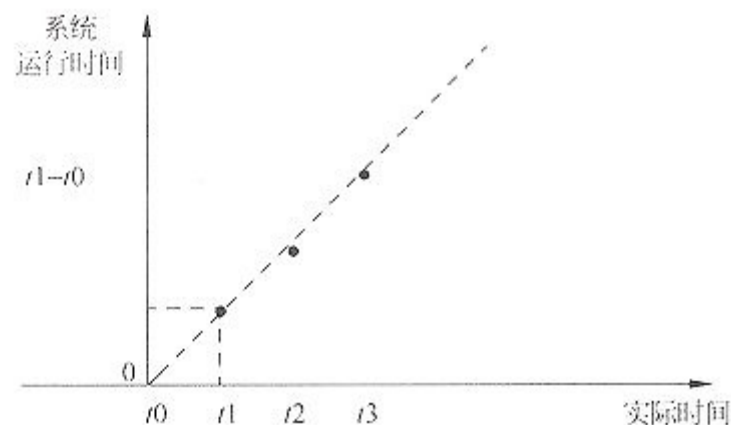


图 4-1 一般系统的时间关系图

然而在虚拟机中，由于多个虚拟机之间是分时共享硬件资源的，因此无论某个客户机是否在调度运行，自然增长的时间都是在不断变化的，所以就会出现一种情况就是某个客户机被调度出来没在运行的时候，系统时间停止，然而自然时间增加，造成时间不一致。如图 4-2，在时间  $t_2$  的时候系统被调度出去，虽然实际时间在增加，但是系统时间停止了，等客户机系统恢复的时候还是从以前的时间开始继续计算时间。

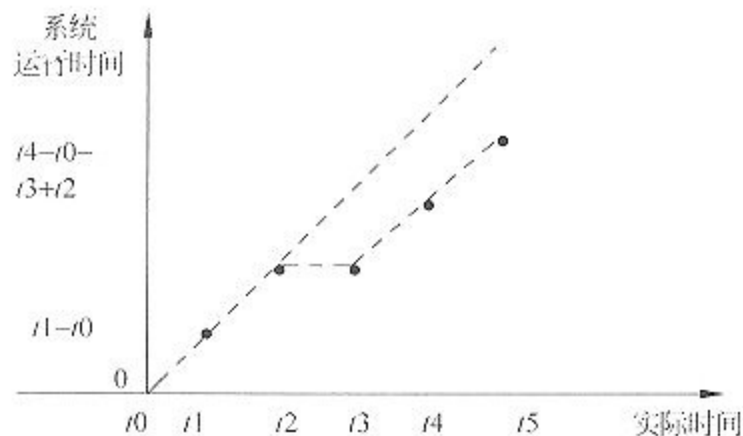


图 4-2 时钟丢失时候的关系图

### 4.2.2 时钟中断的丢失

在上文中可以看到在客户机切换出去的时候，系统时钟和真实时钟发生了偏差，而我们在操作系统中，来度量这个时钟偏差的最小单位就是时钟中断，时间的丢失就相当于时钟中断的丢失。在 VMM 系统里面不管是采用何种硬件中断源，时钟中断都是源源不断的产生的，然后通过转换后再通过 VT-X 技术写入 VMCS 的中断信息位，然后改变客户机时钟。而在某个客户机的操作系统被调度出来，虽然客户机的中断还是在继续注入，但是由于此时处于中断屏蔽阶段，中断无效，当客户机被唤醒以后，中断重新被注入，但是之前的时钟中断已经丢失，对于这种情况，系统时间将会落后于真实时间，与现实情况发生偏差。

### 4.2.3 时钟中断的耗时

在时钟中断在虚拟化过程中会发生丢失的同时，它还会产生很大的性能损耗，拿 8254 时钟来举例，如果 8254 时钟作为系统时钟中断源，系统所有维护的所有定时器的时钟源都来自这个中断，在很多系统中，会有一个专门的进程来维护这些定时器，当时钟中断处理时间受到虚拟化影响时，这个维护定时器的进程将使用更多的时间，因为时钟中断的频率是非常快的，所以，这个中断耗时的增加将是成倍的，并且使得时钟进程的耗时成倍增加，严重影响时钟的性能，甚至影响到每个定时器的更新的精确度。

## 4.3 时钟虚拟化处理和改进

虽然客户机系统时间与真实时间发生了偏差是极其微小的，可能一般人用肉眼无法察觉，但是时钟的不够精准对于一些高性能实时系统是不能容忍的，本课题在这里提出了一系列解决方法并且通过实例测试加以证明。

因为不管是什么时钟源，时钟中断原理都一样，所以本文以 QEMU-KVM 8254I 虚拟时钟来进行说明。它简单而且容易理解。

### 4.3.1 虚拟时钟准确性改进

在 KVM 时钟初始化的时候会创建一个 Hrtimer 计时器，然后将这个计时器作为我们的虚拟硬件时钟源，提供稳定，持久的时钟中断输出，这个就是平时所说的周期性的时钟源，每当计时器到时一次，虚拟机将产生一个虚拟时钟中断，注入到客户机系统，客户机的 8254 时钟驱动就会接收到这次硬件中断。

但是在我们系统的运行过程中，如前面提到客户机是会被调度出来，然而此时的时钟中断将无法注入，要解决这个问题可以设计一个时钟中断计数器，来统计所有的丢失时钟中断，然而将这些时钟中断在系统恢复以后集中注入到客户机操作系统。

在这里一方面可以使用一个时钟计数器 pending，它保存着积累的时钟中断次数，当 HT 定时器到超时，虚拟机就会将时钟注入标志位会被置位，同时等待注入的时钟计数器 pending 会原子性增加 1，此时并不一定会理解将中断注入到客户机，但是虚拟机会在系统调度后以最快的优先级将所有遗留的时钟中断注入到客户机，注入以后虚拟 pit 的中断返回 ack 函数会将计数器 pending 原子性减 1，这样就完成一个时钟中断注入。如果此时一直无法注入时钟，pending 将会不断累积，但是最终将会注入到客户机。注入过程如图 4-3 里所示。

具体过程如图 4-3。在上图中，左边是虚拟机退出以后的处理过程，右边是 Hrtimer 定时器的触发过程，一旦右边定时器超时，就会将 pending 增 1，然后在左图 handle exit 以后将会感知到有时钟中断到来，然后注入时钟中断，如果注入成功，则 pending 减 1，直到 pending 变为 0，说明积累的所有时钟中断都已经注入完成。这个机制也保证了时钟都能够注入到虚拟 8259 中，但是可能有些中断会被延后注入。

