

SHANGHAI JIAO TONG UNIVERSITY

**学士学位论文**

## BACHELOR’S THESIS



论文题目：车载虚拟化选型与性能测试分析

学生姓名: 陈江涛

学生学号: 516030910157

专 业: 软件工程

指导教师: 戚正伟

学院(系): 电子信息与电气工程学院

车载虚拟化选型与性能测试分析

摘要

30年前，软件首次部署到汽车上，用于控制发动机, 软件只在大约4代汽车中成长。从一代到下一代，软件数量增长了10倍，甚至更多。今天，在高级车中的车载系统软件超过一千万行代码，有2000多个独立功能由软件实现或控制，下一代车载系统软件代码可能会多出十倍。汽车智能化的发展使得控制软件复杂度和集成度不断提升，hypervisor的虚拟化方案可以.在多.核异构的单芯片上运行多个不同类型的操作系统，各个系统之间共享硬件资源，互相独立又可以相互通信，提高了.硬件效率，大幅度降低成本。更重要的是，虚拟.化所具备的不.同操作系统之间的间隔能力，可以提升系统的可靠性和安全性。虚拟化技术已经成为下一代智能网联汽车系统平台软件的关键技术之一。由于ARM芯片的高效能特点，. ARM芯片是移.动和嵌入式领域的主.要选择，ARM芯片也是是汽车平台的主流芯片平台。本课题研究基于ARM64-v8架构的硬件平台和开源hypervisor方案，尝试构建虚拟化平台，综合考虑实时响应和性能开销，在其上运行两个及以上的异构操作系统，为今后车载虚拟化平台产业化做好.前期预研工作。本文主要根.据车载系统的特点及需求，深入了解分析现在支持ARM64-v8架构的开源hypervisor，如xen、kvm、Xvisor、jailhouse等，初步筛选出其中适合车载系统的hypervisor，选用合适的benchmark对其进行测试，比对其性能开销。

关键词：车载虚拟化，ARM虚拟化，Xen，jailhouse，嵌入式虚拟化，性能测试.

**SELECTION OF VIRTUALIZATION HYPERVISOR IN AUTOMOTIVE AND PERFORMANCE ANALYSIS**

**ABSTRACT**

Thirty years ago, software was first deployed to cars to control their engines, and it only grew in about four generations of cars. From one generation to the next, the number of software has increased tenfold, or even more. Today, there are more than 10 million lines of code in the on-board system software of advanced vehicles. There are more than 2000 independent functions implemented or controlled by the software. The software code of the next generation of on-board system may be ten times more. With the development of automobile intelligence, the complexity .and integration of control software are constantly improved. The scheme of virtualization can run multiple different types of operating system on a single chip with heterogeneous multi-core. Each system shares hardware resources, which is independent of each other and can communicate with each other, which improves the hardware efficiency and greatly reduces the cost. More importantly, virtualization has the ability to separate different operating systems, which can improve the reliability and security of the system. Virtualization technology has become one of the key technologies of the next generation of intelligent networked vehicle system platform software. Due to the high efficiency of ARM chip, ARM chip is the main choice in the field of mobile and embedded. ARM chip is also the mainstream chip platform of automobile platform. This paper studies the hardware platform and open-source hypervisor based on arm64-v8 architecture, trying to build a virtualization platform, considering the real-time response and performance cost, running two or more heterogeneous operating systems on it, and doing the preliminary research work for the industrialization of vehicle virtualization platform in the future. According to the characteristics and requirements of the automobile system, this paper deeply analyzes the open-source hypervisors that support the arm64-v8 architecture, such as Xen, KVM, Xvisor, jailhouse, etc., preliminarily selects the suitable hypervisors for the automobile system, selects the appropriate benchmark to test them, and compares their performance costs.

**Key words:** Automotive hypervisor, ARM virtualization, xen, jailhouse, Embedded virtualization, performance analysis

第一章 绪论

1.1课题意义与背景

随着物联网、5G、人工智能等技术的发展，传统的汽车产业正快速的转向智能化、电动化、网联化。因此也涌现出了一大批的新能源汽车、智能汽车初创企业，比如特斯拉、蔚来汽车、小鹏汽车、威马等等企业。与此同时，大众、宝马、通用等众多的传统车企也开始快速跟进。在汽车智能化、电动化、网联化的此趋势之下，汽车的电子成本和软件成本占比正在快速提升。汽车中的软件数量呈指数级增长。这种发展的驱动力是更便宜、更强大的硬件和对新功能创新的需求。软件和基于软件的功能的快速增长给汽车行业带来了各种挑战，包括其组织、关键能力、过程、方法、工具、模型、产品结构、分工、物流、维护和长期战略。从软件工程的角度来看，汽车工业是先进技术的理想的应用领域。虽然汽车行业可能采用从软件工程领域获得的其他领域知识的一般结果和解决方案，但汽车行业的特定约束和特定领域的要求要求提供单独的解决方案，并给汽车软件工程带来各种挑战。30年前，软件首次部署到汽车上，用于控制发动机，特别是点火装置。第一个基于软件的解决方案是非常局部的、孤立的和不相关的。硬件/软件系统在自下而上地发展。这就决定了汽车的基本结构，包括用于不同任务的专用控制器（电子控制单元或ecu）以及专用传感器和执行器。随着时间的推移，为了优化布线，总线系统被部署到车内，通过车内ecu与传感器和执行器连接。有了这样的基础设施，ecu也连接起来，可以交换信息。因此，汽车工业开始引入通过总线系统连接的多个ecu上实现的功能。第一款软件是在大约30年前才进入汽车领域的，所以软件只在大约4代汽车中成长。从一代到下一代，软件数量增长了10倍，甚至更多。今天，在高级车中的车载系统软件超过一千万行代码，有2000多个独立功能由软件实现或控制[2]，下一代车载系统软件代码可能会多出十倍。汽车上的许多创新功能都是由软件实现和驱动的。最近的问题是能源管理和目前进入混合解决方案的步骤，这只能通过大量的软件以经济的方式实现。它主要是应用领域特有的创新，具有更强的依赖性和特性交互，要求跨应用领域组织。软件/硬件系统开发成本的50-70%是软件成本。硬件越来越成为一种商品（从ecu的价格下降可以看出），而软件决定了功能，因此成为主导因素。

传统上汽车工业是高度垂直组织的。在软件工程中，我们会说它是模块化的。机械工程师们努力工作了100多年，使汽车上的各种子系统在开发和生产中相当独立。这有助于子部件的独立开发和生产，并允许巨大的分工。因此，供应商可以通过随后的外包来接管工程、开发和生产的相当一部分。理想情况下，汽车零件由一系列供应商生产，或多或少只能由汽车制造商组装（以下称为OEM）。因此，大量的工程和生产外包，可以优化成本和风险分配。汽车被（或更好的是）认为是由原始设备制造商组装的子零件套件。随着软件成为创新的主要力量，情况发生了巨大变化：

•传统上完全不相关且独立的功能（如制动、转向或控制发动机）由驾驶员自由控制。这些功能相互关联并开始相互作用。汽车从一个组装好的装置变成一个集成系统。像无意的特征交互这样的现象成为问题。组装子部件成为系统集成。

•汽车的行为变得更加可编程。某些特性，如舒适性或运动操控性，不再仅仅取决于机械性能，还取决于软件。

•汽车成本越来越受到软件开发成本的影响，传统的以按部件成本模式为主的成本模型已不再完全有效。

汽车嵌入式软硬件系统的规模和结构是巨大的。应用软件建立在实时操作系统和总线驱动程序之上。大多数软件是硬实时关键的，或者至少是软实时关键的。对汽车软件系统的要求相当具体：

•广泛的不同用户（司机和乘客，以及维护）

•具体维护情况

•安全关键功能

•系统运行的特定环境

•功能的多样性（从嵌入式实时控制到信息娱乐，从舒适功能，如空调到驾驶员辅助，从能量管理到软件下载（闪存）功能，从安全气囊到车载诊断和错误记录）。

因此，车载软件的复杂度和需求范围是巨大的。相反，汽车软件和功能的增加并没有接近尾声。我们可以预期未来会有实质性的增长。未来的发展受到以下趋势的推动：

•对新的创新或改进功能的高需求

•快速变化的平台和系统基础设施

•尽管成本压力很大，但开发成本迅速增加

•对更高质量和可靠性的需求

•上市时间更短

•增加个性化

因此，对软件和系统工程的专门研究提出了很高的要求。

在具有高度复杂的功能和软件体系结构、广泛的资源共享、混合的关键性需求和遗留软件组件的集成的应用领域中，虚拟化是管理复杂性以及有效、安全和可靠地共享大量资源的关键技术。传统的虚拟化主要是将物理通用计算资源优化分配到一组虚拟机上，并将这些资源作为虚拟cpu。虚拟机可显示访问对其分配的虚拟CPU，并且任何访问控制都由虚拟化层处理，例如虚拟机监控程序或虚拟机监视器（VMM）。除了访问的时间协调之外，虚拟化层还可以确保每个虚拟机只访问其分配的CPU和内存资源。

汽车智能化的发展使得控制软件的复杂度和集成度不断提升，Hypervisor 的虚拟化方案可以在多核异构的单芯片上运行多个不同类型的操作系统，各系统间共享硬件资源，即是彼此独立又可交互信息。Hypervisor 即满足了日益复杂场景下的不同业务需求，又提高了硬件资源的使用效率，大幅降低了成本，更重要的是，虚拟化所具备的不同操作系统间的隔离能力，可以大大提升系统的可靠性和安全性。虚拟化技术已经成为下一代智能网联汽车系统平台软件的关键技术之一。

1.2相关工作

在嵌入式虚拟化领域，现有的商业和开源产品，他们都是Type-I的，而且支持实时调度，很好的支持了汽车软件的需求，在安全方面也有所保障。黑莓软件系统公司2015年推出QNX Hypervior1.0，Type-I型的虚拟机平台，适用于工业自动化、医疗设备系统以及汽车应用程序，例如车载娱乐系统、数字仪表盘和驾驶员辅助系统。通过部署QNX Hypervisor，嵌入式系统开发人员可以把多个操作系统集成到单一的计算平台或片上系统(SoC)，从而降低产品的尺寸、功耗、重量和成本。QNX Hypervisor使得开发人员可以隔离非安全相关的软件组件和安全相关的软件组件，从而简化安全关键型系统的认证流程。与安全相关的软件组件运行在一个操作系统上，而非安全相关的软件组件则运行在另一个操作系统上。

在2018年的Linux嵌入式大会上发布的ACRN[3]，是一款轻量级的、灵活的hypervisor，以关键的安全性和实时性为设计的出发点，并且通过开源平台为精简嵌入式开发进行优化。ACRN 的最大优势之一是尺寸小，发布时大约只有 25K 行代码。ACRN 有两个关键组成：hypervisor 和 ACRN 设备模块。ACRN Hypervisor 是一个 Type 1 的 hypervior，可以直接运行于裸机上。但遗憾的目前不支持ARM，移植到ARM平台上工作量比较大。

值得注意的是，其中Type-I VMM类型的QNX Hypervisor，实际上是由一个RTOS（QNX）加上一个虚拟化模块构成的，这种模式实际上更接近性能开销更大的Type-II VMM。而 Linux嵌入式大会上发布的ACRN Hypervisor，除了它的开源特性，更重要的是它由Intel公司推动开发，文档以及后续的发展都可以得到保障。但是由于商业而非技术的原因，Intel公司不太可能将ACRN移植到ARM架构的平台上，而ARM架构恰恰是汽车平台的主流芯片平台。所以这一工作需要第三方来完成。

1.3本文工作

本课题研究基于 ARM64-v8 架构的硬件平台和开源 Hypervisor 虚拟化方案，尝试构建 Type-1 型的半虚拟化平台，综合考虑实时响应和各方面系统性能，其上运行两个以上的异构操作系统，例如 QNX+Linux+AUTOSAR，并根据需求从ARM现有的开源虚拟化方案中甄选出适合车载应用的方案，对比其性能并进行分析，为今后车载虚拟化平台产业化做好前期技术预研工作。

1.4本章小结

随着物联网、5G、人工智能等技术的发展，传统的汽车产业正快速的转向智能化、电动化、网联化。在汽车智能化、电动化、网联化的此趋势之下，汽车的电子成本和软件成本占比正在快速提升。今天，在高级车中的车载系统软件超过一千万行代码，有2000多个独立功能由软件实现或控制赛迪智库的数据也显示，2005年汽车电子成本占比仅有20%，而现在汽车的电子成本已经提高到了50%以上。而随着汽车上的电子元器件越来越多，智能化程度越来越高，软件也越来越复杂。显然，在汽车的电动化、智能化和网联化趋势之下，软件在汽车当中的重要性和成本占比也将会越来越高，市场空间和价值极大。

汽车内的电子系统日趋复杂，而它们所需要的软件系统及软件所依赖的操作系统的种类和复杂性也在不断提升。另一个趋势是汽车中所使用的电子芯片单核速度也不断提升，逐渐接近服务器或者PC上所使用的芯片。这两个趋势使得虚拟化技术也开始在车载软件领域使用。本文根据车载系统软件的特点和需求，综合考虑系统各方面性能，对比分析现有的开源ARM架构下的虚拟化平台，如：KVM，XEN，Xvisor，Jailhosue等。为今后车载虚拟化平台产业化做好前期技术预研工作。

第二章 技术背景

2.1虚拟化技术[4]

虚拟化技术(Virtualization Technology)无疑是近十几年最热门的计算机系统软件技术之一。虽然它并不是最近新出现的概念，甚至已经有了四十多年的历史，但它的热度昭示着一波新的技术浪潮将会伴随着虚拟化技术席卷计算机系统领域主要组成部分分甚至是各个角落。从云计算到移动，从高性能到嵌入式，从处理器到存储，从显示到无线，虚拟化技术在各个相当成熟的行业领域仍然活跃，与时代热点结合，迸发出了很多全新的解决方案。虚拟化技术未来可能会很深刻地影响计算机行业，甚至颠覆计算机系统的应用模式乃至开发模式。

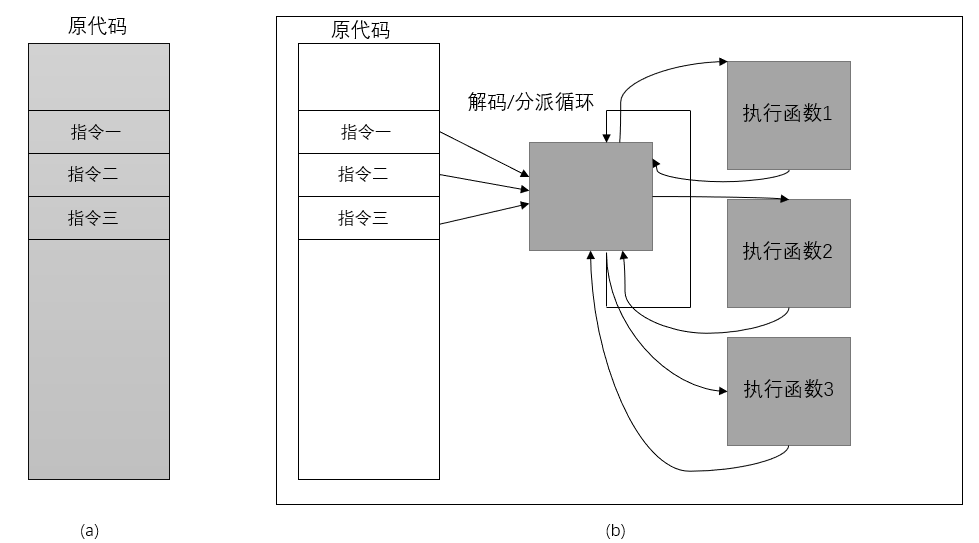
虚拟化技术从概念上看非常相似与仿真技术。在仿真技术中，一个系统“伪装假扮”成另一个系统。而虚拟化技术则是一个系统“假扮成”两个或两个以上的系统。虚拟化的定义很广泛，将任何一种形式的资源抽象成另一种形式的资源的技术都可以称作“虚拟化技术”。在常用的现代操作系统中其实都存在着某种意义上的虚拟化技术，比如说虚拟内存地址和进程概念。如果把真实的物理内存看作是设备资源，那么虚拟内存就是将物理内存虚拟成多个内存空间，每个内存空间都有自己虚拟内存地址，虚拟内存容量可以少于或多于物理内存。另外，进程的概念其实是对物理硬件执行环境的一个抽象，每一个进程都有自己的硬件执行环境，并且进程与进程之间相互隔离开来。

