上海交通大學

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，软件伴随了 4 代汽车的发展。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隔离能力，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 automotives. 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 automotive 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 automotive 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

# 目录

第一章 绪论.....1
1.1 课题意义与背景.....1
1.2 相关工作.....1
1.3 本文工作.....2
1.4 本章小结.....2
第二章 技术背景.....3
2.1 虚拟化技术[4].....3
2.1.1 完全虚拟化技术.....3
2.1.2 硬件辅助虚拟化.....4
2.1.3 类虚拟化技术.....5
2.2 ARMv8 架构[5].....6
2.2.1 AArch64 Exception Level.....6
2.2.2 ARMv8 TrustZone 技术.....7
2.2.3 ARMv8 虚拟化技术.....7
2.3 本章小结.....11
第三章 ARM 虚拟化方案对比分析.....12
3.1 KVM/ARM 架构分析.....12
3.2 Xen/ARM 架构分析.....16
3.3 Xvisor/ARM 架构分析.....18
3.4 Jailhouse.....19
3.4.1 Jailhouse/ARM 架构分析.....19
3.4.2 Jailhouse 启动流程研究.....20
3.5 本章小结.....21
第四章 方案选型与测试设计.....22
4.1 选型准则.....22
4.2 ARM 虚拟化方案比较.....22
4.3 测试设计与实现.....23
4.3.1 基准测试设计与实现.....24
4.3.2 其它测试设计与实现.....25
4.4 虚拟化平台实现与工具移植.....27
4.4.1 Xen 移植.....27
4.4.2 Jailhouse 移植.....29
4.4.2 工具移植.....30
第五章 实验结果与分析.....31
5.1 实验环境.....31
5.2 系统基准测试.....31
5.2.1 CPU 性能测试.....31
5.2.2 文件 IO 测试.....32
5.2.3 内存测试.....35
5.2.4 其它基准测试.....37
5.2.5 系统跑分计算.....39

5.3 实时性测试.....39
5.3.1 单线程延迟测试.....39
5.3.2 多线程延迟测试.....41
5.3.3 网络延迟测试.....43
5.3.4 核间中断延迟测试.....44
5.4 实验结果分析.....45
第六章 总结与展望.....47
参考文献.....48
谢辞.....50

# 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。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。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需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，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。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，因此软件成为成本的主导因素。

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分配到多组虚拟机上，并将这些资源抽象为虚拟硬件资源。每个虚拟机可以运行不同的操作系统，虚拟机可访问对其分配的虚拟硬件资源，不同的系统运行在相互独立的空间内，互不影响。

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，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，既是彼此独立又可交互信息。虚拟化技术既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，现有的商业产品和开源产品，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。2015 年黑莓公司推出了 QNX 1.0，Type-I 型的虚拟系统，其应用范围广泛，包括工业医疗设备、自动化以及汽车应用程序等领域；汽车应用包括车载娱乐软件、驾驶员辅助系统和数字仪表盘等。QNX 一个计算机平台上运行多个操作系统，减少了产品的尺寸、功耗和重量，大大节约了成本。QNX 可以将安全软件和非安全类型的软件隔离开来。在一个操作系统上运行安全软件，用另一个操作系统运行非安全的软件。

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 的优势是非常的轻量级，它的第一版本只有 2.5 万行代码左右。ACRN 有两个关键部分：hypervisor 和 ACRN 设备模块。ACRN 是一个 Type 1 型虚拟化系统，直接裸机运行。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 QNX Hypervisor，实际上是由一个 RTOS（实时系统）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 型的虚拟化平台，综合考虑实时响应和各方面系统性能，并在其上运行两个以上的异构操作系统，例如 QNX+Linux+AUTOSAR，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。相关研究的数据也显示，到 2005 年为止，汽车的电子成本只有 20%左右，而现在汽车电子成本超过了总成本的 50%。随着汽车集成电子元件越来越多，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，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于仿真技术。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”多个系统。广义地来说，将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。现在的操作系统其实都应用了虚拟化技术，比如操作系统的进程概念和虚拟内存地址。虚拟内存就是物理内存的抽象，每个虚拟内存空间都有自己单独的一套页表和内存地址，虚地址范围可以与实地址的范围不相同，可大可小。除此之外，进程其实也可以说是虚拟化技术的一个应用，每一个进程都认为自己独占整套计算机资源，并且进程与进程之间是相互隔离开来的。

操作系统的进程概念是进程级虚拟化，而虚拟机则是另一个级别的虚拟化技术—系统级虚拟化，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。在虚拟化技术可以在一台物理计算机上模拟出多个虚拟执行环境，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 架构的操作系统。但是在物理机上模拟不同指令架构的虚拟机需要模拟执行每一条指令，因为不同架构的指令不可以直接在物理机上运行，这样会引入较大的性能开销。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。用软件实现的完全虚拟化就是利用了模拟执行方案，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。