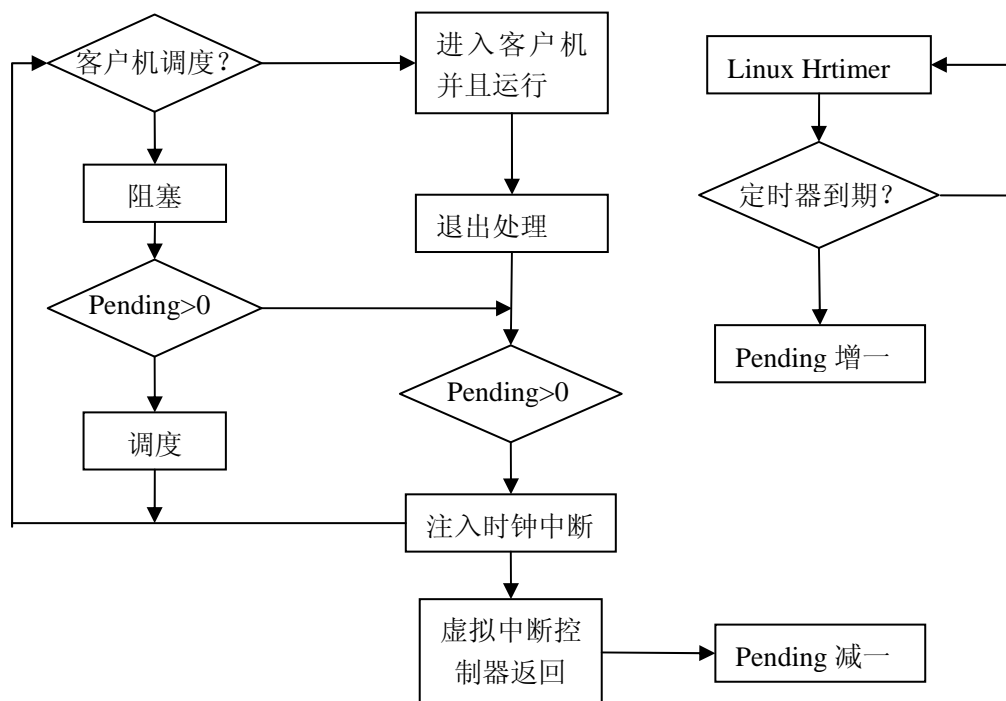


图 4-3 虚拟 8254 注入原理流程图

通过上面方法虽然能够保证多次时钟中断不被合并，并且找回丢失的时钟，但是还是会出现一个问题，当客户机被调度出来的时间过长时，时钟中断累计数 pending 值就会不断增加，以至于无限变大而得不到有效的恢复，这样的话可以使用下面的优化方法来处理：

1. 如果出现时钟中断积累，有两种可能，一种是客户机时钟中断系统出现问题，这种情况下系统不正常，不需要解决。另外一种就是客户机调度频繁，无法接收时钟中断的时间太多，造成时钟积累，这样就可以在一次退出操作系统的过程中，写入多个积累的中断操作并且将 pending 值按数量级减少，这样需要修改 VMCS 中断位，一次写入多个时钟中断。让系统在下次退出前，将继续剩余未消化的时钟中断，返回给虚拟中断控制器，然后再同步到 pending 计数器，也就是说把一般的增一减一提高到一些，这个增减值可以在随着 pending 值得变化动态增减。

2. 而对于一些系统对系统时间的高精度追求，可以通过操作系统 RDTSC 指令来主动校准系统时钟中断次数，让操作系统主动来读取 TSC 时间，并且主动校准系统时间，在此同时监控中断累计数 pending 的变化情况，如果到达一定的限度则清空这个值，重新开始计数，这个对于持久稳定运行的系统很有必要，这样能够避免时钟中断的无限积压，是一种保险手段。

通过上面这套解决方案，就能够稳定安全的给客户机操作系统提供虚拟持续的时钟中断，维持客户机系统时钟的稳定运行。



### 4.3.2 虚拟时钟性能上的提高

由于 QEMU 和 KVM 分别处于用户层和内核层，传统 QEMU 时钟中断需要将中断先从 QEMU 通过 `ioctl` 转移到 KVM，然后通过虚拟中断控制器以及 VMCS 进入客户机，但是 QEMU 和 KVM 之间的相互切换需要耗费一定的性能，因此将 8254 时钟 IO 处理全部从 QEMU 中移动到 KVM 内部来实现，这样就可以避免每次都退出到 QEMU 来，运行性能就能够提高。

由于注入的 8254 时钟中断频率远远低于客户机实际所有的退出和进入频率，所以可以在每次退出客户机后检查是否有时钟中断需要注入，如果有则在下次进入虚拟机之前将时钟中断注入到虚拟 8259 中断控制器，然后通过 VMX 同步到客户机。

### 本章小结

本章主要分析了时钟在虚拟化过程中出现的两种问题，一个是精准度的问题，还有一个是性能损耗的问题。首先一部分详细分析了会出现这两个问题的主要原因，并且进行了理论论证，然后提出了解决这两个问题能够采取的方法，通过在时钟虚拟化中对时钟中断的改进，并且针对这两个问题提出了一套非常有效的改进方案，通过一个时钟中断保存-重注入方式以及一些减少虚拟机退出的方法使得时钟虚拟化更加稳定和高效。

# 第 5 章 虚拟 PCI 设备 I/O 访问起始地址修改

在前文中已经讨论过如何虚拟一个 PCI 硬件设备，如何分配 I/O 地址，在这里我们将着重讨论 I/O 地址映射的问题，因为在真实情况里，很多嵌入式硬件设备，比如服务器上的 FPGA 里面的设备，这些设备一旦与主板物理连接，其中的 I/O 访问起始地址就已经确定，在操作系统中，驱动程序开始使用设备。然而有一些驱动程序里面的数据不是通过 class 和 id 来读取设备，而是通过直接 I/O 访问起始地址来注册设备，在这种情况下，这种方法就会出现问題，因为我们虚拟出来的设备将会被随机在空白区添加一个 I/O 访问起始地址，这样会造成有些驱动无法识别到虚拟设备，所以本课题提出一套如果修改 I/O 访问起始地址的方法来解决这个问题。

## 5.1 PCI 结构

PCI 硬件设备里面有几段内存用来存储 PCI 设备的基本信息，这些信息保存在不同的区域。要对 PCI 设备进行很好的虚拟就需要对 PCI 设备内存每一位存储的东西有所了解。在 PCI 设备内存空间中的信息中有一些是出厂设置，还有一些是用户可以改写的地址，我们在虚拟这种设备的时候需要写入出厂设置，这样才能模拟一个完整的设备，当然这些信息也可以在真实设备里面读出来，如果不存在真实设备，那么就需要完全自己构造。具体内存结构如表 5-1 里所示。

表 5-1 PCI 数据结构表

	0x0	0x04	0x08	0xc
0x00	vendorID devID	command		
0x10	bar0 address	bar1 address	bar2 address	bar3 address
0x20	bar4 address	bar5 address		
0x30			Interrupt line	

在表中，横纵坐标的数字都是表示 PCI 内存空间内的相对偏移地址，总共由 4 个 16 字节位表示这些数据，表格中就是相应地址空间表示的内容含义，比如从 0x0 开始就是我们辨识这个 PCI 硬件设备的相关 ID，从 0x10 开始就是每个 PCI 硬件设备相对应的六段 bar 地址空间，这里存储着六段地址映射，分别可以用作不同的作用。