相对于进程级别的虚拟化，虚拟机则是系统级别的虚拟化。进程虚拟单个进程的执行环境不同，系统级虚拟化技术虚拟的是整个计算机的所有硬件资源，其抽象粒度是整个计算机，包括CPU、内存和外设等计算机资源，这个抽象出来的环境称作虚拟机。在每一个虚拟机环境下都可以运行一个完整的操作系统，这样的话一台真实的计算机就可以借助虚拟化技术虚拟出多个虚拟机，从而在一台计算机上面运行多个操作系统。从本质上来说，实体物理计算机和虚拟机可以使用完全不同的指令集架构(ISA)，比如在x86的物理计算机上可以运行ARM架构的虚拟计算机。但是不同的指令集架构会导致虚拟机性能下降，因为虚拟机的每一条指令都需要在物理计算机上进行模拟执行。相同体系架构的系统虚拟化一般来说性能比较好，虚拟机监控器(VMM)实现起来也会比较简单，因为大部分指令可以直接在处理器上运行，只有部分指令需要VMM模拟执行。

2.1.1完全虚拟化技术

由于硬件体系结构的历史遗留问题，硬件在虚拟化方面设计存在缺陷，会导致部分虚拟化漏洞，因此系统级虚拟化不能直接有效地完成。在硬件厂商还未提供足够的虚拟化支持之前，有两种可行的基于软件的虚拟化技术方案：直接修改代码和模拟执行。其中模拟执行对应的就是基于软件实现的完全虚拟化技术，而直接修改系统的代码对应的是类虚拟化技术。模拟技术通常来说可以被应用到完全不同的体系结构的虚拟化中，即是在一种硬件体系结构上模拟出另一种不同的硬件体系结构的执行环境。而如果是在物理机上模拟与其相同的体系结构的虚拟机会容易一些，因为大部分指令不需要被模拟，可以直接在在真实的硬件上执行。可以采用二进制翻译技术或扫描修补技术进行优化虚拟机性能。

在虚拟化技术中，最简单的就是解释执行，即取一条指令，模拟其执行效果，再取下一条指令，重复如此。如图2-1(a)所示，正常执行的时候是直接在CPU运行编译好的程序；如图2-1(b)所示，灰色部分表示会被载入到物理CPU执行的代码，对于解释执行技术，编译好的代码并不会被加载到物理CPU上直接执行，而是由解释器逐条解码，然后调用对应的函数来模拟其执行效果。虽然这种技术保证了所有指令都受VMM的监控，但是其致命的缺点是性能太差了。对于与物理机具有相同的体系结构的虚拟机，有很多非敏感指令不需要模拟，可以运行在物理CPU上。可以采用扫描与修补技术来优化其性能。



**图2-1 正常执行与解释执行**

扫描与修补技术让大多数指令直接在物理CPU上运行，用跳转指令或者会陷入VMM的指令替换掉系统中的敏感指令，使系统一旦执行敏感指令就会进入到VMM中，由VMM模拟执行。在每一个虚拟机执行每一段代码前，VMM会扫面并检查这段代码，查找到敏感指令和特权指令；接着VMM会为每一个需要修补的指令生成一段动态补丁代码；敏感指令被替换为一个跳转指令，进入VMM空间，在VMM中执行动态生成的补丁代码；执行完补丁代码后跳转回虚拟机中的下一条代码。大多数客户机系统和用户代码都可以直接在CPU上运行，所以扫描修补技术可以让虚拟化性能损失较小。但是各指令模拟执行时间长短不一，并且每个补丁代码引入了额外的跳转，会降低代码的局部性。

二进制翻译技术在VMM空间中开辟一块代码缓存，将代码翻译好后放在其中，客户机的代码不会被直接执行，而是翻译后放在代码缓存中。对于与物理体系结构相同的虚拟机，其大部分指令是可以直接进行等值翻译的，即原代码与目标代码是一样的。

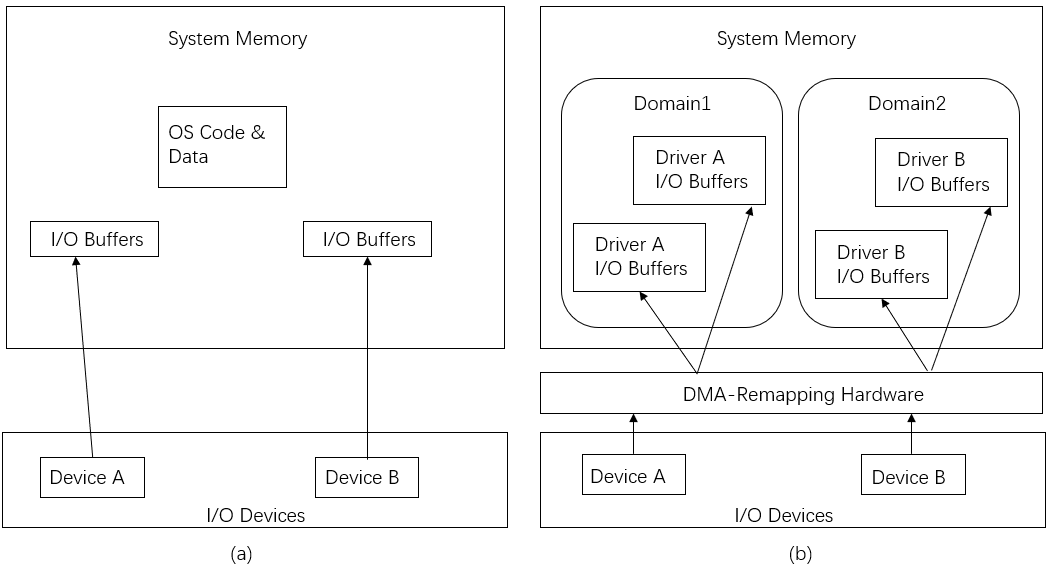
2.1.2硬件辅助虚拟化

由于原有的硬件体系结构对虚拟化技术的支持存在设计上的缺陷，硬件厂商为虚拟化技术提供了虚拟化技术支持，借助硬件虚拟化支持来实现搞笑啊地虚拟化。比如说Intel Virtualization Technology(Intel VT)，Intel VT是intel平台上的硬件虚拟化技术的总称，包含了内存、CPU以及I/O设备的虚拟化硬件支持。对于CPU虚拟化，intel提供了VT-x(Intel Virtualization technology for x86)技术；对于内存虚拟化，intel提供了EPT(Extended Page table)技术；对于I/O设备虚拟化，intel提供了VT-d技术。此外AMD平台也提供了类似的技术支持，即AMD-V技术，其原理与intel VT相似。

对于CPU虚拟化，VT-x引入了两种操作模式：root模式与non-root模式，VMM运行在root模式下，客户机运行在non-root模式，两种模式都有对应的ring0~ring3特权级。通常来说指令虚拟化通过“陷入再模拟”实现，而IA32架构下有19条敏感指令不能通过这种方式处理，直观的办法是使得这些敏感指令触发异常，但是这样会改变原有语义导致软件不兼容问题。而VT-x可以很好解决这个问题，在non-root模式下，所有敏感指令的行为都被重新定义，使得它们可以直接运行或者通过“陷入再模拟”方式处理；而在root模式下所有指令和传统IA32一样，没有改变。VT-x退出non-root模式进入root模式称为VM-Exit，从root模式返回non-root模式叫做VM-Entry，VT-x引入了VMCS(Virtual-Machine Control Structure)来保存CPU运行的相关状态，当COU发生VM-Exit和VM-Entry时会自动查询和更新VMCS。

内存虚拟化主要是完成内存地址空间的虚拟化，一般来说内存虚拟化需要两次地址的转换来实现，客户机虚拟地址GVA→客户机物理地址GPA→宿主机物理地址HPA，其中GVA→GPA是客户机系统实现，GPA→HPA是由VMM实现的。传统的架构仅支持一次地址转换，无法完成两次地址翻译，软件实现用影子页表技术来解决这个问题，VMM将两次转换合并为一次转换，VMM根据GVA→GPA→HPA映射关系得到GVA→HPA的映射关系，并将其写入影子页表，但是影子页表的维护和开发都很困难，并且需要为每个客户机进程都维护一个影子页表，内存开销较大。VT-x提供了EPT(Extended Page Table)技术，从硬件上支持GVA→GPA→HPA两次翻译，CR3寄存器完成GVA→GPA的翻译，然后通过EPT完成GPA→HPA的翻译，两次转换都是有硬件自动实现的，其转换效率非常高，使用EPT技术，客户机内存的缺页、CR3寄存器访问等都不会引起VM-Exit，减少额VM-Exit的次数，大大提高了性能。

对于I/O虚拟化的软件实现方式，有设备模拟和类虚拟化两种方式，前者用软件方式完全模拟，不需要修改系统代码，通用性强，但性能不理想；后者性能不错，但是通用性不强。而英特尔公司的VT-d技术为I/O虚拟化提供了强有力的硬件支持，可以帮助虚拟化软件同时实现性能好，通用性强的虚拟化技术。对于高性能，最直接的方式就是让客户机可以直接操作设备；对于通用性则需要让客户机系统自带的驱动可以访问和操作设备。VT-x技术可以让客户机直接访问I/O设备；而VT-d技术则可以让设备的DMA操作直接访问客户机的内存空间，VT-d技术在北桥引入DMA重映射硬件，提供设备重映射和直接分配设备的功能。如图2-2(a)所示，在VT-d技术没开启前，设备的DMA操作可以访问整个物理内存；如图2-2(b)示，开启VT-d功能后设备所有的DMA操作都会被重映射硬件截获，把DMA中的地址进行转化，让设备只能访问到指定的内存。



**图2-2 VT-d访问内存架构**

2.1.3类虚拟化技术

完全虚拟化技术需要给客户机系统提供一个完整的完全一致的硬件平台抽象，这样的方式使得客户机系统以及上层应用可以不需要修改直接运行在虚拟机中。而类虚拟化技术，也即半虚拟化技术，只实现部分硬件抽象，修改暴露给客户机的硬件抽象和上层操作系统，使得客户机中的系统意识到虚拟环境的存在，并配合VMM工作实现高效的虚拟化。虚拟的硬件抽象可以被修改成各种各样的形式，所以对应运行在其上的虚拟机也具有各种各样的形式。为了实现不同的目的，不同的虚拟化系统的设计可以有非常的区别。实际硬件与虚拟硬件差别越大，客户机操作系统以及上层应用程序需要改动得越大。虽然类虚拟化可以进一步优化虚拟化性能，但类虚拟化最大的问题是需要改动操作系统，增加了操作系统的开发和调试工作量，每一种操作系统都需要针对VMM进行修改才能运行，并且由于现代操作系统一直在进行版本的替代更新，所以对于统一操作系统的不同版本也需要进行类虚拟化的更新维护。另外，对于非开源的操作系统，其代码的修改和维护难度更大，对于已经停止维护的操作系统对其加入类虚拟化支持也是比较困难的。

2.2 ARMv8架构[5]

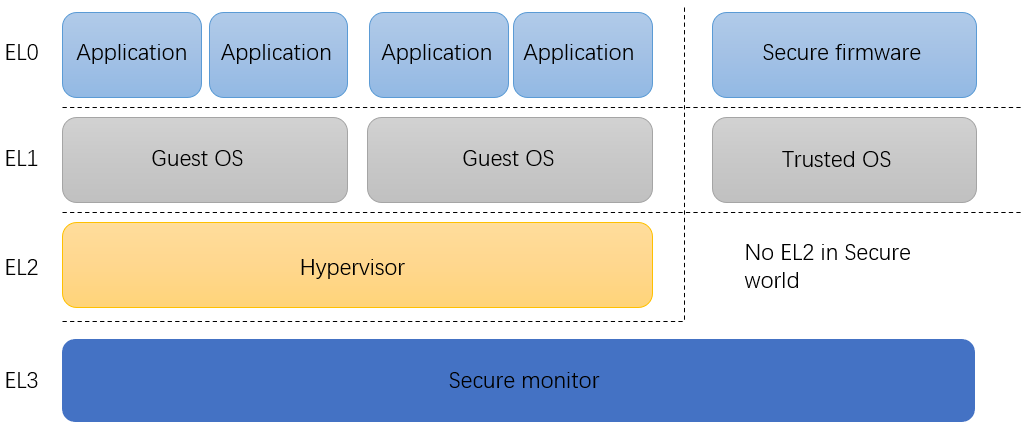
ARMv8架构是ARM公司为满足新需求在32位ARMv7架构的基础之上重新设计的一个架构，ARMv8架构是ARM架构近二十年来改动最大的改动。它引入了64位处理技术、4层异常模型(Exception Level)等等新特性，与以往的32位ARM架构有很大的区别。

ARM诞生之初，intel已经主导了绝大部分PC市场份额，ARM一直没有针对PC端进行架构设计。与x86的复杂指令集(CISC)体系结构不同的是，ARM使用的是精简指令集(RISC)体系结构，从最开始的ARMv4架构到ARMv7架构都是针对低功耗的移动端设备，就性能而言与PC端芯片无法相提并论。但是从ARMv7架构开始，ARM公司开始拓展他们的业务范围，而不仅限于移动设备。ARMv7退出了三个系列分别是：A系列（Application）、R系列（Real-time）和M系列（Microcontroller），分别对应不同的细分应用市场，其中A系列就是针对性能要求较高的应用。尤其在Cortex-A9之后，ARM的性能有了很大的提升，开始逐渐吸引一些PC端的用户，开始出现基于ARM的类PC产品，如平板电脑、数字电视等等，甚至高性能的ARM处理器开始有机会应用于企业设备、服务器等其它领域。与此同时，新的需求也越来越多，比如说大内存、虚拟化以及安全需求，虚拟化在ARMv7架构上已有了简单的支持实现，安全问题也可以在ARMv7的架构基础之上进行扩展，而大内存是比较棘手的问题。由于ARM处理器性能越来越好，并且运行在其上的软件实现也越来越复杂，导致单一的应用对内存的需求可能会超过32-bit架构所能支持的最大范围(4G)，所以ARM公司设计了新的架构，使用64-bit的指令集，即ARM64。

2.2.1 AArch64 Exception Level

ARMv8以前的处理器架构具有9种不同的工作模式(processor mode)，包括8种特权模式(privilege level)和1种非特权的用户模式(non-privilege level)。分别是User(USR)、FIQ、IRQ、Supervisor(SVC)、Monitor(MON)、Abort(ABT)、Hypervisor(HYP)、Undef(UND)、System(SYS)。不同的模式具有不同的硬件访问权限，在用户模式下，某些操作被限制，如MMU的访问。除User模式外其它模式基本与各种异常一一对应，而且不同的模式下都有一些独立的寄存器，仅在该模式下可访问。而ARMv8则摒弃了processor mode的概念，取而代之的是4个Exception Level，分别是EL0、EL1、EL2和EL3。ARMv8需要向前兼容ARMv7的AArch32状态，所以ARMv8提供了AArch32和AArch64两种运行状态，当ARMv8处于AArch32状态时将User、SVC、ABT等这些模式映射到EL0-EL3。

如图2-3所示，用户应用运行在最低的特权级EL0，系统以及虚拟化客户机系统都运行在EL1特权级，而EL2特权级则是专门为虚拟化提供的硬件支持，虚拟化Hypervisor就运行在EL2特权级，EL3则是为安全设计的安全模式，即ARM的Trust Zone技术。当异常发生时会切换Exception Level。对于AArch32，user模式对应AArch64的EL0特权级；Supervisor, IRQ，Abort，fiq，Undef,，System，六种特权模式，对应AArch64的EL1特权级；HYP与AArch64的EL2对应，运行虚拟机；而MON则对应AArch6的EL3特权级，运行安全管理。对于普通的操作系统，EL2与EL3特权级别都是不一定需要的，一个传统的操作系统仅仅需要EL0和LE1两个权限级别即可正产运行，内核态运行在EL1，用户态则运行在EL0权限级别。



**图2-3 ARMv8特权级架构**

2.2.2 ARMv8 TrustZone技术

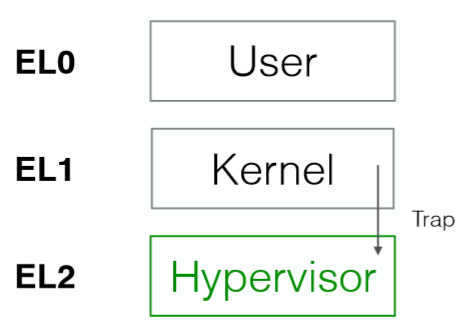
现如今移动端设备以及穿戴设备越来越普遍，人们时时刻刻都在使用这些设备，而这些设备又恰巧关乎人们的私密信息，比如个人指纹数据、财务账户密码、个人私密信息等等。TrustZone就是ARM公司设计出来保护人们数据安全的一种架构技术，其目的是消费电子产品简历一个安全框架抵御各种各样的攻击。