## 5.2 QEMU-KVM 中 I/O 访问起始地址的指定分配

### 5.2.1 I/O 起始地址的主动分配和统一刷新

为了使得 PCI 硬件设备能够得到固定的 I/O 地址，就要从根源上来过滤操作系统对虚拟硬件设备内存的读写。如前面所说在 PCI 内存地址中有多个 bar 地址，里面就保存着对应映射 I/O 的基地址，只要不改变这些部分地址，PCI 硬件设备的起始地址就能够被固定住。在 QEMU 中，所有 PCI 设备起始地址默认是空，当系统启动以后设备发生第一次重置时，还会清空所有的 PCI 硬件设备每个 bar 上面保存的初始地址，接着会刷新这个地址，这个时候会发现设备 I/O 起始地址不合法，然后通过读写 I/O 配置地址的数据，来对具体设备进行起始地址分配。这个主要通过 config\_read 和 config\_write 这组函数来进行地址的分配，主要还是遵照设备的先后顺序，将新设备不断安排在高地址，默认 QEMU 新设备一般在 0xa000 以上地址段。在系统的运行过程中，如果发现 I/O 地址存在重叠现象，那么将再次重新重置设备 I/O 起始地址，然后再次分配一个新的没有冲突的 I/O 空间。在 QEMU 中，设备的 I/O 地址区间不仅受到整体 I/O 地址的影响，还受到其直接父桥上 I/O 地址的影响，必须在合理地址范围内，不然所有相关设备依然会重置地址。虚拟设备如何指定分配 I/O 起始地址在上文中 PCI 设备部分已经给出，所以在这里主要给出刷新部分代码如图 5-1 所示。

```
void pci_device_reset(PCIDevice *dev)
{
    for (r = 0; r < PCI_NUM_REGIONS; ++r) {
        /*遍历所有的 region， region 就是 bar，清空 region 里面的 IO 地址*/
        PCIIORegion *region = &dev->io_regions[r];
        if (!region->size) continue;
    }
    if (!(region->type & PCI_BASE_ADDRESS_SPACE_IO) {
        pci_set_quad(dev->config + pci_bar(dev, r), region->type);
    } else {
        pci_set_long(dev->config + pci_bar(dev, r), region->type);
        pci_update_mappings(dev); /*刷新设备*/
    }
}
```

图 5-1 刷新虚拟设备函数代码

上面的代码主要是对设备进行刷新，这里是刷新部分的核心代码，主要就是循环虚拟设备的每个虚拟 bar 空间，看里面的地址是否合法，如果不合法则清空，并且调用 update mappings 从新映射新的 I/O 地址。所以如果要固定之前分配的

I/O 起始地址，就可以在将地址设置成能够在这里判断成合法地址，或者是直接跳过这个重置函数，不做任何的刷新操作。

### 5.2.2 PCI 虚拟设备 I/O 地址和虚拟桥 I/O 地址关系

因为 QEMU 中 PCI 设备地址受到桥地址的影响，两个地址要合理才能够正常使用虚拟设备，不然系统将一直重复刷新虚拟设备 I/O 地址，因此不仅要指定好虚拟设备的 I/O 地址，还需要指定好相应桥的 I/O 地址，代码如图 5-2 所示：

```
pci_bridge_filter(PCIDevice *d, pcibus_t *addr, pcibus_t *size, uint8_t type)
{
    base = MAX(base, pci_bridge_get_base(br, type));
    limit = MIN(limit, pci_bridge_get_limit(br, type));
    if (base > limit)        goto no_map;
    *addr = base; /*匹配成功*/
    *size = limit - base + 1;
    return;
no_map:
    addr = PCI_BAR_UNMAPPED;
    *size = 0;
}
```

图 5-2 虚拟桥和虚拟设备匹配代码

从上面函数得到，经过桥过滤时设备地址会重新和桥地址进行检查，看设备是否在正常地址范围内，比如 I/O 起始地址是否大于桥的起始地址，I/O 长度是否小于桥的长度等等，如果不符合则重新刷新设备 I/O 地址。

因此这种情况下只需要给虚拟桥设备一个合理的地址，这样桥地址就不会因为冲突进行刷新，或者直接赶快在桥地址重置的时候也跳过重置函数，一直保持初始指定的正确 I/O 地址。桥初始化函数添加的修改核心代码如图 5-3 所示：

```
pci_bridge_filter(PCIDevice *d, pcibus_t *addr, pcibus_t *size, uint8_t type)
{
    base = MAX(base, pci_bridge_get_base(br, type));
    limit = MIN(limit, pci_bridge_get_limit(br, type));
    if (base > limit)        goto no_map;
    *addr = base; /*匹配成功*/
    *size = limit - base + 1;
    return;
no_map:
    addr = PCI_BAR_UNMAPPED;
}
```

图 5-3 虚拟桥和虚拟设备匹配代码

### 5.2.3 虚拟设备跳过 I/O 重置直接刷新

通过上文已经将虚拟设备 I/O 地址进行初始化指定，并且同时将附属的虚拟桥设备的 I/O 地址指定在合理范围内。因此最后只要在设备 I/O 地址冲突以后调用重置函数时候跳过之前分配的地址，直接进行整体的刷新就能实现 I/O 基地址的指定分配了，其中最核心的判断如图 5-4 所示：

```
void pci_device_reset(PCIDevice *dev){
    if(strcmp(dev->name,"hard_manager")==0){
        pci_update_mappings(dev);
        return
    }
    ... ..
}
```

图 5-4 改进后设备刷新函数代码

按照上面所说的方法就能实现 I/O 基地址的指定分配，分配以后可以通过 cat /proc/ioports 命令在 Linux 下面进行验证。

### 5.2.4 虚拟 PCI 配置读写函数的重写

虽然通过上面的方法已经能够初步实现之前的目的，但是由于特殊情况，需要不仅跳过虚拟设备的 I/O 地址刷新，还需要将读写配置函数 config\_write 重写，并且将所有重置 I/O 地址空间的写入操作都过滤。

如前文讲述 QEMU-KVM 里面的虚拟设备 I/O 地址是通过对 config 位进行读写后然后系统再对指定的 I/O 地址区域进行读写，实现动态的 I/O 地址分配。所以这个读写函数就是改变 I/O 地址的最直接地方，只要操作系统将新的地址写入到硬件内存，硬件虚拟地址就发生了改变，所以这里可以进行过滤，不允许操作系统更改这个地址。当然这个有点强制性，其中代码如图 5-5 所示：

```
void hardware_manager_write_config(PCIDevice *d, uint32_t addr, uint32_t val, int l)
{
    if(addr = 0x10) pci_default_write_config(d,addr,0x20,l);
    else pci_default_write_config(d,addr,val,l);
}
```

图 5-5 改进后设备刷新函数代码

和上文类似不仅要对单独的虚拟设备进行修改，还需要对相应的桥进行修改，具体修改如图 5-6 所示：

```
static void pci_dec_bridge_write_config(PCIDevice *d,uint32_t address,
uint32_t val,int len)
{
    if ((address == 0x1c || address == 0x1d)&&pci_get_byte(d->config +
0x1a)==0x5)
        pci_bridge_write_config(d,address,0x20,len);
    else    pci_bridge_write_config(d,address,val,len);
}
```

图 5-6 桥设备写配置修改后代码

通过上面方法就彻底能够实现指定 I/O 起始地址了。

## 本章小结

本章最开始介绍基本的 PCI 硬件设备，然后依次分析了 QEMU 中如何实现总线，桥和设备的模拟过程，并且分析研究了约束虚拟设备 I/O 地址的一些根本的原因，最后从源头提出了一套解决方案来对总线进行扩展和虚拟 I/O 地址的分配。本章提出的方案首先初始化硬件 PCI 内存 bar 地址，然偶跳过 PCI 硬件设备的 I/O 地址的刷新并且强行指定分配 PCI 硬件设备的 I/O 地址以达到我们的需求，这样做能够适应更多的特殊操作系统的需求，而且还能够让大家更了解 PCI 硬件构造和交互功能原理。

## 第六章 实验测试与结果分析

### 6.1 基于 VT-X 的 QEMU-KVM 的性能对比测试

传统 QEMU 虚拟机是完全通过软件陷入模拟方法实现的虚拟机，没有加任何的硬件辅助虚拟化相关技术。而 QEMU-KVM 虚拟机是通过 KVM 内核模块加速，添加了 Intel 公司的硬件支持技术，通过高效的使用硬件辅助虚拟化技术，实现性能上的大幅度提高。本节主要针对 CPU 运算和内存，IO 等方面对纯软件 QEMU 和带入硬件辅助虚拟化技术的 QEMU-KVM 以及 QEMU-KVM 和单一操作系统两种情况下进行性能上的测试比较。