TrustZone架构技术为系统开发人员提供了系统级安全扩展和安全硬件支持，以保护运行在硬件上的操作系统的安全。ARM TrustZone技术为用户提供了硬件支持的安全隔离执行环境，将可信软件或程序运行在隔离的安全环境内保证其安全性。在以前，TrustZone技术仅出现在ARM Cortex-A系列处理上，随着ARMv8-M系列架构的发布，Cortex-M系列微处理器引入了TrustZone技术，让TrustZone技术可以被应用到嵌入式安全系统中，比如说对车载实时控制系统TrustZone技术是一个很好的安全解决方案。

TrustZone是一种基于硬件的安全隔离技术，为软件提供更高权限和独立的执行环境；TrustZone将硬件分为两个执行环境，一个正常执行环境，一个安全执行环境；正常的执行环境不可以访问安全执行环境下的硬件资源，而安全执行环境则有所有硬件资源的访问权限。并且在安全环境下可以单独运行一个可信操作系统，同时在正常执行环境下可同时运行一个正常的操作系统。TrustZone可以将很多硬件资源进行分区，比如将内存分为正常内存和安全内存，正常的执行环境下的系统不可以访问被标记为安全的内存，而安全环境可以访问整个物理内存；除此之外，还有I/O设备和中断也可以用TrustZone技术进行分类，一个I/O设备可以被分配给特定的执行环境，与内存类似，TrustZone确保安全的I/O设备不会被正常执行环境访问到，而安全环境可访问所有I/O设备；中断也可以分为安全中断和非安全中断，当中断到达时，先判断是安全中断还是普通中断，如果是安全中断，TrustZone将切换到安全执行环境下来处理。值得一提的是，TrustZone技术在设计之初并未针对虚拟化进行技术支持，这也为以后的ARM应用到云服务器端留下了安全隐患[1]，上海交通大学软件学院ipads实验室针对这个问题提出了解决方案，并且在USENIX Security会议发表了相关论文，该领域不属于本文范围，不在此作详细讨论。

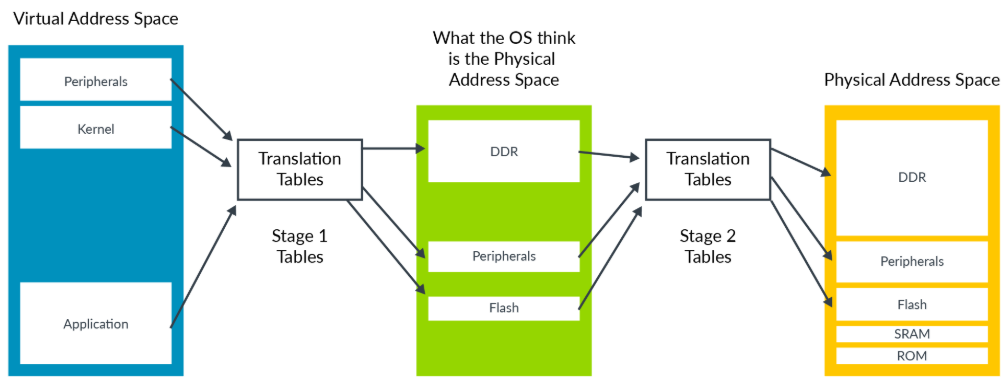
2.2.3 ARMv8虚拟化技术

ARM体系结构并不具备虚拟化功能，ARM公司在最新的ARMv7和ARMv8体系架构中引入了硬件虚拟化支持。虽然在ARMv7架构中已经设计了特殊的CPU执行模式来运行虚拟机管理程序，但在ARMv8架构中直接将虚拟化技术作为架构的一部分，可见虚拟化技术在现代操作系统中的重要性，ARMv8将虚拟化技术集成到特权级EL2中。在ARMv8架构下，当虚拟化功能使能后，Hypervisor运行在EL2异常级别，只有EL2异常级别或者更高的异常级别才可以访问和配置各个虚拟化功能设置，而客户机系统运行在EL1异常级别，客户机普通应用程序运行在EL0异常级别，如图2-4所示，客户机正常运行时处于EL0用户模式与EL1系统模式，直到虚拟机执行了某些敏感指令或者访问敏感资源，虚拟机将会trap到EL2级的hypervisor进行处理，hypervisor处理完成后CPU切换回EL0与EL2权限级别。ARM架构允许一些trap能被客户机内核处理，而不是所有trap都进入hypervisor处理，比如说系统调用或者内存缺页异常可以直接被客户机内核处理而不需要hypervisor处理，这避免了每次系统调用或者内存缺页都切换到hyp模式，减少了虚拟化开销。ARM公司为了简化hypervisor的开发，减少了EL2 Hyp模式下的寄存器。ARMv8硬件虚拟化支持主要包括二段翻译[6](Second-stage of translation)、中断控制器(GIC Generic Interrupt Controller)等等硬件虚拟化支持。



**图2-4 trap and emulate**

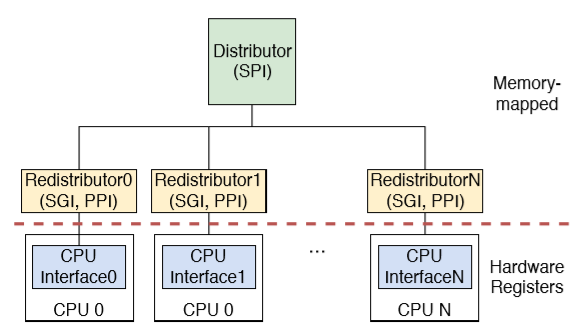
两级地址转换类似于x86平台上的EPT技术，帮助客户机系统完成两次内存地址翻译，而不需要使用影响性能的影子页表实现方式。在EL2与EL1异常级别具有独立的一套寄存器，如图2-5所示，在内存翻译第一阶段，使用TTBR0\_EL1/TTBR1\_EL1寄存器存放第一级页表的地址，实现从客户机虚拟地址到中间物理地址(IPA)的翻译，中间物理地址也即是客户机的物理地址，这一部分由客户机里运行的操作系统自己完成。要得到真实的物理地址还需要进行第二次地址翻译，当HCR\_EL2的第0位即VM位使能后，则开启了虚拟化功能，运行在EL2异常级别的Hypervisor可通过VTTBR\_EL2访问第二级翻译的页表，将中间物理地址翻译为真实物理地址。



**图2-5 ARMv8两段内存翻译[6]**

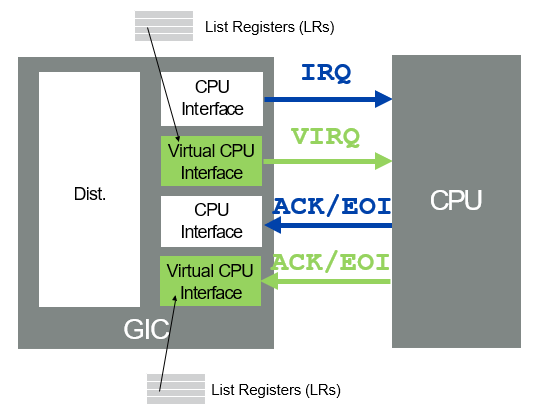
ARMv8架构下使用SMMUv3(类似于x86平台下的IOMMU)来实现虚拟化平台下外设的直接存储器访问DMA重映射，SMMU是一个MMU部件，但不提供给CPU使用，而是给外设使用。计算机外设通过DMA技术将数据直接传输到物理内存上，从而减轻CPU的工作，数据传输完成后通过中断通知CPU去读取。SMMUv3支持I/O地址的一级转换和两级转换，在使能虚拟化的情况下，SMMU两级翻译与内存stage-2两级翻译类似，第一段将客户机虚拟地址转换为中间物理地址，第二段将中间物理地址翻译为真实的物理地址。一个平台上可以有多个SMMU设备，设备之间可能会用不同的页表，不同的页表需要加以区分，ARMv8平台下SMMU用StreamID进行区分，类似于x86架构下用于区分PCI设备的BDF号码，同时一个设备可能同时会被多个进程使用，每个进程都有不同的页表，SMMU用SubstreamID进行区分。

中断控制器(GIC)对虚拟化技术有着至关重要的作用，ARMv8架构下使用的中断控制器是GICv3[7]。在ARMv8架构下有四种中断类型：软件中断(SGI software generated Interrupt)、私有外设中断(PPI Private Peripheral Interrupt)、共享外设中断(SPI Shared Peripheral Interrupt Distributor)和基于消息类型的中断(LPI Locality-specific Peripheral Interrupt)。如图2-6所示，GICv3由三个独立的组件组成：Distributor, Redistributor 和CPU interface。整个硬件平台只有一个Distributor，而每个CPU核都有对应的一个Redistributor和一个CPU interface。Distributor管理SPI中断，负责将中断发送给Redistributor；Redistributor负责PPI，SGI和SPI中断的管理，并将中断发给CPU interface；而CPU interface则负责将中断传输给对应的CPU，CPU interface也用于ACK(acknowledge)和返回EOI(End-Of-Interrupt)信号，比如当一个CPU核收到一个中断时，CPU会去读CPU interface对应的寄存器，并向CPU interface应答中断，然后Distributor将中断状态从pending改为active，这样在CPU处理完中断前该类型中断不会被发起处理，处理完后CPU写EOI寄存器表示处理结束。每个Redistributor都和一个CPU interface相连。其中CPU interface是实现在CPU内部的，而Distributor和Redistributor则是实现在GIC内部的。



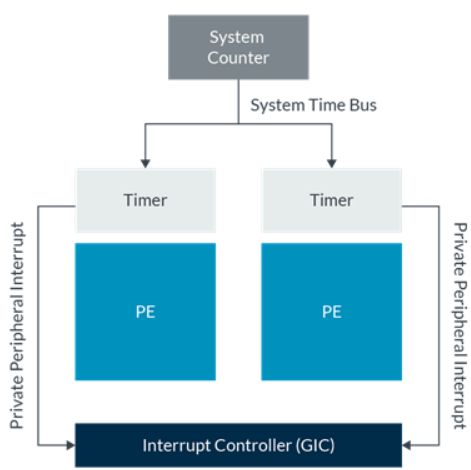
**图2-6 GICv3架构图**

中断既可以配置为trap到EL1级的系统处理，也可以配置为trap到hyp模式下处理。CPU interface实现为CPU内部的一组寄存器，并且从硬件上支持虚拟化，virtual GIC(VGIC)，VGIC为每个CPU都引入了一个VGIC CPU interface和对应的Hypervisor控制接口，虚拟机可访问操作的是VGIC CPU interface但无权访问GIC CPU interface。虚拟中断通过hypervisor写特殊的寄存器而产生，list registers(LRs)。VGIC支持ACK和EOI，所以这些操作不会陷入hypervisor，减少了虚拟中断开销。如图2-7所示，当使能虚拟化功能后，外设产生的中断首先会发送到Distributor，Distributor将这个中断发给CPU interface，CPU interface让运行在EL2异常级别的hypervisor去处理这个中断，hypervisor会检查这个中断，如果是发送给客户机的中断，则将设置一个虚拟中断，并且将虚拟中断映射到对应的物理中断，把虚拟中断通过virtual CPU interface，即写对应的寄存器，发送给对应的客户机，客户机处理完这个虚拟中断后返回结果，virtual CPU interface会发现这个虚拟中断来自于一个物理中断，将清除Distributor上的物理中断表示中断处理完毕，至此整个虚拟中断过程结束。



**图2-7 GIC虚拟中断[8]**

ARMv8架构下的时钟架构叫做ARM generic timer[9]，该硬件支持虚拟化。如图ARMv8时钟硬件主要由系统计数器(System timer)和附着在各个处理器上的通用时钟(timer)组成，每个处理器有四个timer：三个物理时钟和一个虚拟时钟。系统计数器记录的是真实的物理世界的时间，系统计数器可将计数值广播给每个处理器；而附着在处理器上的时钟处理器上的时钟可根据与系统计数器进行比较而出发时钟中断；除此之外还有每个处理器还有一个虚拟时钟，虚拟时钟是在物理时钟的基础之上减去一个偏移，虚拟时钟只记录虚拟机运行的时长。

****

**图2-8 ARM时钟架构[9]**

2.3 本章小结

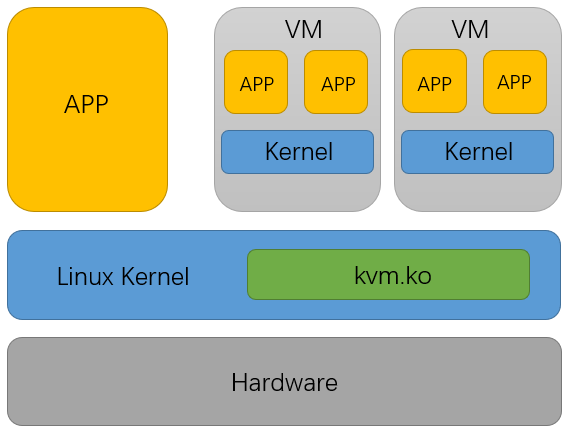
本章对本课题的技术背景进行了详细的介绍，虚拟化技术无疑是近十几年最热门的计算机系统软件技术之一，从该技术提出到如今，虚拟化技术出现了完全虚拟化、硬件辅助虚拟化以及类虚拟化等不同实现方式。ARMv8架构被普遍应用到现如今的车载硬件平台，ARMv8架构是近几十年来ARM公司最大的一次架构改动，它引入了完全不同的异常级别EL0-EL3，专门为虚拟化设计了EL2异常级别，并且利用GIVv3、SMMUv3以及stage-2翻译等硬件对虚拟化做了很好的辅助支持，深入了解虚拟化技术以及ARMv8架构下的硬件辅助虚拟化技术对本课题下一步工作至关重要。

第三章 ARM虚拟化方案探索

3.1 KVM/ARM架构

3.1.1 KVM介绍

KVM是2006年以色列Qumranet 组织开发的一种开源虚拟化解决方案，并且自2007年起Linux就开始正式将KVM并入内核。如图3-1所示，KVM是基于Linux内核的type2型虚拟化方案，其实质是一个Linux系统下的一个内核模块，通过加载内核模块将Linux kernel变成一个Hypervisor。



**图3-1 KVM架构图**

KVM是基于硬件虚拟化扩展(intel VT或者AMD-V)的完全虚拟化方案，在KVM方案中，虚拟机被实现为Linux系统下的一个进程，KVM复用Linux调度程序进行虚拟机调度，KVM可以最大限度复用很多Linux内核的代码，从而减少了开发量。KVM的内核模块叫做kvm.ko，主要实现CPU和内存虚拟化；KVM本身不做任何设备模拟，这部分工作借助QEMU来进行，QEMU运行在用户空间，而KVM运行在内核态，两者通过对/dev/kvm的IOCTL调用进行交互[10]。

KVM内核模块是KVM的核心部分，其主要功能是初始化CPU硬件，并打开其虚拟化功能，让虚拟机运行在虚拟化模式下。KVM内核加载后首先初始化其数据结构，内核模块检查当前CPU，通过操作CR4寄存器使能虚拟化功能，并通过执行VMXON指令将当前宿主机系统运行于虚拟化模式下的根模式，最后KVM创建设备文件/dev/kvm。

如上所述，KVM初始化后只有/dev/kvm设备文件，创建虚拟机则需要对该文件进行IOCTL调用，创建虚拟机实质上就是KVM为某个虚拟机创建对应的内核数据结构，同时KVM返回一个文件句柄表示创建的虚拟机，对该虚拟机进行操作管理则可以通过对该文件句柄的IOCTL调用进行，同样的，KVM也会为每个虚拟CPU创建句柄，对虚拟CPU的操作也可以通过对该文件句柄的IOCTL调用进行。对于内存虚拟化KVM开始使用影子页表技术实现，但其实现复杂，维护开销较大，目前KVM通过EPT技术实现。

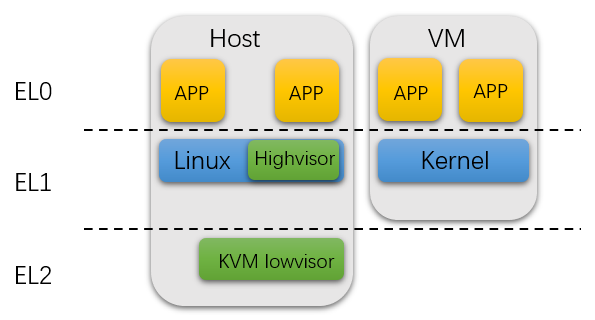
QEMU本身并不是KVM的一部分，它自己就是一个著名的开源虚拟化方案，QEMU是基于纯软件模拟实现的，性能相对比较差，但其优点是QEMU代码具有整套虚拟机实现，包括CPU虚拟化、内存虚拟化以及KVM使用到的虚拟设备模拟。为了复用代码简化开发，KVM在QEMU的代码基础之上进行了修改，当虚拟机进行外设I/O通信时，KVM通过QEMU来解析和模拟这些设备。

3.1.2 ARM架构下的KVM

KVM一开始并没有针对ARM架构进行设计，KVM针对ARM架构的移植实现并没有去重新设计和实现复杂的功能，这样会引入很多棘手和致命的问题，KVM/ARM在KVM的基础之上利用利用Linux内核现有的基础设施来对ARM架构进行移植。ARM硬件在很多方面比x86平台硬件更多样化，不同的设备厂商经常将各种硬件组件以非标准的方式集成到ARM设备中，而ARM平台又缺乏像BIOS或PCI总线这样的硬件发现功能，这样的话对Type1型的Hypervisor造成了困扰，比如说对于每一个Xen需要支持的SoC，开发人员必须在Xen hypervisor核心中实现设备驱动。然而几乎所有的ARM平台都支持Linux，只需要将KVM/ARM于Linux集成就可以在最新版的Linux中使用KVM/ARM。

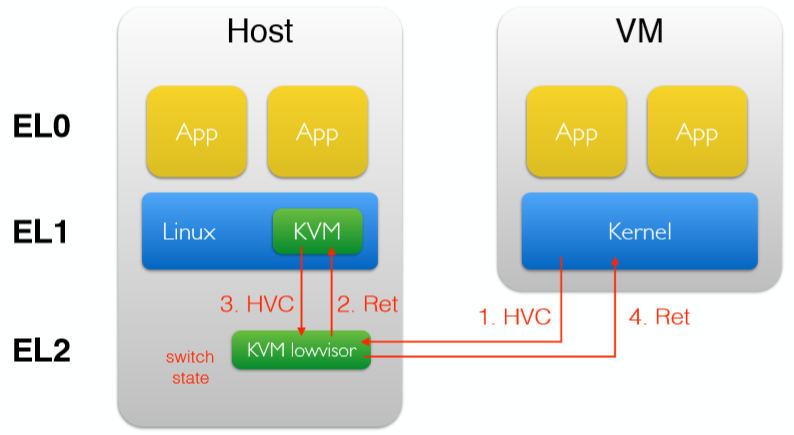
如果能将KVM直接运行在ARMv8的Hyp模式下是最好的，因为这个模式权限更高。但是由于KVM复用了现有的内核基础，如调度器，在Hyp模式下运行KVM意味着需要在Hyp模式下运行Linux内核。这样至少会引入两个致命的问题，首先Linux底层架构相关代码是针对EL1级的内核模式编写的，不能在未经修改的情况下就运行在Hyp模式下。在Hyp模式下运行Linux内核就需要大量改动内核代码，这对Linux社区来说是不可接受的，并且还需要考虑Linux兼容EL1级的内核模式，这是一个非常大工程量。其次，在Hyp模式下运行Linux系统可能会对性能造成很大的影响。比如说，EL1的内核模式有两个页表寄存器，分别保存用户态地址页表和内核态页表地址，然后EL2级的Hyp模式只有一个页表寄存器，这样的话导致无法直接访问用户地址空间，内核需要将用户地址空间映射到内核空间，并且还有TLB表的维护工作，这会导致Linux运行在Hyp模式性能较差。

KVM/ARM引入了分割模式虚拟化[11](split-mode virtualization)，这是一种全新的虚拟化方案，它将hypervisor分割为两个部分，分别运行在不同的CPU模式下，以充分地利用每个CPU模式的有点和功能。KVM利用分割模式虚拟化爱利用ARM的Hyp模式对虚拟化的硬件支持，同时复用运行在内核模式下Linux的内核服务，可以将KVM于Linux集成，而无需对现有的代码进行重大改动。如图3-2所示，将KVM分割为两个部分：lowvisor和highvisor。Lowvisor利用Hype模式下的硬件虚拟化支持来听三个主要的功能。首先，lowvisor通过硬件配置来设置正确的执行上下文(execution contexts)，并且在不同的执行上下文间实施保护与隔离；其次lowvisor管理虚拟机执行上下文与主机执行上下文之间的相互切换，主机执行上下文包括hypervisor和宿主机Linux，由于lowvisor是运行在Hyp模式下的唯一组件，所以只有它才能管理主机执行上下文与虚拟机执行上下文之间的切换所需的硬件配置；第三，lowvisor提供虚拟化陷入处理程序(execution contexts)来处理陷入到Hypervisor的中断和异常。因为lowvisor有最高权限，直接与硬件交互，接触一些敏感资源，所以lowvisor代码量应当尽可能的少，lowvisor只完成最小数量的处理，大部分工作切换到highvisor再进行处理。



**图3-2 ARMv8下的KVM架构图**

Highvisor作为宿主机Linux内核的一部分以内核模式进行运行，因此它可以利用现有的Linux内核功能，如调度器，也可以复用标准内核数据结构和机制来实现其功能，如锁机制和内存分配功能。这个特点使得高级功能更容易在highvisor中实现。比如，虽然lowvisro提供了更底层的陷入处理程序，但虚拟机的stage2页表缺页异常是由EL1级的highvisor来处理和指令模拟。值得注意的是，如图3-3所示，虚拟机内核态是运行在与highvisor一样的EL1级的内核模式，但是stage2翻译需要陷入到EL2级的Hype模式，因为KVM是跨模式运行的，所以从虚拟机切换到highvisor涉及到多个模式的切换。一次到hypervisor的陷入首先需要从虚拟机陷入到运行在Hyp模式下的lowvisor，然后lowvisor会接着陷入到highvisor，类似的，从highvisor到虚拟机之间的切换也先需要从内核模式切换到Hyp模式下的lowvisor，然后再切换到虚拟机。因此在highvisor和虚拟机之间进行切换时，会导致双重陷入。



**图3-3 KVM/ARMv8陷入处理**

3.1.3 KVM/ARMCPU虚拟化

为了虚拟化CPU，KVM必须为虚拟机提供与底层硬件CPU相同的接口，同时需要确保Hypervisor对硬件的控制。这需要确保在虚拟机中运行的软件与运行在物理CPU上的软件具有相同的寄存器状态和持久访问权。不影响虚拟机隔离的寄存器状态可以通过将虚拟机状态保存并从内存中恢复主机状态进行上下文切换。KVM将虚拟机对其它敏感状态的访问都陷入到Hyp模式，由hypervisor进行模拟。比如虚拟机可以直接对Stage1页表寄存器进行操作，不需要trap到Hypervisor；而如果虚拟机执行WFI指令就会陷入Hypervisor，因为该指令会将CPU断电，这个操作只能由Hypervisor控制。KVM将某些寄存器状态切换尽可能推迟，对性能有一定的提升。从主机切换到虚拟机涉及以下操作：(1)将所有主机通用寄存器(General Purpose (GP) Registers)保存在Hyp栈上；(2)为虚拟机配置VGIC；(3)为虚拟机配置timer；(4)将主机特有的配置寄存器保存在Hyp栈上；(5)将虚拟机的配置寄存器加载到硬件上，这不会影响到当前的执行状态，因为Hyp模式有自己独立的配置寄存器；(6)在Hyp模式下配置，配置中断陷入、CPU暂停指令陷入、SMC(安全调用)指令陷入、访问特定配置寄存器陷入和调试寄存器陷入；(7)将虚拟机的特定编号ID写入shadow ID寄存器；(8)设置Stage2页表寄存器，并启用二段地址翻译；(9)恢复虚拟机通用寄存器，然后进入用户或者内核模式。CPU将一直运行在虚拟机内，直到执行产生陷入到Hyp模式的时间发生，需要来自highvisor的服务，KVM必须从虚拟机世界切换到highvisor，该切换需要一下步骤：(1)保存所有虚拟机的GP寄存器状态；(2)禁用两段地址翻译；(3)配置Hyp模式下不捕获任何寄存器访问和指令；(4)保存所有虚拟机特有的寄存器；(5)将主机寄存器状态加载到硬件上；(6)为主机配置timer；(7)保存虚拟机VGIC状态；(8)还原主机GP寄存器状态；(9)进入内核模式下的主机。

3.1.4 KVM/ARM内存虚拟化

KVM通过使能ARMv8两段翻译来为虚拟机提供内存虚拟化服务。Stage2翻译只能在Hyp模式下进行配置，highvisor部分管理stage2页表，只允许虚拟机访问为该虚拟机分配的内存；其他访问会导致从虚拟机陷入到hypervisor，确保虚拟机无法访问Hypervisor或者其他虚拟机的内存。在highvisor和lowvisor运行时，stage2翻译会被禁用，因为此时highvisor完全控制整个系统并且直接管理物理地址。当切换回虚拟机时重新使能stage2翻译，虽然虚拟机内核与主机内核运行在同一权限级别，stage2阶段地址翻译可以确保highvisor内存不被虚拟机访问。KVM/ARM以分离模式运行，可以借助Linux内核现有的内存分配、页面引用计数和页表操作的代码进行页表管理。相比之下，type1型的hypervisor需要编写一个全新的内存分配系统。

3.1.5 KVM/ARM I/O及中断虚拟化

KVM/ARM利用现有的QEMU和Virtio[12]用户设备模拟来提供I/O虚拟化。在硬件层面，ARM机构上的所有I/O机制都是基于对MMIO(Memory-mapped I/O)实现的。除了直接分配给虚拟机的设备外，所有硬件的MMIO区域虚拟机都不可以访问。KVM使用stage2两段翻译来确保虚拟机不可以直接访问物理设备，访问分配给虚拟机RAM区域之外的任何区域都将导致陷入到hypervisor，hypervisor可以根据故障地址将访问路由到QEMU中的特定模拟设备。

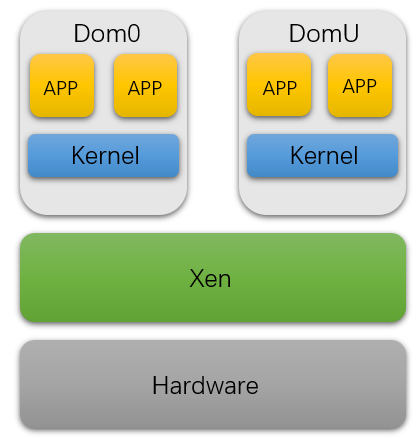
KVM与Linux是紧密集成的，重用了现有的设备驱动和相关功能，包括中断处理。在虚拟机运行时，KVM将CPU配置为所有硬件中断都会陷入Hyp模式，然后切换到highvisor，交给宿主机来处理该中断，保持highvisor对硬件资源的完全控制。当highvisor和宿主机运行时，中断不会陷入EL2级的Hyp模式，而是直接在EL1内核模式处理，避免两次权限级切换增加开销。以上两种情况，所有硬件中断处理都是在主机中通过重用Linux现有的中断处理功能来完成的。但是虚拟机必须接受来自设备模拟的虚拟中断形式的通知。KVM借助VGIC将虚拟中断注入虚拟机，如2.2.3节所述hypervisor通过操作CPU 控制接口中的列表寄存器产生虚拟中断，通过stage2页表配置防止虚拟机访问控制接口，但虚拟机可以访问VGIC的虚拟CPU interface。

KVM引入了虚拟distributor，这是GIC的distributor的软件模型，是highvisor的一部分，虚拟distributor对用户空间是开放的，因此用户空间中的模拟设备可以向虚拟distributor发起虚拟中断。虚拟distributor保存每个中断的内部软件状态，在调度到该虚拟机时通过操作列表寄存器进行中断注入。例如，如果VCPU0向VCPU1发送IPI，distributor将为CPU1操作列表寄存器，当CPU1再次运行时注入该中断。当不同的虚拟机运行在同一个物理CPU上时，需要对列表寄存器进行上下文切换，但在虚拟机和hypervisor之间进行简单切换时不一定需要该操作。例如，如果没有挂起等待的中断则没有必要访问任何列表寄存器。注意，当从hypervisor切换到虚拟机时将虚拟中断写入了列表寄存器，在从虚拟机切换回hypervisor时hypervisor也必须去读列表寄存器，因为列表寄存器描述了虚拟中断的状态，比如说虚拟机确认虚拟中断(ACK)。

3.2 Xen/ARM架构

3.2.1 Xen介绍

在虚拟化领域，除了KVM之外，Xen是另一个非常著名的开源虚拟化系统，Xen最初是剑桥大学计算机实验室的研究项目，2003年就发布了第一个版本，是更早于KVM的Type1型的虚拟化方案，如图3-4所示，与KVM不同的是Xen直接运行在裸机上，而不寄生与宿主机系统。一个好的系统设计思想应当是方案与机制分离，这也是Xen的基本设计原则[13]。Xen运行在操作系统和硬件之间，为上层的操作系统提供虚拟的运行环境。Xen设计了一些机制，然后把这些机制的具体方法实现交给了一个特权虚拟机实现Dom0。Xen只直接管理CPU和内存，不支持任何外部设备，所有的硬件设备驱动由运行在Dom0环境下的操作系统来提供，为众多IO设备开发驱动是一个很庞大的工程，这样可以精简Xen的实现。 所以一般来说Xen需要先运行一台特权虚拟机，并且该虚拟机下运行的系统内核必须支持修改，因此选择开源的Linux作为特权虚拟机的系统是最合适的，通常来说该特权虚拟机选择当前较流行的Linux发行版作为系统，因为支持更多的IO硬件设备。



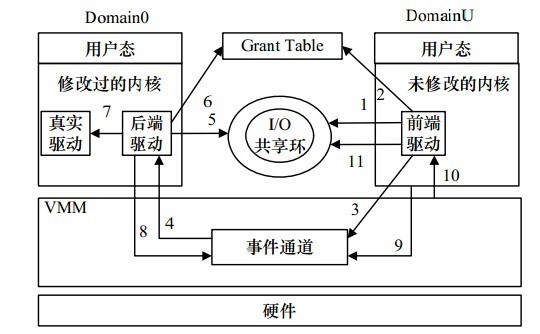
**图3-4 Xen架构**

总的来说Xen可以分为三大部分：Hypervisor层，第一个特权虚拟机和其它虚拟机。Hypervisor直接控制CPU和内存这些基础硬件，为所有虚拟机提供CPU、内存以及中断管理；特权虚拟机负责创建用户级虚拟机，并为其分配IO设备，一般来说运行在其中的系统时经过特殊修改后的操作系统，特权虚拟机可以直接访问IO硬件，在Xen中担任管理员的角色；其他虚拟机时提供给用户使用的虚拟机执行环境，这些虚拟机相互隔离。Xen对所有虚拟机统称为Domain，特权虚拟机称为Domain0(Dom0)，后续其它虚拟机称为Domain1, Domain2….，统称为DomU。

Xen为虚拟机提供vCPU时，hypervisor层启动一个线程，并将该线程映射到某个物理CPU上，通过修改虚拟机对应配置文件可以将vCPU指定映射到某个物理CPU，而内存的虚拟化则是通过内存页的映射将连续或者不连续的内存页映射给虚拟机，让虚拟机看起来是连续的内存。

对一个Xen虚拟机所需要的CPU和内存都由Xen hypervisor提供，而需要I/O设备则需要向Dom0虚拟机发起请求，Dom0将为该虚拟机创建一个模拟设备线程，该线程运行于Dom0用户空间，当用户虚拟机发起I/O调用时，Dom0里的模拟设备收到请求，并将其交给Dom0内核态处理，内核将其转换为Dom0对硬件的操作，这些模拟的设备借助Qemu完成，Xen本身不提供任何设备模拟。例如，当用户虚拟机向Dom0发起磁盘访问请求，Dom0将一个分区或者文件模拟成该虚拟机的磁盘，Dom0创建一个镜像文件，用Qemu模拟一个磁盘控制器并映射到用户虚拟机，所以虚拟机写磁盘可分为以下步骤：(1)虚拟机应用发起写磁盘操作；(2)进入虚拟机内核调用磁盘驱动进行写操作；(3)用户虚拟机将编码后的信息发给Dom0的模拟设备；(4)Dom0将编码后的信息还原后转发给Dom0内核；(5)Dom0内核调用磁盘驱动对真实物理磁盘进行写操作。

除此之外，Xen还支持半虚拟化IO设备，DomU是知道自己运行在虚拟化环境下的，Xen知道这个磁盘不是真正磁盘，只是一个设备前端驱动(Device Frontend)，需要对磁盘进行操作时直接将数据交给前端驱动，而不是去调用设备驱动，在Dom0里由设备后端(Device Backend)接收来自前端的数据，然后直接转发给内核调用物理驱动来进行处理。DomU与Dom0之间通过环状队列(I/O环)来进行数据传递，通过事件通道实现异步通知机制，通过授权表把内存映射到目的虚拟机，进行内存共享。如图3-5所示，当DomU向IO