#### 6.1.1 测试环境配置条件

课题的各项测试在统一的硬件和软件配置环境下进行，具体配置如表 6-1 和表 6-2 所示。

表 6-1 系统硬件配置

硬 件	配 置
厂商型号	Dell optiplex980
CPU	Nehalem/3.33GHZ/4 核/2 级 Cache/4096KCache 容量
内存	8GB
硬盘	16M Flash(当前为 2GB U 盘)
网卡	NIC 82576
串口	1 个

表 6-2 软件配置

软 件	软件包	应用程序	依赖的内核程序
OS	Debian	Base commands	/
Kernel	Linux 2.6.36	N/A	KVM module
Qemu_kvm	Qemu-kvm-0.14.0	Qemu-system-x86_64	Kvm kvm-intel.ko
Consle	Consle_cx	Consle_cx	N/A
Bond_process	Bond_process	Qemu-system-x86_64	N/A

用到的专业测试软件：Unix Benchmark 界面如图 6-1 所示。

```
# # # # # # # # ##### ##### # # ##### # #
# # ## # # # # # # # # # # # # #
# # # # # # ## ##### ##### # # # # #####
# # # # # # ## # # # # # # #
# # # ## # # # # # # # # # #
##### # # # # # #

Version 5.1.3                                     Based on the Byte Magazine Unix Benchmark

Multi-CPU version                               Version 5 revisions by Ian Smith,
January 13, 2011                               Sunnyvale, CA, USA
                                                johantheghost at yahoo period com
```

图 6-1 Unix Benchmark 截图

6.1.2 纯软件虚拟化和基于 VT 技术的 KVM 虚拟化对比

1. 测试条件

在各种条件下执行一段运算程序，包括浮点，堆栈等运算，并且经过 8 次重复测试。

2. 测试目的

通过比较得到加入硬件辅助虚拟化，虚拟机在 CPU 运算性能上提高了多少。

3. 测试结果

经过 8 次重复测试得到结果如表 6-3 所示。

表 6-3 运算测试结果

system	测试次数序号									
	1	2	3	4	5	6	7	8	9	10
Host	78.669	83.380	79.202	78.499	83.519	74.718	74.348	71.769	71.893	72.698
Guest1	117.922	118.755	87.340	87.156	90.212	89.179	87.822	87.359	87.369	87.317
Guest2	98.027	120.306	86.863	88.67	88.852	89.262	87.692	87.000	87.269	87.301
Guest3	1017.822	1019.635	887.252	988.356	818.712	807.166	997.342	997.222	996.998	997.327
Guest4	953.499	1053.451	1054.827	954.135	956.716	956.400	954.835	954.567	954.847	954.895

Host：单独运行测试工具测得 CPU 所耗时时间。

Guest1：加入 Intel VT-X 技术支持的客户机 1 内部测试结果。

Guest2：加入 Intel VT-X 技术支持的客户机 2 内部测试结果。

Guest3：未加入 Intel VT-X 技术支持的客户机 1 内部测试结果。

Guest4：未加入 Intel VT-X 技术支持的客户机 1 内部测试结果。

从结果中可以分析，加入 VT-X 技术支持以后虚拟出来的 CPU 能够在性能上提升将近十倍。



### 6.1.3 纯软件虚拟化和基于 VT 技术的 KVM 虚拟化对比

#### 1. 测试条件

试用 Unix Benchmark 软件对操作系统进行性能评测。

#### 2. 测试目的

通过测试结果得到加入硬件辅助虚拟化的虚拟机在各方面性能上相比原始运行在真实硬件上的系统下降了多少。

#### 3 测试结果

原始 Linux 系统测试结果如图 6-2（BYTE UNIX Benchmarks (Version 4.1.0)）。

TEST	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	35221972.5	3018.2
Double-Precision Whetstone	55.0	4353.1	791.5
ExecI Throughput	43.0	5909.4	1374.3
File Copy 1024 bufsize 2000 maxblocks	3960.0	1106314.0	2793.7
File Copy 256 bufsize 500 maxblocks	1655.0	333205.0	2013.3
File Copy 4096 bufsize 8000 maxblocks	5800.0	2125473.0	3664.6
Pipe Throughput	12440.0	2917789.6	2345.5
Pipe-based Context Switching	4000.0	261479.6	653.7
Process Creation	126.0	19325.5	1533.8
Shell Scripts (8 concurrent)	6.0	4613.0	7688.3
System Call Overhead	15000.0	4661777.2	3107.9
=====			
FINAL SCORE			2115.7

图6-2 单纯Linux系统测试结果截图

同时运行两个虚拟机情况下各个操作系统结果

TEST	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	35622326.6	3052.5
Double-Precision Whetstone	55.0	4308.6	783.4
ExecI Throughput	43.0	4712.1	1095.8
File Copy 1024 bufsize 2000 maxblocks	3960.0	1024757.0	2587.8
File Copy 256 bufsize 500 maxblocks	1655.0	327631.0	1979.6
File Copy 4096 bufsize 8000 maxblocks	5800.0	2064989.0	3560.3
Pipe Throughput	12440.0	2972321.1	2389.3
Pipe-based Context Switching	4000.0	478878.7	1197.2
Process Creation	126.0	16359.4	1298.4
Shell Scripts (8 concurrent)	6.0	1215.0	2025.0
System Call Overhead	15000.0	4878720.5	3252.5
=====			
FINAL SCORE			1900.6

图 6-3 虚拟客户机 1 内部测试截图

TEST	BASELINE	RESULT	INDEX
Dhrystone 2 using register variables	116700.0	34779363.0	2980.2
Double-Precision Whetstone	55.0	4173.7	758.9
Exec1 Throughput	43.0	4580.6	1065.3
File Copy 1024 bufsize 2000 maxblocks	3960.0	902083.0	2278.0
File Copy 256 bufsize 500 maxblocks	1655.0	330706.0	1998.2
File Copy 4096 bufsize 8000 maxblocks	5800.0	1663555.0	2868.2
Pipe Throughput	12440.0	2966854.8	2384.9
Pipe-based Context Switching	4000.0	468859.9	1172.1
Process Creation	126.0	15876.7	1260.1
Shell Scripts (8 concurrent)	6.0	1077.3	1795.5
System Call Overhead	15000.0	4679668.6	3119.8
=====			
FINAL SCORE			1794.3

图 6-4 虚拟客户机 2 内部测试截图

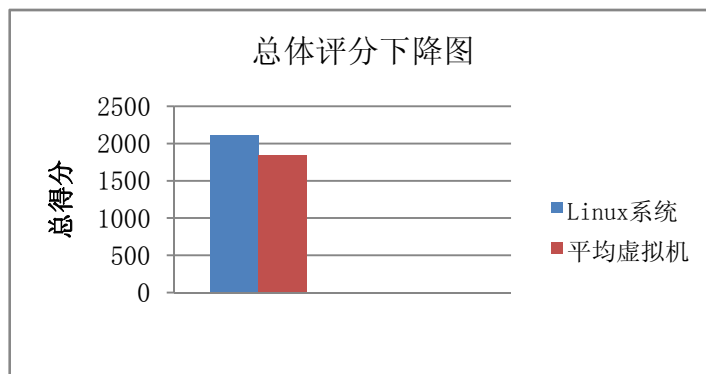


图6-5 性能总分下降直观图

从这些图可以说明加入Intel VT硬件辅助虚拟化技术的虚拟机在性能上比原始系统在性能上下降非常小。当虚拟机数量不断增多时，这个总分下降成线性关系而且损失非常小。