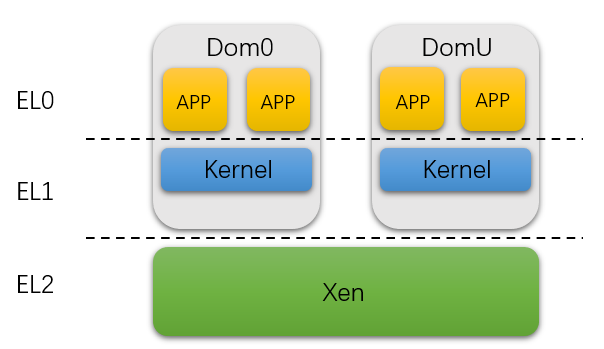


**图3-5 写入I/O设备流程图[14]**

设备写入数据时，(1)DomU产生I/O请求，并且在I/O环中产生请求，并将I/O数据放在授权表指向的内存中；(2)通过hypervisor的事件通道通知Dom0处理该请求；(3)Dom0收到来自事件通道的通知，并从I/O环中取出请求；(4)解析I/O请求，并从授权表指向的内存中取出数据进行处理；(5)将处理后的信息放入I/O环，并通过事件通道通知DomU；(6)DomU收到通知，并从I/O环取出响应事件，并进行处理。

3.2.2 ARM架构下的Xen

早在2007年官方就已经开始了Xen hypervisor对ARM架构的移植，与x86平台上的架构类似，Xen Hypervisor是一个轻量级微内核实现，只提供CPU、内存和中断的管理与虚拟化，I/O设备模拟仍然是由Dom0特权虚拟机借助修改过的Linux内核完成，与x86类似，Dom0与DomU之间的通信与I/O半虚拟化也是借助事件通道完成，完全虚拟化则用运行在Dom0内的Qemu模拟实现。如图3-6所示为Xen在ARMv8平台下的架构图，KVM/ARM不同的是，Xen整个Hypervisor直接运行在EL2异常级别，而没有跨级别情况，因此Xen在异常级别之间的上下文切换次数远小于运行在ARMv8平台上的KVM；虚拟机内核运行在EL1异常级，虚拟机用户态运行在EL0异常级。

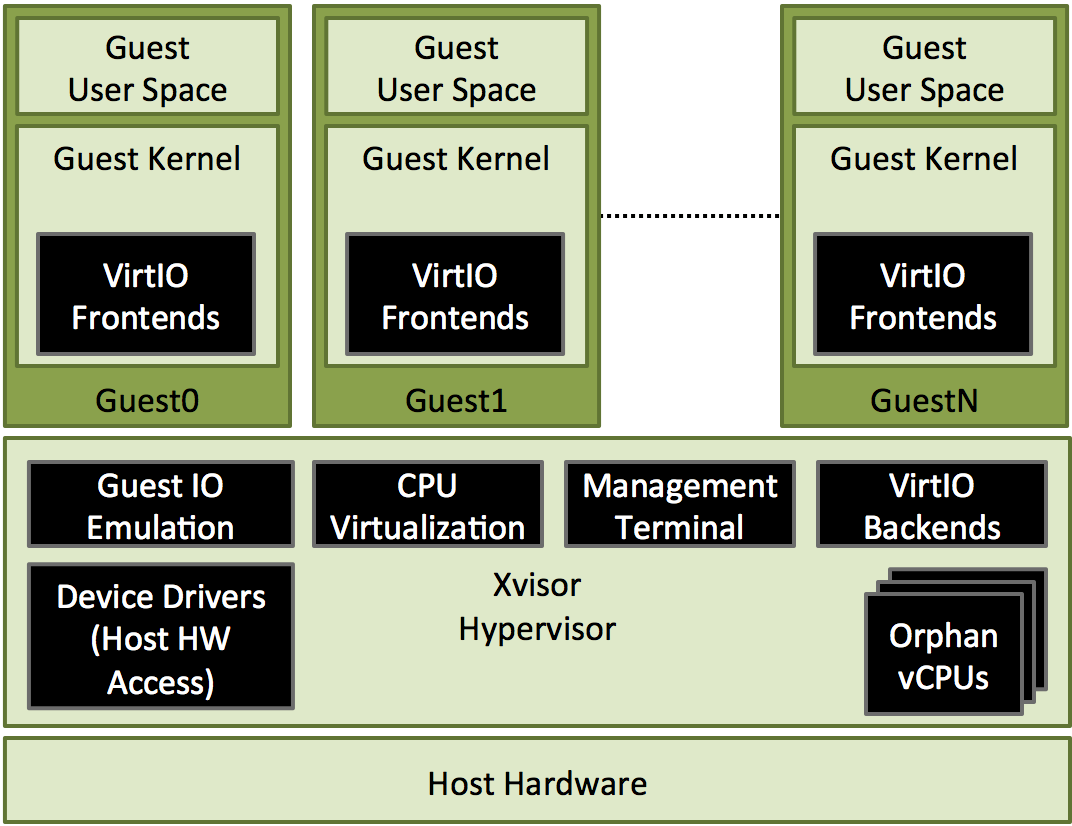


**图3-6 Xen/ARM架构图**

Xen在ARM平台下的内存虚拟化也是借助了ARMv8硬件虚拟化支持，用stage2两段翻译技术来实现客户机虚拟地址到中间地址再到真实物理地址的翻译，stage2页表支持4KB、2MB以及1GB内存页大小。中断与时钟的实现与KVM在ARMv8下的架构类似，中断借助GIC中断控制器来实现中断以及虚拟中断的管理；时钟使用ARMv8提供的通用时钟来为虚拟机提供虚拟时钟。

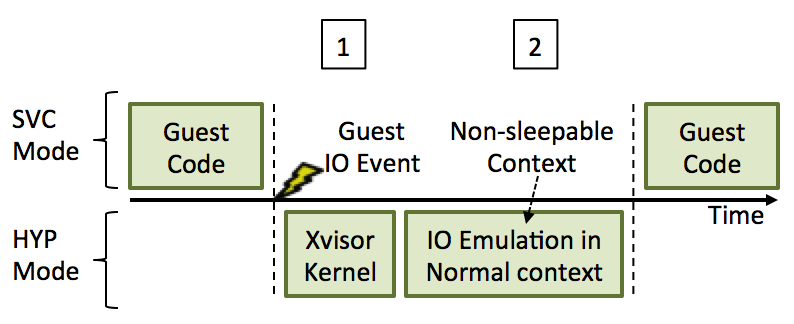
3.3 Xvisor/ARM架构

Xvisor是一个开源的type1型的虚拟化方案，致力于提供一个宏内核、轻量级、便捷、高性能、灵活的虚拟化解决方案。现在已经可以运行在ARMv5， ARMv6，ARMv7a，ARMv7a-ve，ARMv8a，x86\_64，RISC-V以及其它的一些CPU架构下。与其他虚拟化系统不同的是，Xvisor是少数同时支持有虚拟化扩展的ARM架构和无虚拟化扩展的ARM架构。Xvisor主要使用完全虚拟化方案，操作系统可不经修改就运行在其平台上，半虚拟化对Xvisor是可选项。如图3-7所示，Xvisor由以下几个部分组成：CPU虚拟化、客户机I/O虚拟化、半虚拟化服务(VirtIO后端)、设备驱动以及一些管理服务。另外，因为在ARM平台没有PCI总线来自动读外设资源，在ARM架构下用设备树(device tree)来配置所有硬件资源，所有的设备都用一个DTS(device tree script)描述。Xvisor hypervisor与Xen类似作为一个整体运行在EL2级，这样Xvisor的上下文切换比起KVM开销更低，能够更快地处理缺页异常、中断、指令陷入以及IO事件。并且与Xen借助Dom0不同的是，所有的物理设备驱动直接运行在Xvisor hypervisor内部，不会降低设备驱动的性能。但是Xvisor有一个致命的缺陷，它不像Linux那样支持丰富的硬件设备和多种多样的ARM单板，因为所有的驱动都是写在Xvisor内的，去完成这些驱动是一个非常庞大的工程。为了解决这个问题Xvisor提供了Linux兼容的头文件方便从Linux移植设备驱动和设备驱动框架。虽然不能完全解决问题，但是也大大简化了移植成本。



**图3-7 Xvisor架构[16]**

与其他ARM架构下的Hypervisor不同的是，Xvisor模拟客户机IO时不会引发多余的调度或上下文切换。如3-8图所示，一次客户机IO事件简单地陷入到Xvisor Hypervisor，然后直接进行处理返回结果即可。



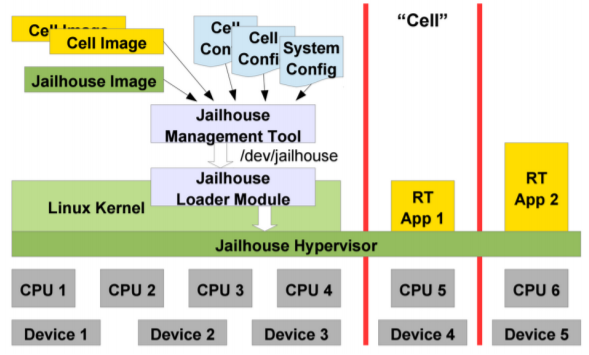
**图3-8 Xivsor客户机IO事件[16]**

Xvisor的内存虚拟化也借助了ARMv8的两段翻译，Xvisor支持4KB、2MB以及1GB的页大小翻译，与ARM架构下其它虚拟化方案不同的是，Xvisor在创建客户机内存前，提前为客户机分配好连续的物理内存，这种机制可以有助于内存提前预读，一定程度上改善客户机内存访问速度。

3.4 Jailhouse

3.4.1 Jailhouse介绍

Jailhouse是德国西门子公司推出的一款基于ARM架构的虚拟化方案，2015年五月才发布第一版Jailhouse 0.5，是一个非常年轻的支持ARM架构的虚拟化方案，现在也还在持续优化改进中。如图3-9所示，与传统虚拟化方案完全不同的是，Jailhouse时基于静态分区的虚拟化方案，不支持任何设备模拟，不同客户虚拟机之间也不共享任何CPU，所以也没有调度器。Jailhouse的设计理念是尽可能地简化hypervisor层的代码，jailhouse可以运行支持裸机运行的程序以及经过修改过的Linux操作系统，但是它不可以直接运行未经修改的通用现代操作系统，比如Windows和FreeBSD。Jailhouse的工作是将硬件资源进行静态分区，每个分区称为一个Cell，每个Cell之间是相互隔离开的，并且拥有只属于该客户机的硬件资源(CPU、内存、外设等)，运行在Cell内的软件或操作系统称为inmate。Jailhouse十分轻量级，比Xvisor、Xen的代码量更像，只有不到一万行代码，值提供基本的硬件抽象和虚拟化，没做任何资源的超售，所以几乎可以达到裸机运行的性能。Jailhouse启动后直接运行在裸机上，它控制了所有的硬件资源并且不需要其它的任何技术支持。



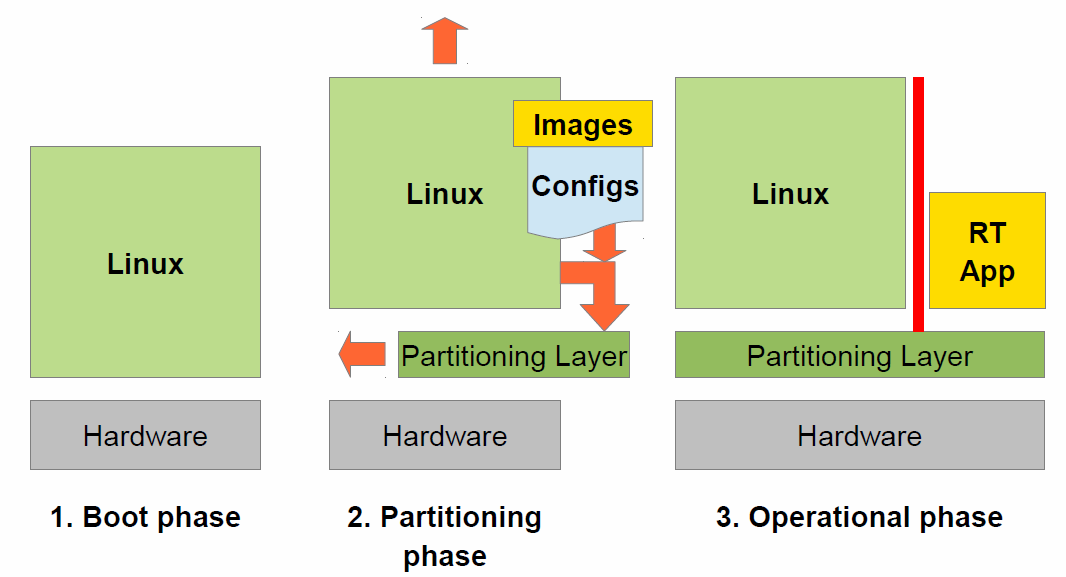
**图3-9 Jailhouse架构图[19]**

Jailhouse的第一个Cell叫Root Cell，这是一个特权Cell，与Xen的Dom0有异曲同工之妙，但也有很大的不同，在Root cell内部运行着一个Linux系统，Jailhouse与该系统紧密集成，并且依赖该Linux系统进行硬件初始化和启动(Bootstrapping)，现代计算机的启动过程是一个非常复杂的过程，如果在Jailhouse内部实现该过程会导致Jailhouse不够精简，与其设计原则相违背。Jailhouse与KVM也不同，KVM是作为Linux的一个内核模块，与内核融为一体，而Jailhouse作为固件镜像加载，就像WiFi适配器(Wi-Fi adapters)加载固件(firmware blobs)一样，在Linux执行引导程序前在指定的内存区域为Jailhouse预留一段专有内存。Jailhouse的内核模块(jailhouse.ko)加载固件并创建/dev/jailhouse设备文件，用户空间会使用到该设备文件，但该设备文件不做任何虚拟化。除了Root Cell的其它Cell统一称为Non-root Cell，与其它Hypervisor的客户机概念类似。Jailhouse主要由三个部分组成：内核模块，hypervisor固件以及一些工具，这些被用来使能hypervisor、创建Cell虚拟机，加载系统镜像，运行客户系统或者停止运行客户系统。

3.4.2 Jailhouse启动流程

Jailhouse逻辑上的启动流程如图3-10所示，Jailhouse利用Linux初始化硬件并进行启动，Linux启动完成后，通过装载内核模块jailhosuse.ko，将jailhosue代码拷贝到内核预留的指定内存位置，让所有CPU都执行Jailhouse Hypervisor的引导程序，并且保存当前Linux执行现场，进入EL2异常级别，配置Hypervisor的内存空间，建立页表映射，初始化中断设备，配置中断表，最后该程序初始化Root Cell，并且恢复之前Linux的执行现场，将Linux运行到Root cell内，到此Jailhouse启动完成，可以进行Non-Root Cell的创建，当前Root Cell里的Linux系统拥有当前所有的硬件资源。

创建新的Cell需要用.c配置文件描述该Cell需要的硬件资源，该描述文件通过编译后得到.cell二进制文件。当Jailhouse需要创建新的Cell时，则需要根据.cell配置文件从Root cell内划分一部分资源出来供新的Cell使用，比如说当创建一个新Cell时，Linux 发起一个ioctl调用(JAILHOUSE\_CELL\_CREATE)，会引起Linux内核调用jailhouse\_cell\_create()函数，读客户机硬件资源配置以及系统镜像，Hypervisor调用cpu\_down()函数停止部分CPU的运行并将其分配新的Cell，最后发起一个hypercall调用，将描述新Cell的数据结构指针传给Hypervisor，这个过程称为shrinking。



**图3-10 Jailhouse启动流程**

3.4.3 Jailhouse/ARMv8[20]

华为公司也参与了Jailhouse项目，并且是其主要贡献者之一，2016年华为公司ERC Munich团队将Jailhouse移植到了ARMv8架构下。在ARMv8架构下，Jailhouse hypervisor运行在EL2异常级别，Cell里的内核系统运行在EL1权限级别，Cell里的用户应用运行在EL0异常级。

在ARMv8平台上Jailhouse利用中断控制器GICv3来管理中断[21]，当一个中断到达时，不是直接给客户机，而是先到达EL2级的Jailhouse hypervisor，然后通过GICv3的硬件虚拟化支持，向客户机以虚拟中断的形式将中断转发给客户机，这个机制给Jailhouse的中断带来了一定的延迟。

另外，Jailhouse利用ARMv8的两段翻译技术来做内存隔离，Jailhouse使用stage2页表，使得Cell难以攻击Hypervisor。

第四章 车载虚拟化选型

4.1 选型指导

首先，以开源为基础，只在开源的虚拟化系统中寻找车载解决方案。因为开源系统更容易理解其设计架构、处理逻辑，让开发者更容易针对自己特定的需求来进行系统移植或者改动，以满足特殊需求，开发者可以自己定制软件，修改代码实现自己想要的功能。开源方案还可以节约成本，除了购买软件的成本，还可以节约防病毒、系统持续升级等费用；开源软件更安全，所有开发者和用户都可以看到开源软件的具体实现，也更容易发现系统中的漏洞，Linux之父Linus Torvalds说：“曝光足够，所有的Bug都是很容易发现的。”开源软件的漏洞可以尽可能地被发现然后迅速被解决。

除了开源以外，还要考虑技术的成熟度。开源软件背后如果没有成熟的技术团队或者大公司支持，其代码质量可能堪忧，会有不少安全问题，甚至性能不好。另外成熟开源方案具有更好的代码结构，可以减少重复非必要开发，更好地进行代码复用。

支持ARM平台架构也是非常必要的，现在车载平台绝大多数都是使用的是低功耗的ARM架构平台，如果虚拟化框架不支持ARM架构，其移植工作量是非常巨大的。初期之外，最好能支持ARM平台丰富的生态，ARM平台外设非常之多，如果都通过自己开发设备驱动，那也是非常巨大的工程量。

4.2 ARM虚拟化方案比较

根据3.1节的介绍，KVM是Type1型的hypervisor，依赖于宿主机，而Xen是Type1型的，直接运行与裸机上，同时支持全虚拟化和半虚拟化，一般来说Type2型的KVM虚拟化开销要比Type1型的Xen更高，比较难以满足汽车中某些软件的实时需求。况且，KVM在ARMv8下的运行模式是分离模式(split mode)，其highvisor与虚拟机之间的切换设计多次上下文切换，在ARMv8架构下的性能开销非常巨大，如下表所示，在ARM架构的HP Moonshot m400和x86架构的Dell PowerEdge r320上(具体配置见附录)，对Xen和KVM分别做无任何操作的Hypercall操作所需要的cycles开销。可以看出在ARMv8架构下KVM hyperecall的开销远远大于Xen的开销，非常不适合嵌入式虚拟化方案。针对KVM这个问题，哥伦比亚大学操作系统团队在2017年的系统会议ATC发表了与该问题相关的论文，该论文提出了一种VHE(Virtualization Hosted Extension)架构，该架构已经被ARM官方采用到了ARMv8.1架构下，VHE提出一种设计，扩展EL2硬件，让Linux可以运行在EL2异常级别，这样就可以减少大量无必要的上下文切换，但是遗憾的是，ARMv8.1架构现在为止只是一种架构，并没真正的实现，没有真正的生产出AMRv8.1架构的单板。

**表4-1 KVM和Xen hypercall调用开销[15]**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CPU Clock Cycles | ARM | | X86 | |
| KVM | Xen | KVM | Xen |
| Hypercall | 6500 | 376 | 1300 | 1228 |

Xvisor从设计上来说是非常适合车载虚拟化方案的，与Xen和KVM都不同，它是type1型的完全宏内核的Hypervisor，它将所有的IO设备实现都在Hypervisor层完成，hypervisor直接处理IO事件然后返还给虚拟机，这种设计在IO设备方面性能要优于Xen，Xen的IO需要通过Dom0内的Linux进行处理，中间有很多不必要的开销。但是Xvisor有一个致命的缺点，Xvisor所支持的ARM单板非常有限，本项目准备的imx8qm设备Xvisor也不支持，并且Xvisor的支持的设备也很有限，虽然Xvisor做了Linux头文件的兼容，方便从Linux移植设备驱动，但移植大量的设备驱动也是很大的工作量。

所以综上所述，目前支持ARMv8架构的虚拟化框架中，Xen和Jailhouse是比较适合车载虚拟化的解决方案，该项目目前也是主要围绕这两个接解决方案进行开发。

第五章 实验结果与分析

本文实验主要测试Jailhouse与Xen在ARMv8平台上运行时的性能，将其数据与Linux直接运行在硬件平台上的数据进行对比，并对测试结果进行一定的解释分析。

5.1 实验环境

**表5-1 实验环境**

|  |  |
| --- | --- |
| 主机型号 | NXP® i.MX 8QuadMax (i.MX 8QM) |
| 处理器型号 | 2x Arm® Cortex-A72, 4x Cortex-A53, 2x Cortex-M4 |
| 操作系统 | Linux 4.14.98 |
| 内存 | 2GB |
| SD卡 | 32GB |
| 交叉编译器 | aarch64-linux-gnu |
| CPU数量 | 2 |
| Xen |  |

5.2 系统基准测试

5.2.1 CPU性能测试

采用Unixbench 5.1.3[22]对Xen Dom0、Xen DomU、Jailhouse以及Linux裸机运行进行系统基准测试，所运行的操作系统都为Linux 4.14.98版本。

Dhrystone是常见的测量CPU运算能力的基准测试程序之一，Dhrystone的测试标准也很简单，该测试主要聚焦与字符串的处理，关键标准就是在单位时间内运行了多少个Dhrystone。总共测试7个样本，每个样本测试10秒，最后取平均值得到单位时间处理次数。

除了Dhrystone，Double-Precision Whetstone测试也可以对CPU性能进行测试，与Dhrystone不同的是，这一测试主要聚焦于浮点数操作的速度和效率，测试包括多个模块，每个模块有一组科学计算的操作，比如：sin、cos、sqrt、exp、log等等，同时测试了整数和浮点数运算。与Dhrystone类似，Double-Precision Whetstone也是运行7次，每次运行10秒，最后测试结果取平均值。

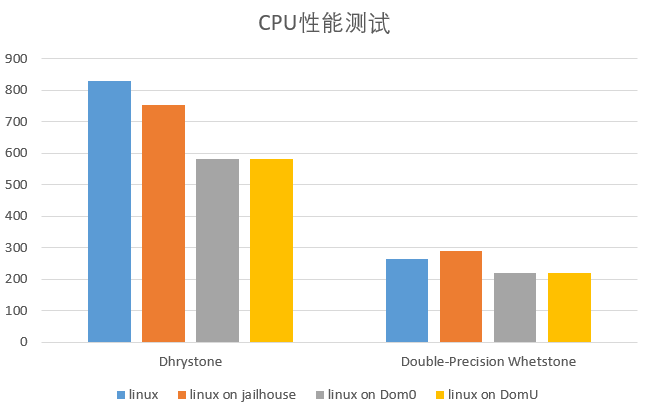
表5-2是Dhrystone测试结果，其中Result项是真实运算结果，Index项是与一个基准值比较换算得到的分数。表5-3是Double-Precision Whetstone的测试数据。Dhrystone的基准值为116700.0，而Double-Precision Whetstone测试的基准值为55.0。图5-1是两项测试直方统计图，两项数据都去经过换算后的Index数据。Dhrystone数据结果单位为lps(loop per second)，即每秒循环次数；Whetstone数据单位为MWIPS(Millions of Whetstone Instructions Per Second)，即每秒百万指令数

**表5-2 Dhrystone测试数据**

|  |  |  |
| --- | --- | --- |
| 测试环境 | Result(lps) | Index |
| Linux | 9669728.9 | 828.6 |
| Jailhouse Root Cell | 8797398.2 | 753.8 |
| Xen Dom0 | 6793731.9 | 582.2 |
| Xen DomU | 6793826.4 | 582.2 |

**表5-3 Double-Precision Whetstone测试数据**

|  |  |  |
| --- | --- | --- |
| 测试环境 | Result(MWIPS) | Index |
| Linux | 1456.4 | 264.8 |
| Jailhouse Root Cell | 1457.9 | 265.1 |
| Xen Dom0 | 1217.2 | 221.3 |
| Xen DomU | 1215.2 | 220.9 |



**图5-1 CPU性能测试图**

5.2.2 文件IO测试

Unixbench的File copy测试主要测试read和write两个函数，每次测试运行30秒。该测试实现比较简单，先循环写入一个文件2秒，再从刚刚写入的文件读出2秒，将读出的数据循环写入到另一个文件，统计在规定时间内文件的读写复制的字符数。总共做了三次测试，分别设置不同的缓冲区大小(bufsize)1024、256、4096个字节，和三组不同的文件最大块数(maxblocks)：2000、500和8000。每次测试运行其次，每次执行30秒，取平均值。表5-4，5-5和5-6是不同参数值的测试数据，同样的Index项是与基准值换算后的分数，越高性能越好。图5-2是File copy测试的统计图展示。

**表5-4 File copy 1024 bufsize 2000 maxblocks**

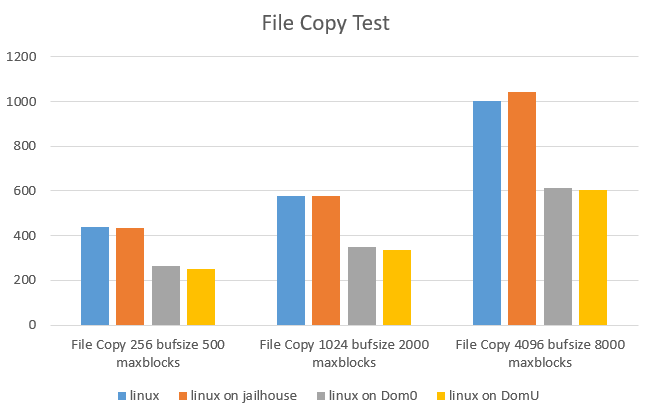
|  |  |  |
| --- | --- | --- |
| 测试环境 | Result(KBps) | Index |
| Linux | 228268.8 | 576.4 |
| Jailhouse Root Cell | 217117.8 | 548.3 |
| Xen Dom0 | 138514.4 | 349.8 |
| Xen DomU | 133092.4 | 336.1 |

**表5-4 File copy 256 bufsize 500 maxblocks**

|  |  |  |
| --- | --- | --- |
| 测试环境 | Result(KBps) | Index |
| Linux | 72642.1 | 438.9 |
| Jailhouse Root Cell | 69101.2 | 417.5 |
| Xen Dom0 | 43926.6 | 265.4 |
| Xen DomU | 41886.5 | 253.1 |

**表5-4 File copy 4096 bufsize 8000 maxblocks**

|  |  |  |
| --- | --- | --- |
| 测试环境 | Result(KBps) | Index |
| Linux | 580751.5 | 1001.3 |
| Jailhouse Root Cell | 606723.9 | 1046.1 |
| Xen Dom0 | 356295.8 | 614.3 |
| Xen DomU | 350814.0 | 604.9 |



**图5-2 文件读写性能测试**

5.2.3 其它基准测试

Execl Throughput测试，此测试项实际上是每秒对execl函数的调用次数，本实验中总共运行两次该测试，每次运行10秒，记录每次调用次数，取其平均值。

Pipe Throughput(管道吞吐)测试，该测试建立一个管道，向管道里写512Bytes数据，然后读出该数据，记录读写次数。本实验中该测试运行7次，每次都运行10秒，最后结果取其平均值。

Pipe-based Context Switching(基于管道的上下文切换)测试，该测试启动两个进程，进程间建立2个通信管道，1号进程向管道1写数据，从管道2读数据；2号进程从管道1读数据，向管道2写数据，每个进程完成一次读写，计数器加一，最后统计读写次数。本实验中运行该测试7次，每次运行10秒，最后取其平均值。

Process Creation测试，该项主要测试进程fork出子进程然后立马退出的次数，不停地Fork子进程然后退出，统计一段时间内的执行的次数。本实验中该测试运行两次，每次运行30s记录次数，取两次平均值。

Shell Scripts测试，该测试一分钟内不断地启动并停止shell脚本，统计次数。本实验中，该测试运行两次取其平均值。

System Call overhead(系统调用开销)测试，该测试通过系统调用，测试进入系统内核和离开系统内核所产生的开销，统计测试运行一段时间后系统成功调用的次数；在本实验中，该测试重复运行7次，每次运行10秒，取每次调用次数的平均值。

以下6个表为以上六项的测试具体数据和与基准值换算后的数据(Index)，图5-3为所有测试项的统计图。

**表5-5 Execl Throughput测试数据**

|  |  |  |
| --- | --- | --- |
| 测试环境 | Result(lps) | Index |
| Linux | 1327.7 | 308.8 |
| Jailhouse Root Cell | 1413.9 | 328.8 |
| Xen Dom0 | 1426.7 | 331.8 |
| Xen DomU | 1394.5 | 324.3 |

**表5-6 Pipe Throughput测试数据**

|  |  |  |
| --- | --- | --- |
| 测试环境 | Result(lps) | Index |
| Linux | 425510.7 | 342.1 |
| Jailhouse Root Cell | 371021.6 | 298.2 |
| Xen Dom0 | 359973.1 | 289.4 |
| Xen DomU | 322573.2 | 259.3 |

**表5-7 Pipe-based Context Switching测试数据**

|  |  |  |
| --- | --- | --- |
| 测试环境 | Result(lps) | Index |
| Linux | 54384.7 | 136.0 |
| Jailhouse Root Cell | 65613.8 | 164.0 |
| Xen Dom0 | 28617.2 | 71.5 |
| Xen DomU | 28846.9 | 72.1 |

**表5-8 Process Creation测试数据**

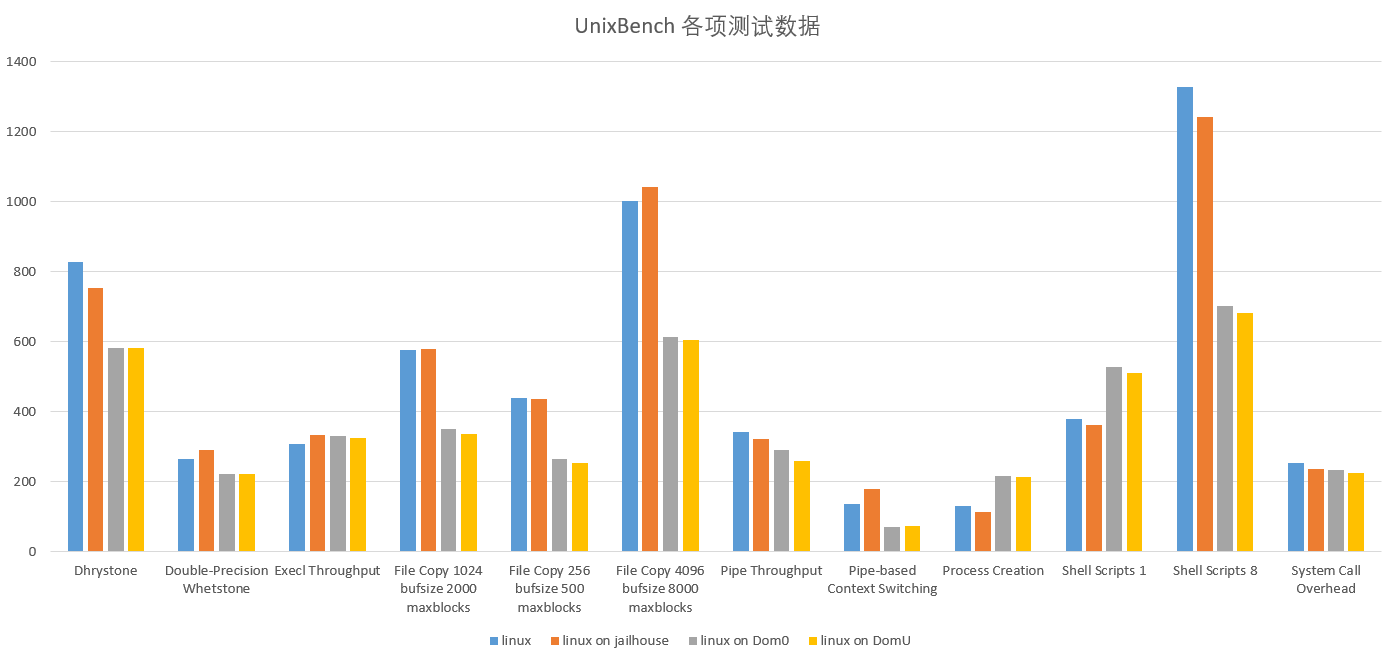
|  |  |  |
| --- | --- | --- |
| 测试环境 | Result(lps) | Index |
| Linux | 1646.6 | 130.7 |
| Jailhouse Root Cell | 1447.7 | 114.9 |
| Xen Dom0 | 2725.3 | 216.3 |
| Xen DomU | 2680.9 | 212.8 |

**表5-9 Shell Scripts测试数据**

|  |  |  |
| --- | --- | --- |
| 测试环境 | Result(lpm) | Index |
| Linux | 1601.9 | 377.8 |
| Jailhouse Root Cell | 1527.8 | 360.3 |
| Xen Dom0 | 2233.2 | 526.7 |
| Xen DomU | 2165.4 | 510.7 |