## 6.2 设备管理器虚拟功能测试

### 1. 测试条件

在服务器上面运行经过修改后的 Linux 内核系统，然后安装经过修改的 QEMU-KVM 虚拟机，部署虚拟机管理系统。然后进行多组测试实验，在服务器外部发送控制消息到服务器上运行的多个虚拟机。

### 2. 测试目的

验证设备管理器的虚拟化是否高效正常的运行，是否能将不同的信息发送到指定的虚拟机。

### 3. 测试结果

表 6-4 测试表格

测试项	测试结果
通过外部控制信息启动多台虚拟机	对应的虚拟机客户机在 Linux 系统上面以后台形式启动，运行正常。
通过外部控制信息关闭指定虚拟机	指定虚拟机客户机进程被杀掉，其他虚拟机正常运行。
通过外部控制信息发送 1000 消息	客户机内部消息接收率 100%，没有丢失消息。
在客户机内向外部发送 1000 消息	外部控制器接收率 100%，没有消息丢失。
接收消息中断和轮询比	977/23，1000 条消息中，有 977 条消息及时通过中断来接收，剩余 23 条消息中断未及时，通过通断超时后轮训的方式进行读取。

管理系统用户主界面截图如 6-6 所示。

```

/*****\
|
|               XXX virtual machine console
|
|               S:{ID} Start XXX {ID}
|               K:{ID} Kill XXX {ID}
|               E:{ID} Enter XXX {ID}
|               ... ..
|               ... ..
|               D:      Show running XXX
|               Q:      Quit console
|
|*****\
XXXVMM:

```

图6-6 虚拟机管理系统用户界面图

通过选择启动命令，或者接收到总线上来的启动消息，对应的虚拟机将会开启，启动的界面如图6-7所示。

```

LOAD LIST LOADED FROM DRIVE U0
MODULE LOADING STARTS
LOADING MODULES FROM LOCAL DISKS
UTE UTE TOP DYL HAP POH LG4 LOE LOE TOH HAP WYN BIB TUK ETH ETH TOM EPI UXM
UX2 UAF UXD TOE SHI POZ KER JUP INV ETO CEP BLT BLN SEK NPZ AIH AIM POH EKL
POF POZ SYH NAS HMR RUB PUL SQQ SQP POM POX PBH GOG FIS FIZ NPE NP3 TOV HME
BLS RA2 RAU ESZ POD HMI PGE PEM PAG OSI LOG LIB ES3 ISR STQ PRS ENA DXP DMX
DEB CLU BUR BOX BOS BEO APP
ALL MODULES LOADED

DEBUG MODE = 00000000

BOLERO RUN TIME: 00:03
LOADING TIME:    00:02

STARTING OS ...

DEBUGGER READY
STARTING FAMILIES    BIB HMR FIZ NPZ SYH NAS UXM ESZ UXDLIB: PIU type not support
ted 19dWYN AIM

```

图 6-7 客户机启动串口显示截图

通过对设备虚拟化的测试，可以得到设备管理器虚拟化功能基本实现。在每

个客户机内都能够看到虚拟的设备管理器，并且所有的 IO 和中断都能正确被虚拟，虚拟客户机内部和总线上的消息能够正常通讯。多个虚拟机能够通过外部控制进行开关。只是部分中断可能由于虚拟化过程中性能损耗或者设备本身问题不能够百分之百得到，但是通过中断超时加轮询的方式能够实现所有消息的读取。

## 6.3 时钟准确度测试

### 1. 测试条件

1) 针对 8254 始终测试，将虚拟客户机 Linux 内核重新编译，将 SMP 和 LAPIC 的选项去掉，使得系统使用 8254 作为系统的默认时钟中断。

2) 同时在 KVM 模块中加入输出，并且监控统计积累的时钟中断次数。

3) 在客户机中通过 RDTSC 指令读取相同时间内，系统时钟中断变化次数，通过多次取平均值。

### 2. 测试目的

通过计算相同真实时间内各个虚拟机内部时钟中断变化，得到时钟丢失程度，再通过比较优化前和优化后的丢失程度比来测试精度是否得到提高。

### 3. 测试结果

通过脚本监控虚拟机内的中断次数的变化可以看到，经过改进后的时钟虚拟化设备的数据输出如图 6-8 所示。

在图中，本课题将客户机系统运行高负载程序，系统 CPU 如果呈现基本满负荷运转，在这种情况下测试时钟中断精准度问题。其中所有的中断次数增加和注入中断基本一致，累积中断处于 1-2 之间，基本不丢失时钟中断。因为每次剩余中断都及时的进行了补注入。

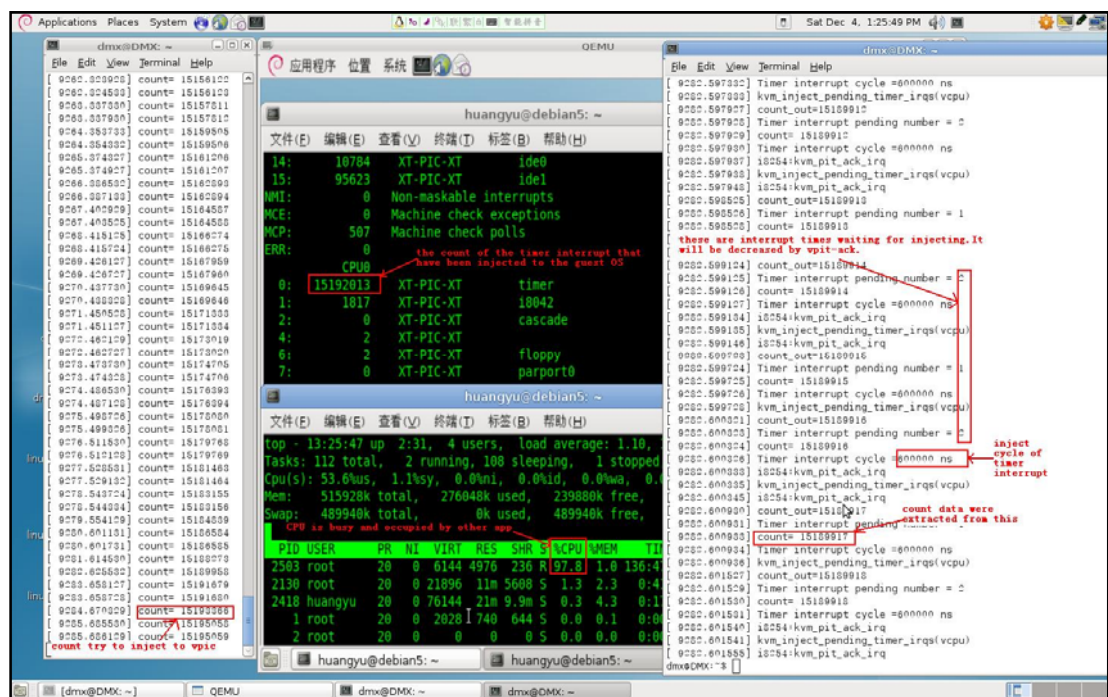


图6-8 时钟中断累积次数图

通过 RDTSC 在相同真实时间内客户机内部时钟变化改变：

表 6-5 改进前后时钟精度准确测量结果表

系统	时钟频率	真实时间（ms）	中断增加（1000 次平均值）
改进前	100	10000	978
改进后	100	10000	1003

从表中我们可以看出，在 100 频率下的系统中，经过精确的 10000 毫秒内统计得出分别在改进前和改进后的时钟中断次数更加接近真实的 1000 次，由于 RDTSC 可能得到过程中有些误差造成改进后测得中断测试反而超过了 1000，但是整体精确度提高了将近 2%-3%，使得系统更加稳定。