**表5-10 System Call overhead测试数据**

|  |  |  |
| --- | --- | --- |
| 测试环境 | Result(lps) | Index |
| Linux | 379835.4 | 253.2 |
| Jailhouse Root Cell | 348213.0 | 232.1 |
| Xen Dom0 | 351633.6 | 234.4 |
| Xen DomU | 335184.9 | 223.5 |



**图5-3 各项测试数据对比**

5.2.4 系统跑分计算

由于unixbench各项测试得分高低差异很大，所以需要用公式结合各项测试数据重新整理算出一个可以描述系统各方面性能够的分数。

计算思路先通过log将各项数据降维处理[22]，把不同测试的数据尽可能拉平差距，然后进行均值计算再阶乘得到一个描述系统性能的分数。公式5-1先降维求和，然后将结果进

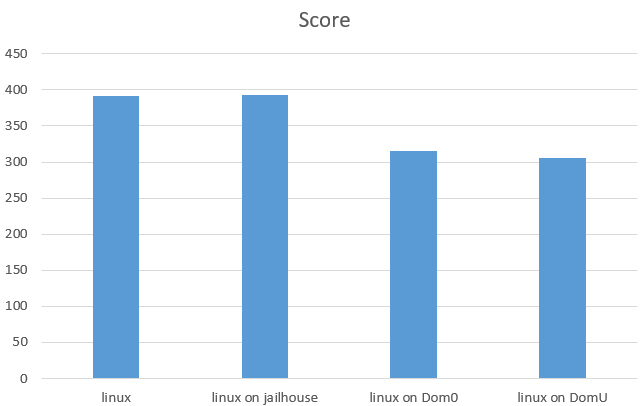
(5-1)

行阶乘得到最后的系统跑分，阶乘公式5-2。最后得到系统跑分表5-11和统计图5-4。

(5-2)

**表5-11 系统跑分数据**

|  |  |
| --- | --- |
| 测试环境 | 分数 |
| Linux | 392.2 |
| Jailhouse Root Cell | 380.4 |
| Xen Dom0 | 314.7 |
| Xen DomU | 305.7 |



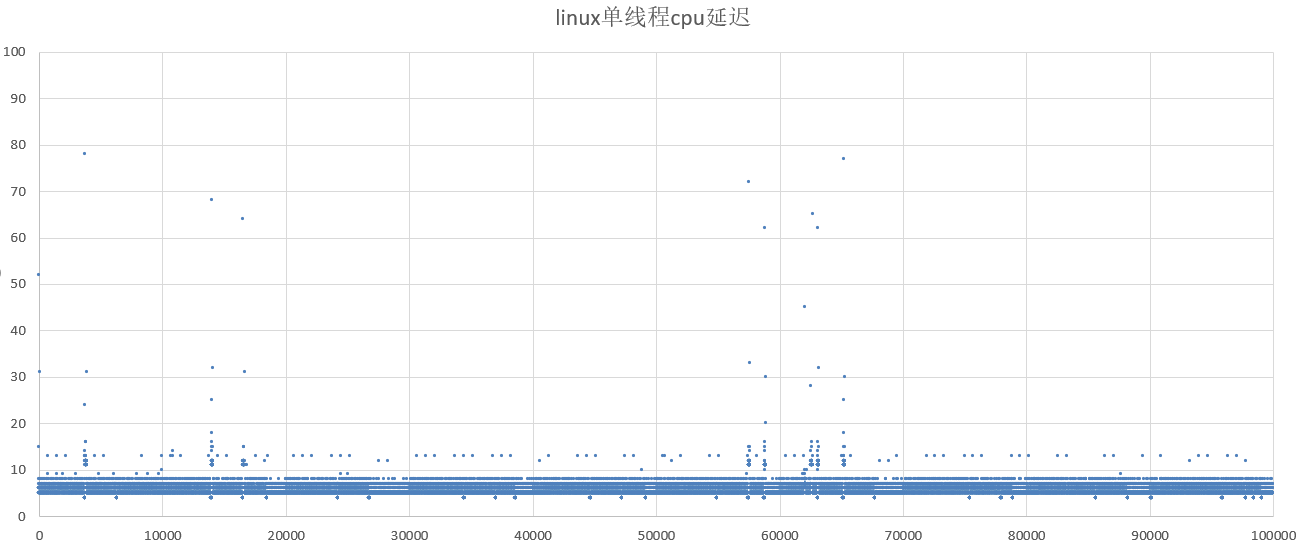
**图5-4 系统跑分统计**

5.3 实时性测试

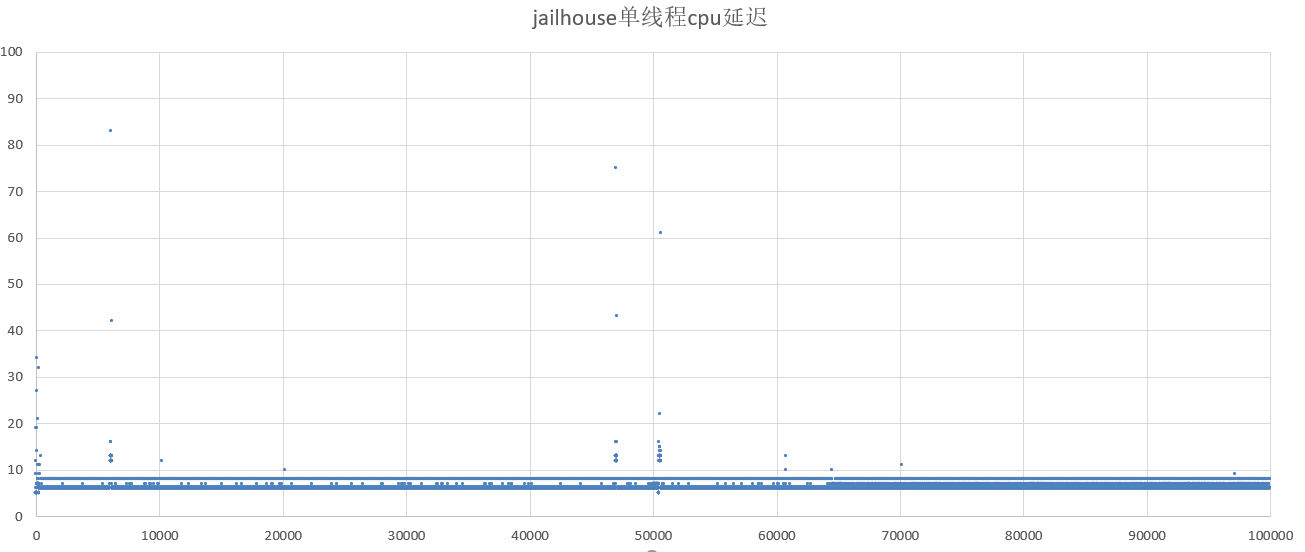
该部分测试主要使用Cyclictest来进行测试，Cyclictest是一个高精度测试程序，是rt-tests中使用最广泛的测试工具，一般用来测试系统内核的延迟，判断内核实时性，主要通过CPU响应线程处理时间来计算延迟，CPU响应越快，延迟越低。本实验分为两个部分，一个是单线程延迟测试，另一部分是引入多线程进行干扰，测试延迟及其延迟抖动。

5.3.1 单线程测试

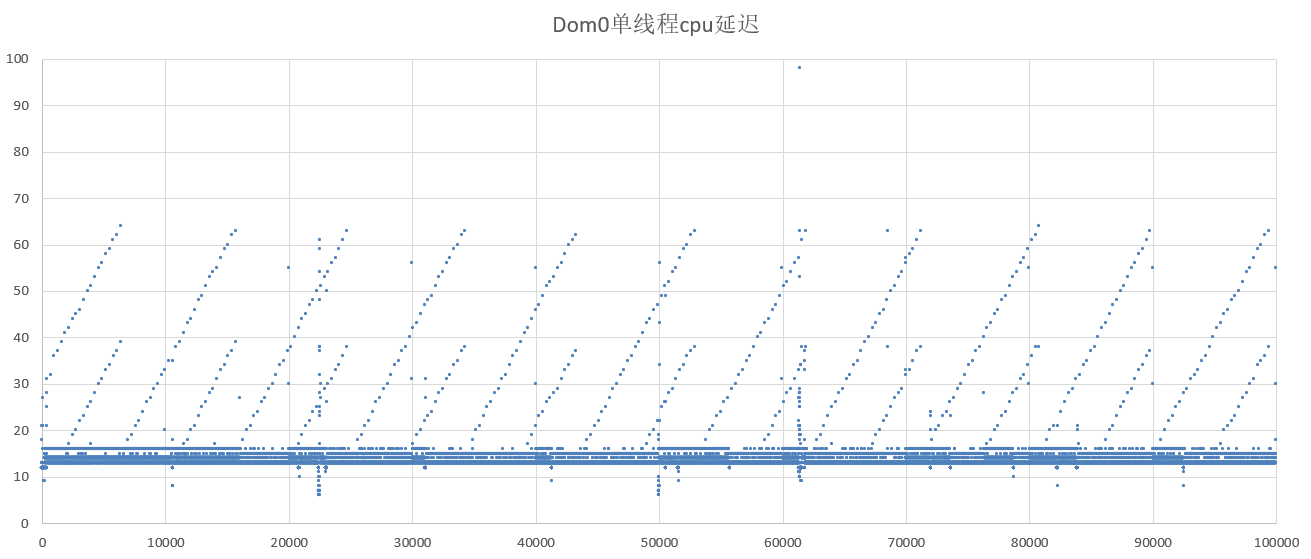
将线程优先级设置为95，进行10万次测试，每次循环睡眠100us。将测试结果重定向输出到某个文件，用python代码(见附录)提取出有效数据，并输出到Excel文件，用Excel工具进行统计。以下4张图分别为四种实验环境下单线程的延迟分布情况。



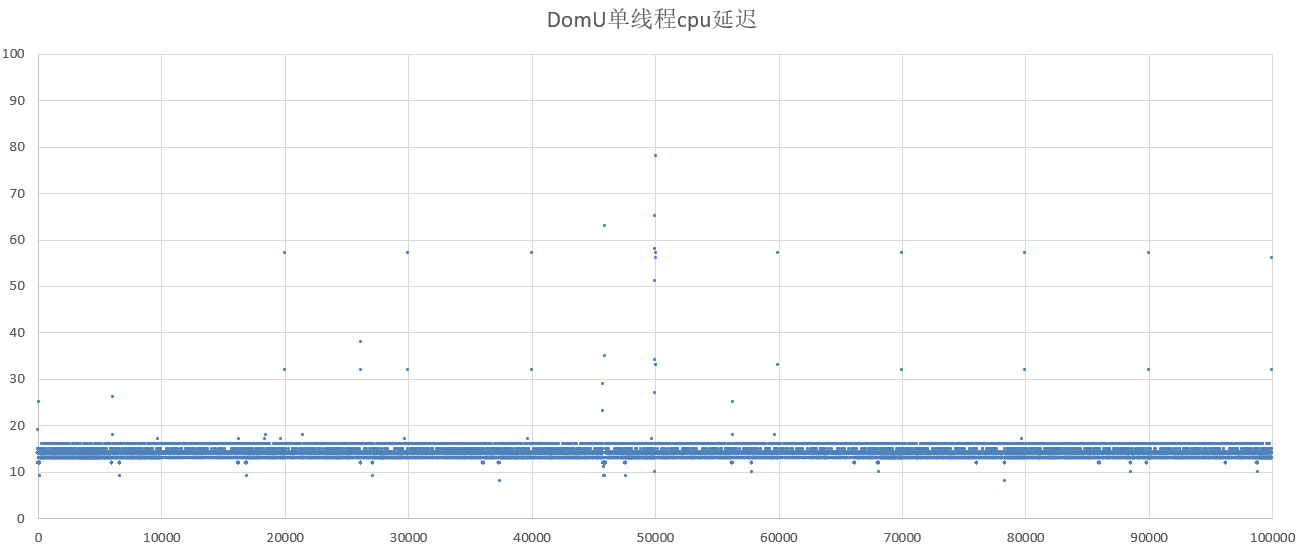
**图5-5 Linux单线程延迟**



**图5-6 Jailhouse单线程延迟**



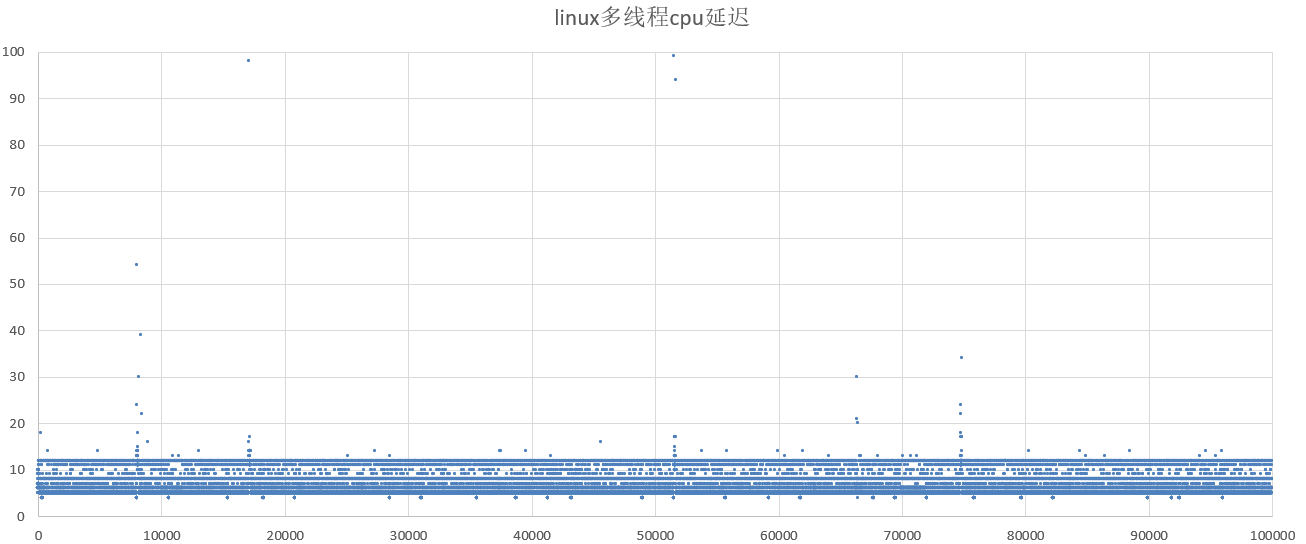
**图5-7 Dom0单线程延迟**



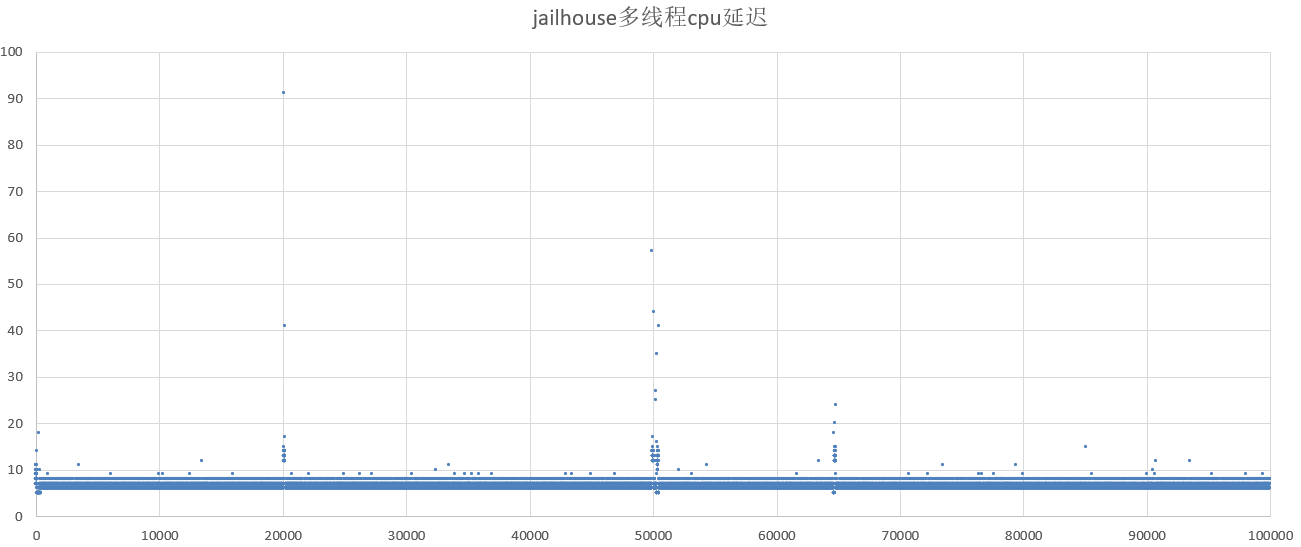
**图5-8 DomU单线程延迟**

5.3.2 多线程测试

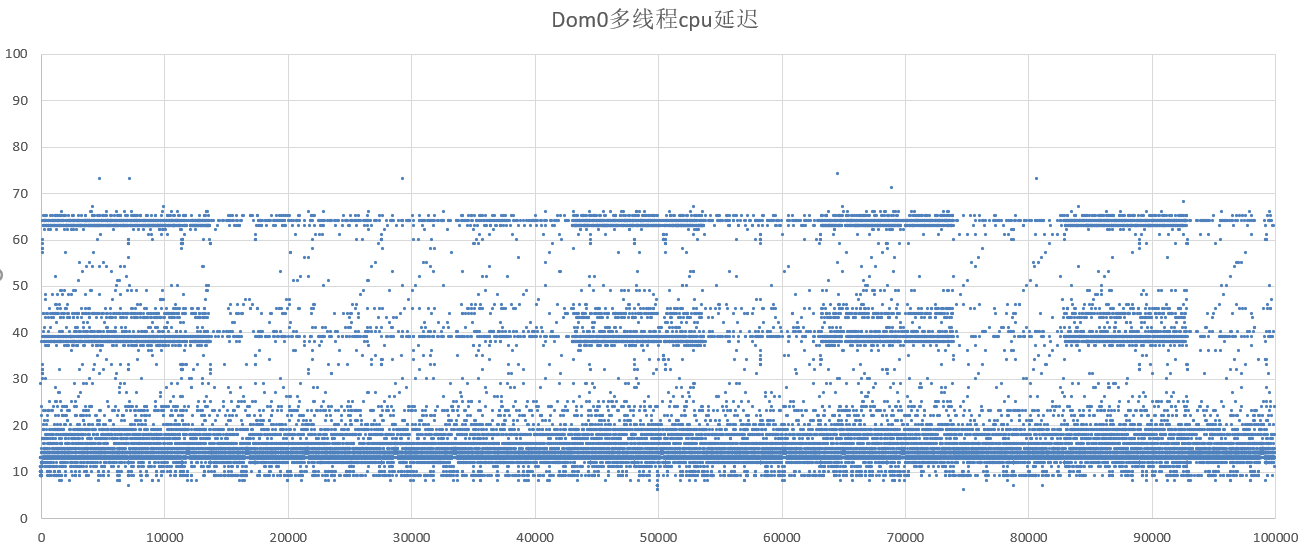
该部分测试引入另外19个同一优先级的线程进行干扰，对目标线程进行延迟和延迟抖动统计，与上一测试类似该测试运行10万次。以下4张图对应4种环境下的多线程干扰延迟测试。



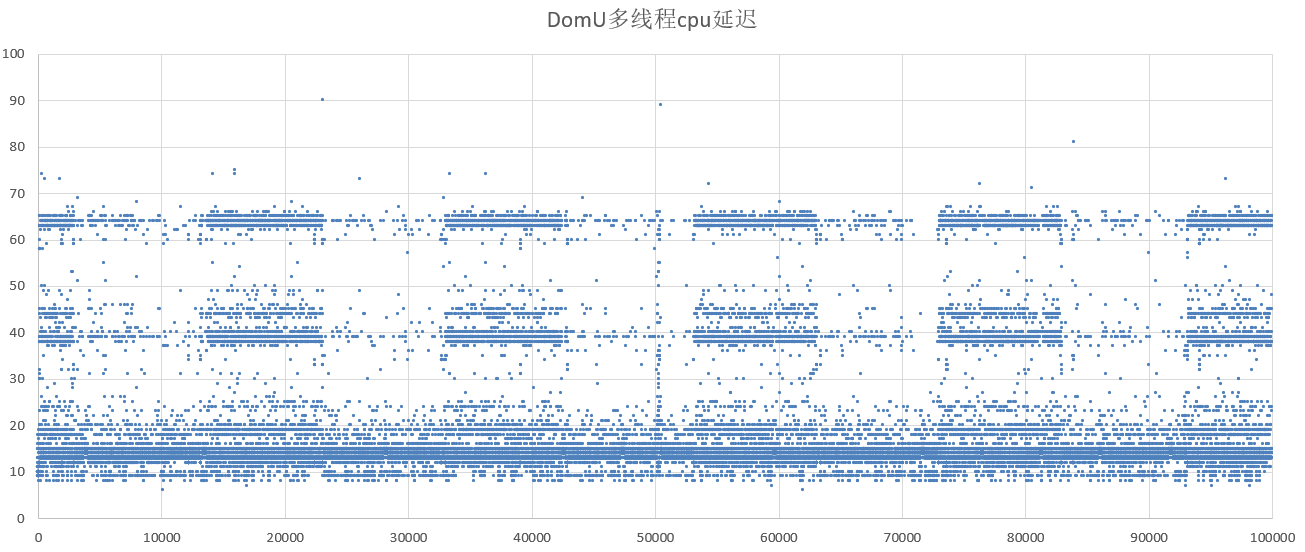
**图5-9 Linux多线程干扰测试**



**图5-10 Jailhouse多线程干扰测试**



**图5-11 Dom0多线程干扰测试**



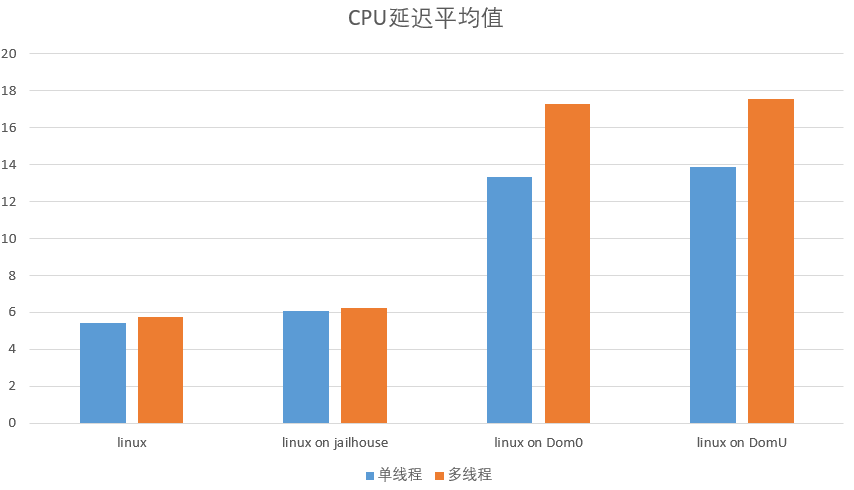
**图5-12 DomU多线程干扰测试**

5.3.3 平均延迟统计

对以上八组数据进行均值处理，得到图5-13统计图对比。

**表5-12 平均延迟数据**

|  |  |  |
| --- | --- | --- |
| 测试环境 | 单线程延迟(us) | 多线程延迟(us) |
| Linux | 5.44 | 5.74 |
| Jailhouse Root Cell | 6.09 | 6.22 |
| Xen Dom0 | 13.35 | 17.27 |
| Xen DomU | 13.88 | 17.56 |



**图5-13 平均延迟统计图**

5.4 实验结果分析

总体来说Linux裸机运行性能最好，Jailhouse Root Cell内运行的Linux系统总体性能接近于裸机Linux，其性能损耗仅仅3.01%，可忽略不计。而Xen Dom0和DomU内运行的相同版本Linux系统，其性能损耗比Jailhouse大，分别是19.76%和22.06%。尤其在CPU运算性能和文件读写操作上，其虚拟化性能损耗明显。

对内核延迟测试，内核延迟Linux、Jailhouse Root Cell、Xen Dom0和Xen DomU，内核平均延迟依次升高，Jailhouse Root Cell平均延迟非常接近于Linux直接运行，并且引入干扰线程后，其抖动也很小。而Dom0和DomU的延迟相对而言就比较高了，并且其它线程对延迟干扰较大，当引入其它同优先级线程进行干扰时，可由分布图看出，Dom0和DomU的延迟抖动都比较大，并且平均延迟也有较大增加，而Linux和Root Cell依然很稳定，只是平均延迟稍有增加。

综上，Jailhouse有接近Linux裸机运行的性能，其性能损耗在3%左右，并且CPU延迟和延迟抖动表现也很好，依然较接近于裸机运行。而Xen的性能损耗较高，接近20%左右，延迟抖动也比较大，其中Dom0的系统性能和延迟表现稍好与DomU。

第六章 总结与展望

汽车智能化的发展使得控制软件的复杂度和集成度不断提升，Hypervisor 的虚拟化方案可以在多核异构的单芯片上运行多个不同类型的操作系统，各系统间共享硬件资源，即是彼此独立又可交互信息。Hypervisor 即满足了日益复杂场景下的不同业务需求，又提高了硬件资源的使用效率，大幅降低了成本，更重要的是，虚拟化所具备的不同操作系统间的隔离能力，可以大大提升系统的可靠性和安全性。当前车载平台大多使用ARMv8架构，本文基于该设定，探索研究ARMv8平台的虚拟化方案，从中选出适合车载系统的方案做性能测试分析。

本文探索比较了ARMv8架构下的KVM、Xen、Xvisor以及Jailhouse四种虚拟化方案，其中KVM与Xvisor对于车载系统来说有比较致命的缺陷，KVM在ARMv8架构下以分离模式运行，其上下文切换开销极大，不适合车载场景对性能以及实时性的需求；而Xvisor是Type1型宏内核Hypervisor，从设计与实现上来说是比较适合车载需求的，但是Xvisor是一个比较年轻的项目，又是自己完成设备驱动，而在车载场景下有非常多种类的外设，如果将Xvisor应用到车载上，需要自己手动完成或者移植大量设备驱动，这需要非常大的工作量。所以本文选取了Jailhouse与Xen两个虚拟化方案进行性能测试分析，主要从系统整体性能和实时性两个方便对两者性能损耗进行测试评估。测试结果显示，Jaillhouse性能损耗较小，仅3%的性能损耗，有接近于Linux裸机运行的性能，其延迟表现也很好，延迟稳定，受干扰抖动较小。而Xen不论是Dom0还是DomU性能损耗都比Jailhouse高，且延迟也比较高，受干扰抖动较大。

从测试结果来看，Jailhouse更能满足车载场景需求，但Jailhouse是硬件静态分区的hypervisor，不能根据负载和优先级动态调度硬件资源，但Xen可以给虚拟机设置优先级，根据优先级来动态调度虚拟机，可以满足车载系统如果当某一应用出现突发状况急需更多硬件资源的需求。并且Xen的技术度更成熟，一般虚拟化工程师也对Xen比较熟悉，更容易在Xen框架下进行开发，如果能对Xen进行一些修改，通过半虚拟化方式运行系统，优化运行在Xen上的系统更好的性能，对调度算法保护实时应用，减少其延迟抖动，那么Xen会是比较好的车载虚拟化解决方案。

总而言之，Xen和Jailhouse都是比较适合车载虚拟化的解决方案，如果能对Xen进行一些改动，优化其性能和延迟抖动，Xen可能更适合车载场景。虚拟化在车载领域的应用是一个非常新的研究方向，也很适合车载系统，可以大大减少车载硬件的成本，提高硬件资源利用率，当今工业界和学术界几乎没有对该领域的研究，本文也只是根据车载系统软件的特点和需求，综合考虑系统各方面性能，对比分析现有的开源ARMv8架构下的虚拟化方案，为今后车载虚拟化平台产业化做好前期技术预研工作。

参考文献

[1] Hua, Zhichao, et al. vTZ: Virtualizing ARM TrustZone[J]. 26th USENIX Security Symposium (USENIX Security 17). 2017.

[2] Broy, Manfred. Challenges in automotive software engineering[J]. Proceedings of the 28th international conference on Software engineering, pp. 33-42. 2006.

[3] Li, Hao, Xuefei Xu, Jinkui Ren, and Yaozu Dong. ACRN: a big little hypervisor for IoT development[J]. Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 31-44. 2019.

[4] 英特尔开源软件技术中心, 复旦大学并行处理研究所. 系统虚拟化: 原理与实现[M]. 北京：清华大学出版社，2009.

[5] Arm Limited or its affiliates. Arm® Architecture Reference Manual[GB/T]. 2013-2020.

[6] <https://developer.arm.com/architectures/learn-the-architecture/armv8-a-virtualization/stage-2-translation>.

[7] Elisei, Alexandru, and Mihai Carabas. bhyvearm64: Generic Interrupt Controller Version 3 Virtualization. 2019.

[8] Lim, Jin Tack, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. NEVE: Nested virtualization extensions for ARM[J]. In Proceedings of the 26th Symposium on Operating Systems Principles, pp. 201-217. 2017.

[9] <https://developer.arm.com/architectures/learn-the-architecture/generic-timer/single-page>

[10] 任永杰，单海涛. KVM虚拟化技术：实战与原理解析[M]. 北京：机械工业出版社，2013.

[11] Dall, Christoffer, and Jason Nieh. KVM/ARM: the design and implementation of the linux ARM hypervisor[J]. Acm Sigplan Notices 49, no. 4 (2014): 333-348.

[12] R. Russell. virtio: Towards a De-Facto Standard for Virtual I/O Devices[J]. SIGOPS Operating Systems Review, 42(5):95–103, July 2008.

[13] Chisnall David. The Definitive Guide To the Xen Hypervisor[M]. 北京：北京航空航天大学出版社，2014.

[14] <https://segmentfault.com/a/1190000016152284>

[15] Dall, Christoffer, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. ARM virtualization: performance and architectural implications[J]. In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pp. 304-316. IEEE, 2016.

[16] Patel, Anup, Mai Daftedar, Mohamed Shalan, and M. Watheq El-Kharashi. Embedded hypervisor xvisor: A comparative analysis[J]. In 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 682-691. IEEE, 2015.

[17] <https://github.com/siemens/jailhouse/blob/master/Documentation/articles/LJ-article-04-2015.txt>

[18] Siemens Corporate Technology. Hard Partitioning for Linux: The Jailhouse Hypervisor[R]. 2015.

[19] Diana Ramos. Exploring IVSHMEM in the Jailhouse Hypervisor[D]. CISTER Research Centre. 2019.

[20] Huawei Technologies Duesseldorf GmbH. ARMv8 port of the Jailhouse hypervisor[R]. 2016.

[21] Ramsauer, Ralf, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look mum, no VM exits!(almost)[M]. arXiv preprint arXiv:1705.06932 (2017).

[22] <https://www.alibabacloud.com/blog/unixbench-score-an-introduction_594677>

[23] <http://manpages.ubuntu.com/manpages/cosmic/man8/cyclictest.8.html>

谢辞

在完成毕业设计的过程和我的整个大学四年中，有很多人对我的学习和生活都有莫大的帮助，在此借毕设论文谢辞由衷地感谢一下各位。

首先感谢我的毕设导师戚正伟教授，毕业设计的立项、开题、中期检查和论文撰写都离不开戚老师的悉心指导，每周进行的组会，也大大拓展了我对该领域的深入认识，提高了我阅读学术论文的能力，让我能更好地理解我的毕业设计，有更好的理论基础去撰写论文。

感谢刘焰强博士学长、张正君同学和小米的工程师团队。刘焰强学长是我在该项目中的直接代领人，从我参与该项目到论文撰写，刘焰强学长都有很多建设性指导，尤其在做实验方面，嵌入式实验环境搭建比较困难，有幸借刘焰强学长的经验才能顺利进行实验测试。同时，参与该项目的华中科技大学的张正君同学也给了我很多帮助，我们经常会交流项目经验，遇到问题互相交流借鉴经验。另外感谢小米工程师团队，该项目是与小米团队合作开发的，本论文只涉及其中一部分，而其它部分则是由小米团队进行开发，项目初期长期和小米工程师一起，感谢工程师程沛、曹武国和沈金华的照顾与监督。

感谢ICS和操作系统上课老师，夏虞斌老师、臧斌宇老师和陈榕老师，感谢他们在课堂上的精彩讲解，让我对系统和虚拟化领域有了较为深入的了解，培养了对这个领域的兴趣，也使得我从事这方面的研究。

感谢我的父母、姐姐和家人、朋友和同学，在完成毕设这段期间我还要准备春季招聘和考研复试准备，期间有很多困难和挑战、烦恼和忧虑，是他们的支持和关爱给了我动力和信心。

感谢我的室友和同学大学四年一起陪我度过。来的时候满怀期待，走的时候满怀留恋，虽然这四年也有不如意，但也是人生经历中的唯一。

**SELECTION OF VIRTUALIZATION HYPERVISOR IN AUTOMOTIVE AND PERFORMANCE ANALYSIS**