## 6.4 指定 PCI 硬件设备 IO 地址测试

### 1. 测试条件

假设需要修改的设备 IO 访问起始地址是 0x2000，通过修改 QEMU 中的 PCI IO 地址分配代码，然后运行 Linux 客户机镜像，检测虚拟设备 IO 地址是否达到预定的 0x2000。

### 2. 测试目的

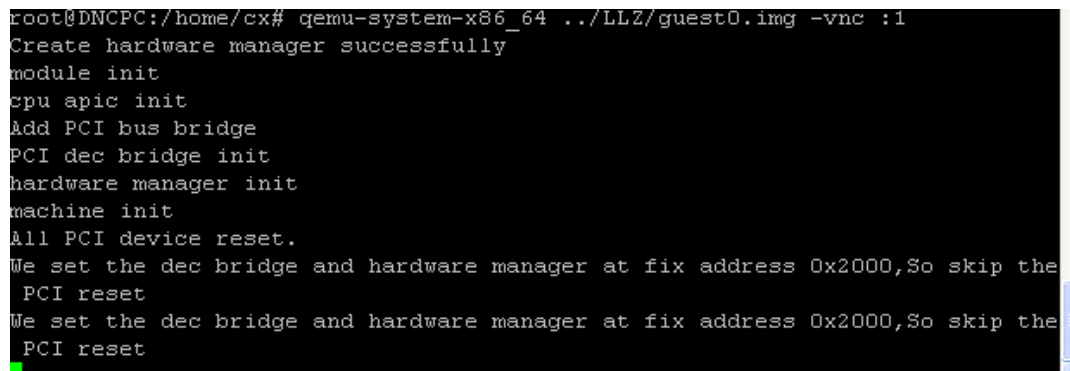
通过测试证明固定 IO 访问起始地址是否成功。

### 3. 测试结果

通过修改后的 IO 访问起始地址和没有修改的发生了改变，而且分配到了指

定的 0x2000。

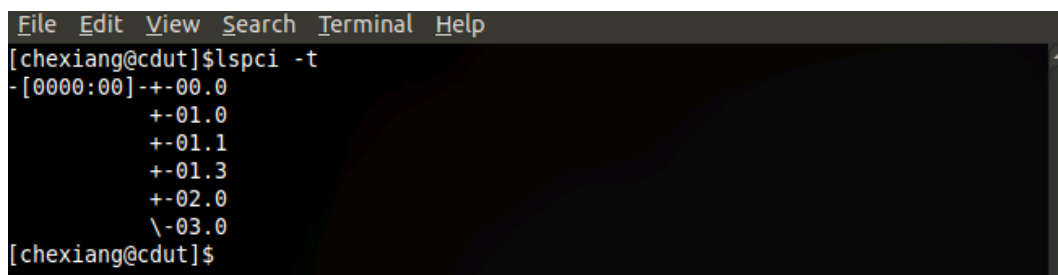
(1) 测试环境里虚拟机启动后如图 6-9 里所示。



```
root@DNCP: /home/cx# qemu-system-x86_64 ../LL2/guest0.img -vnc :1
Create hardware manager successfully
module init
cpu apic init
Add PCI bus bridge
PCI dec bridge init
hardware manager init
machine init
All PCI device reset.
We set the dec bridge and hardware manager at fix address 0x2000, So skip the
PCI reset
We set the dec bridge and hardware manager at fix address 0x2000, So skip the
PCI reset
```

图 6-9 启动界面截图

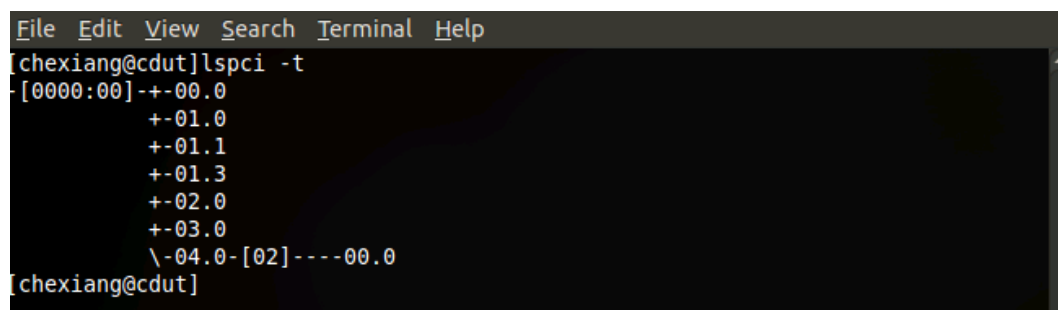
(2) 设备添加前的总线结构图如图 6-10 里所示。



```
File Edit View Search Terminal Help
[chexiang@cdut]$ lspci -t
-[0000:00]--+-00.0
          +-01.0
          +-01.1
          +-01.3
          +-02.0
          \-03.0
[chexiang@cdut]$
```

图 6-10 VNC 连接后原始总线结构截图

(3) 设备添加以后的总线结构图如图 6-11 里所示。



```
File Edit View Search Terminal Help
[chexiang@cdut]$ lspci -t
-[0000:00]--+-00.0
          +-01.0
          +-01.1
          +-01.3
          +-02.0
          +-03.0
          \-04.0-[02]---00.0
[chexiang@cdut]
```

图 6-11 VNC 连接后修改总线后结构截图

从上图比较结果可以看出，虚拟以后多出了一条 PCI 总线节点，这个就是前面需求中需要构造的虚拟 PCI 总线。

(4) 设备设置以后查看设备内存数据如图 6-12 所示。



```
00:04.0 PCI bridge: Digital Equipment Corporation DECchip 21154
00: 11 10 26 00 07 00 a0 00 00 00 04 06 00 00 01 00
10: 00 00 00 00 00 00 00 00 00 02 02 00 d0 f0 a0 00
20: f0 ff 00 00 f0 ff 00 00 00 00 00 00 00 00 00 00
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00

02:00.0 Bridge: Adlink Technology Device 13b8 (rev 02)
00: 4a 14 b8 13 03 00 00 00 02 00 80 06 00 00 80 00
10: 00 00 00 00 01 20 00 00 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 f4 1a 00 11
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

图 6-12 PCI 设备内存结构截图

(5) 设备虚拟后设备详细信息图如图 6-13 所示。

```
[chexiang@cdut]lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.3 Non-VGA unclassified device: Intel Corporation 82371AB/EB/MB PIIX4 ACPI
(rev 03)
00:02.0 VGA compatible controller: Cirrus Logic GD 5446
00:03.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL-8139/8139C/8139
C+ (rev 20)
00:04.0 PCI bridge: Digital Equipment Corporation DECchip 21154
02:00.0 Bridge: Adlink Technology Device 13b8 (rev 02)
```

从上面截图可以看出虚拟出来了 PCI 桥结构和设备管理器结构。

图 6-13 设备结构详细信息图

(6) 指定 I/O 后的地址显示如图 6-14 所示。

```
File Edit View Search Terminal Help
03f6-03f6 : ata_piix
03f7-03f7 : floppy
03f8-03ff : serial
0cf8-0cff : PCI conf1
2000-2fff : PCI Bus 0000:01
2000-27ff : 0000:05:00.0
afe0-afe3 : ACPI GPE0_BLK
b000-b03f : 0000:00:01.3
b000-b003 : ACPI PM1a_EVT_BLK
b004-b005 : ACPI PM1a_CNT_BLK
```

图 6-14 IO 地址分配图

从图 6-14 可以看出，桥和设备管理器地址都虚拟 I/O 正确并且设置到了当初预定的 0x2000 上。

通过上面的实验结果得出，本课题成功的进行了总线结构的虚拟，以及虚拟设备 I/O 起始地址的指定。

## 本章小结

本章主要对前文的各种分析研究做了统一的测试和数据分析。首先对 QEMU-KVM 虚拟机的性能进行测试，然后对虚拟的设备管理器进行功能和性能上

进行测试,接着分别对时钟虚拟化的 PCI 设备虚拟化中 I/O 基地址固定功能进行系统的测试,最后得出测试结果,结果达到预期。



## 结论与展望

通过本文的研究，对 Intel VT-X 技术以及传统虚拟化技术有了深层次的认识。通过性能上的测试证明了硬件辅助虚拟化在虚拟机中起到的重要作用。在加入硬件辅助虚拟化的虚拟机比纯软件实现的虚拟机性能上超过将近 10 倍。而加入硬件辅助虚拟化的虚拟机在 CPU 处理，内存读写等方面相比自然系统只损耗了很小一部分，几乎接近自然系统。

本文依靠 QEMU-KVM 虚拟机为基础，针对设备虚拟化部分，从总线虚拟到 I/O 和中断虚拟，逐层研究和改进虚拟化过程中的各个环节，并且通过测试基本得到了预期的效果，并且总结了以下几点创新：

1. 提出了一个时钟设备的虚拟化改进方案。

本文通过对中断和时钟虚拟化的研究，借助 Intel VT 技术通过保存-再触发方式解决时钟中断的注入丢失问题，并且通过批量注入方式来提高实现时钟中断的注入效率，减小系统损耗。

2. 提出了一个固定虚拟 PCI 设备 I/O 基地址的改进方法。

本文通过对虚拟 PCI 设备地址初始化进行拦截和修改，实现给指定的设备固定分配 I/O 基地址。

基于以上的创新点研究以及虚拟化技术的基础研究，本文取得了以下研究成果。

1. 实现了一个硬件管理器的虚拟化。

本文针对 Intel VT 硬件辅助虚拟化技术，将 QEMU 中的硬件模拟部分转移到 KVM 中实现，完成了一个硬件管理器的虚拟化实例，同时提出了一个完整的设备虚拟化应用解决方案。

2. 完成了系统时钟和基于 VT-x 的虚拟中断研究。

提出了一个在虚拟化过程中减少虚拟时钟丢失的改进方法。通过将虚拟时钟丢失中断进行保存和再触发以及虚拟中断批量注入来解决时钟丢失问题。

3. 完成了对 QEMU-KVM 中设备 I/O 虚拟化的研究。

本文一方面对 QEMU 中的虚拟设备模拟进行研究，同时另一方面对 KVM 中的设备注册和 I/O 拦截进行研究，通过两个方面的结合，实现了将设备虚拟化 I/O 部分从 QEMU 中转移到 KVM。

总之本文主要是为探索现代开源虚拟机贡献了一部分力量，对硬件辅助虚拟化技术进行了很好的诠释，特别是在完全虚拟化这个比较复杂的领域，提出了一些自己的思想和见解。目前虚拟化技术趋于多样化，而且很多新技术比如云存储都是基于完全虚拟化技术的，这种易于移植和应用的系统虚拟化技术将在计算机领域发挥越来越重大的作用。

## 致 谢

在论文完成之际，谨向我的导师王华军教授表示由衷的感谢！在成都理工大学攻读硕士学位的三年时间里，在学习上、生活上王老师和苗老师给了我极大的关怀和帮助。在学术上他为我指明方向，对我的研究进行悉心指导，而且通过言传身教，他给了我很多启发。他深厚的学术造诣、严谨的治学态度、高度的责任感给了我很大的影响，这些都将使我终身受益。

感谢师兄王合闯、李英豪、黄伟、卢涵宇、冷小鹏、王淼、程文波、李宽，师姐郭曦容、刘婷对我的指引和帮助。

感谢同门好友徐文、朱圣才、章伟、付军安、赵新、颜亮、李娟、刘慧芳在学校时给我的悉心指导和关怀，和你们在一起的经历和感受，是我人生中的一笔巨大财富。

感谢信息工程学院和研究生院的领导与老师们，在学习期间他们给了我无私的帮助与支持，特别是罗省贤老师在项目上对我的帮助和指点。

特别要提出感谢的是由王华军老师带领的虚拟机开发课题团队的所有参与成员，在整个课题的研发中，团队成员互相激励，共同学习，相互配合，攻克难关，让我学到了很多理论知识与团队合作。团队中尤其感谢的是黄煜、林龙增同学在整个课题的研究过程中，默默付出，踏踏实实的工作，为整个项目推进起到了至关重要的作用。最后在此向他表示深深的感谢。

最后衷心感谢我的家人，二十余年来，他们竭尽全力地支持与关心着我的学习和生活，没有他们的辛苦付出，就没有我今天的成绩。

感谢所有关心和帮助我的人们，我深深祝福你们！

## 参考文献

- [1] 虚拟化技术基础 [http://blog.chinaunix.net/u3/110913/showart\\_2185421.html](http://blog.chinaunix.net/u3/110913/showart_2185421.html)
- [2] 探索 Linux 内核虚拟机 <http://www.ibm.com/developerworks/cn/Linux/l-Linux-kvm/>
- [3] 英特尔开源软件技术中心 复旦大学并行处理研究所著 系统虚拟化 清华大学出版社.2009
- [4] 徐磊 Linux 系统下 C 程序开发详解 电子工业出版社.2008
- [5] 纪纯杰 贺晓能 Linux 内核分析及常见问题解答 著人民邮电出版社 2000
- [6] Naba Barkakati Red Hat LINUX 核心技术精解 北京: 中国水利水电出版社.1999
- [7] 李善平 李文峰等 著 Linux 内核 2.4 版源代码分析大全 机械工业出版社.2002
- [8] BOVET&CESATI 著 深入理解 Linux 内核中国电力出版社.2010
- [9] KVM 开源项目 官方网址 [http://www.Linux-kvm.org/page/Main\\_Page](http://www.Linux-kvm.org/page/Main_Page)
- [10] QEMU 开源项目 官方网址 [http://wiki.QEMU.org/Main\\_Page](http://wiki.QEMU.org/Main_Page)
- [11] The Intel® 64 and IA-32 Architectures Software Developer's Manual.2009
- [12] Claudia Salzberg Rodriguez 等 Linux 内核编程 机械工业出版社.2006
- [13] 吴国伟 李张等 Linux 内核分析及高级编程 电子工业出版社.2008
- [14] 陈莉君 深入分析 Linux 内核源码 <http://www.kerneltravel.net/>
- [15] 鲁松主编 计算机虚拟化技术及应用 机械工业出版社.2008
- [16] Mel Gorman. 深入理解 Linux 虚拟内存管理[M]. 北京: 航空航天大学出版社.2006
- [17] 纪纯杰 贺晓能 Linux 内核分析及常见问题解答.2000
- [18] 虚拟机中 Guest OS 时钟 (TIMEKEEP) 问题探讨 <http://www.ibm.com/developerworks/cn/Linux/l-Linux-kvm/>
- [19] Uresh Vahalia .UNIX Internals:The New Frontiers . Pearson.2003
- [20] Robin JS,Irvine CE.Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor,Proc.9th USENIX Security Symposium,2000
- [21] 李庆华 罗犀劲 基于检查点进程迁移机制的改进[J].计算机仿真, 2003,20(5): 50-52
- [22] 周全 卢显良 任立勇等. 基于 Linux 的进程迁移机制设计[J]. 计算机应用, 2003,23(8): 58-60
- [23] 吴松 金海. 存储虚拟化研究[J]. 小型微型计算机系统, 2003,24(4): 728-732
- [24] 虚拟化 IT 效率最大化[J]. 信息系统工程, 2007(3)
- [25] 王迪. 分析: 虚拟化技术对企业应用带来的优势[EB/OL]. <http://tech.sina.com.cn/smb/2008-11-26/0615888394.shtml>,2008-11-26
- [26] 龚哎斐, 张文静. 基于虚拟化架构的软件开发与测试环境自动化[J]. 自动化与信息工程,2008(2)
- [27] 张萧, 祝明发, 肖利民. 分布式 I/O 资源虚拟化技术的研究[J]. 微电子与计算机, 2008,25(100)
- [28] Martin F. Maldonado. 虚拟化概述: 模式的观点[EB/OL]. <http://www.ibm.com/developerworks/cn/grid/grvirt/>
- [29] IBM.. Using the Virtual I/O Server, 2007,9
- [30] User-mode Linux. <http://user-mode-linux.sourceforge.net/>
- [31] Aloni D. Cooperative linux. Proceedings of the Linux Symposium. 1: 23-32,2004
- [32] Linux\_VServer. <http://linux-vserver.org/>

- [33] Kivity A, Kamay Y, Laor D, Lublin U, Liguori A. kvm: the Linux Virtual machine monitor, Proceedings of the 2007 Ottawa Linux Symposium, 2007
- [34] R. Iguest: Implementing the little Linux hypervisor. Proceedings of the 2007 Ottawa Linux Symposium, 2007
- JS, Irvine CE. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, Proc. 9th USENIX Security Symposium, 2000
- [35] 车翔 blog.csdn.com/yeart520
- [36] 宋宝华. 设备驱动开发详解[M]. 北京: 人民邮电出版社. 2009.
- [37] (美) 吕斯特 等著, 陈奋 译. 虚拟化技术指南[M]. 北京: 机械工业出版社, 2011.
- [38] 于淼, 戚正伟 等编著. NewBluePill: 深入理解硬件虚拟机[M], 北京: 清华大学出版社, 2011.
- [39] 郝旭东. Intel VT-d 技术的研究及其在 KVM 虚拟机上的实现[D]. 成都: 电子科技大学, 2009, 6.
- [40] 林昆. 基于 Intel VT-d 技术的虚拟机安全隔离研究[D]. 上海: 上海交通大学, 2011, 1.
- [41] (美) Robert Love 著, 陈莉君, 康华, 张波 译. Linux 内核设计与实现(第 2 版)[M], 北京: 机械工业出版社, 2010.
- [42] 李胜召, 郝沁汾, 肖利民. KVM 虚拟机分析[J]. 计算机工程与科学, 2008, 30(A1): 129-132.
- [43] Intel. Intel Virtualization Technology for Directed I/O. Intel Corporation, 2006.
- [44] Avi Kivity, Yan Kamay Dor Laor KVM: the Linux Virtual Machine Monitor[C] proceedings of Linux Symposium, 2007. Ottawa, Canach.
- [45] 费春. 基于 ST5516 芯片的 KVM 虚拟机移植[J]. 中国有线电视, 2006, 12: 1171-1173.
- [46] 刘锋. Kernel-based Virtual Machine 研究与其事件跟踪机制实现[D]. 成都: 电子科技大学, 2009, 5.
- [47] 宗红红. KVM 在嵌入式系统上的移植研究[D]. 南京: 南京理工大学, 2008, 6.
- [48] 张扬. XEN 下基于 Intel VT-d 技术的 I/O 虚拟化的实现[D]. 成都: 电子科技大学, 2010, 5.
- [49] 张鑫. 基于 Intel VT-d 在安腾平台的高效虚拟 IO 模型的实现与研究[D]. 成都: 电子科技大学, 2008, 5.
- [50] 催泽永, 赵会群. 基于 KVM 的虚拟化研究及应用[J]. 计算机技术与发展, 2011, 21(6): 108-115
- [51] 时卫东. 基于内核的虚拟机的研究[D]. 吉林: 吉林大学, 2011, 4.
- [52] 李永达. 虚拟机应用系统的设计与实现[D]. 西安: 西安电子科技大学, 2010, 3.
- [53] 张彬彬, 汪小林, 杨亮, 赖荣风, 王振林, 罗英伟, 李晓明. 修改客户操作系统优化 KVM 虚拟机的 I/O 性能[J]. 计算机学报, 2010, 33(12): 2312-2320.
- [54] 严东华, 张凯. Java 虚拟机及其移植[J]. 北京理工大学学报, 2002, 22(1): 64-67.
- [55] 高清华. 基于 Intel VT 技术的虚拟化系统性能测试研究[D]. 杭州: 浙江大学, 2008, 5.
- [56] KVM 开源项目. 官方网址 [http://www.Linux-kvm.org/page/Main\\_Page](http://www.Linux-kvm.org/page/Main_Page).
- [57] QEMU 开源项目. 官方网址 [http://wiki.QEMU.org/Main\\_Page](http://wiki.QEMU.org/Main_Page).
- [58] 鲁松. 计算机虚拟化技术及应用[M]. 北京: 机械工业出版社. 2008.
- [59] 李允, 罗蕾, 雷昊峰, 熊光泽. 嵌入式 Java 虚拟机的性能优化技术[J]. 计算机工程, 2004, 30(18): 46-48.
- [60] 姚远. 虚拟机高校设备访问模型设计与实现[D]. 国防科技大学, 2010, 11.
- [61] 陈诚. 虚拟化系统在桌面应用中的一种新型应用模式[D]. 上海: 复旦大学, 2010, 4.
- [62] 探索 Linux 内核虚拟机 <http://www.ibm.com/developerworks/cn/Linux/l-Linux-kvm/>.
- [63] 周显军, 李众立, 张俊然. 基于 S3C4510B 芯片 KVM 虚拟机的移植和测试[J]. 微计算机信息. 2007, 29: 127-128.

- [64] Qumranet Inc.KVM:Kernel-based Virtual Machine. <http://kvm.sourceforge.net/>.
- [65] Darren Abramson, Jeff Jackson. Intel Virtualization Technology for Directed I/O [J]. Intel Technology Journal Volume 10.
- [66] 刘学勇,陈建伟.精通 Linux C 编程[M].北京:清华大学出版社.2008.
- [67] 金海 等著.计算系统虚拟化----原理与应用[M].北京:清华大学出版社,2008.
- [68] Qumranet Inc.KVM:Kernel-based Virtualizationg Driver White Paper.Qumranet Inc,2006.

## 攻读学位期间发表的学术论文

- [1] 车翔, 王华军.. QEMU-KVM 虚拟 PCI 设备优化方法, 电脑与电信, 20011,11
- [2] 车翔, 王华军.. Research of KVM Interrupted and Virtual Clock, ICOEIS, 20011,11