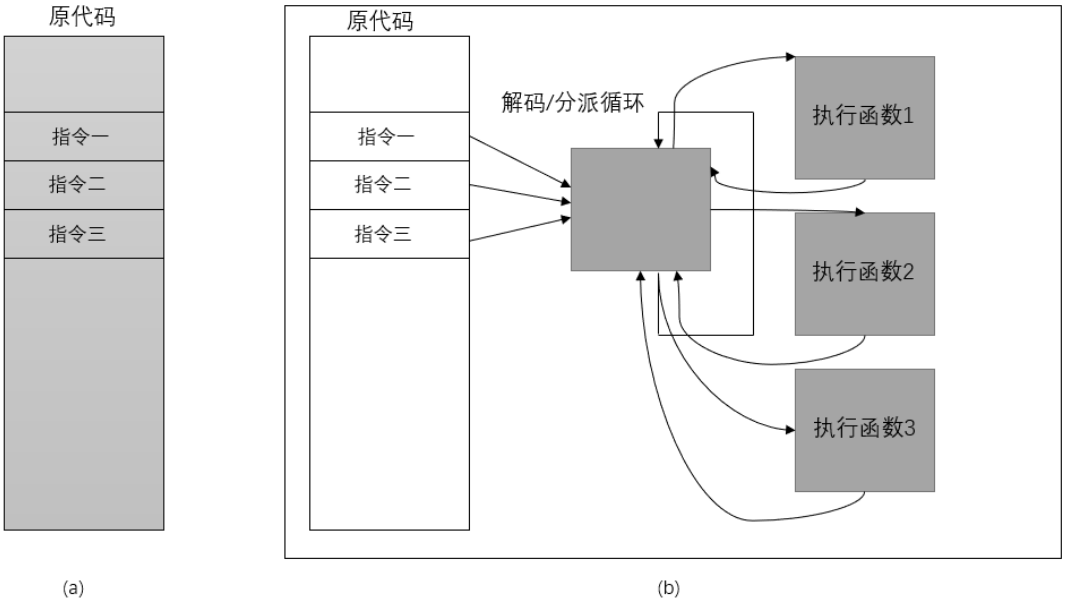


**图 2-1　正常执行与解释执行**

**Figure 2-1 Normal Execution and Interpretation execution**

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 会扫描并检查这段代码，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高效的虚拟化。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。对于支持虚拟 CPU，英特尔公司设计了 VT-x(Intel Virtualization technology for x86)技术；英特尔公司提供了 EPT(Extended Page table)硬件支持内存虚拟化；英特尔公司设计实现了 VT-d 技术支持外设虚拟化。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。通常来说在虚拟化环境下指令的执行需要先陷入，然后再进行模拟执行，但是 IA32 有十九条敏感指令的执行不会导致陷入，所以也无法进行陷入模拟，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，重新定义了敏感指令，使得这些敏感指令要么通过陷入模拟执行，要么重定义为可直接执行；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。

内存虚拟化主要的任务是完成两段地址翻译，虚拟机的虚拟地址到虚拟机物理地址，然后将中间物理地址翻译为物理机物理地址，即 GVA 翻译为 GPA，GPA 翻译为 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，但是开发影子页表工作量比较大，其维护工作更是复杂，并且每个虚拟机的每个进程都需要维护一套影子页表，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地址，VT-d 可以重映射外设的 DMA。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。

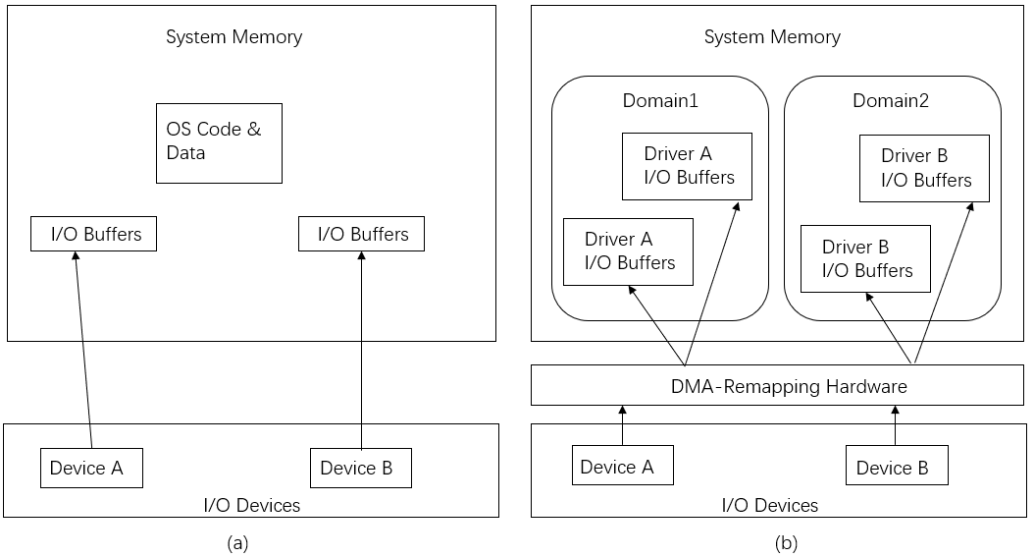


**图 2-2 VT-d 访问内存架构**

**Figure 2-2 Access Memory Based on VT-d**

### 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，为了操作系统课正常运行，需要修改系统的代码去适应半虚拟化，或者修改提供给虚拟机的虚拟硬件接口，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，尤其如果操作系统开发人员停止了对该版本的维护，为这个系统实现虚拟化支持也是非常难的。

## 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 推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。

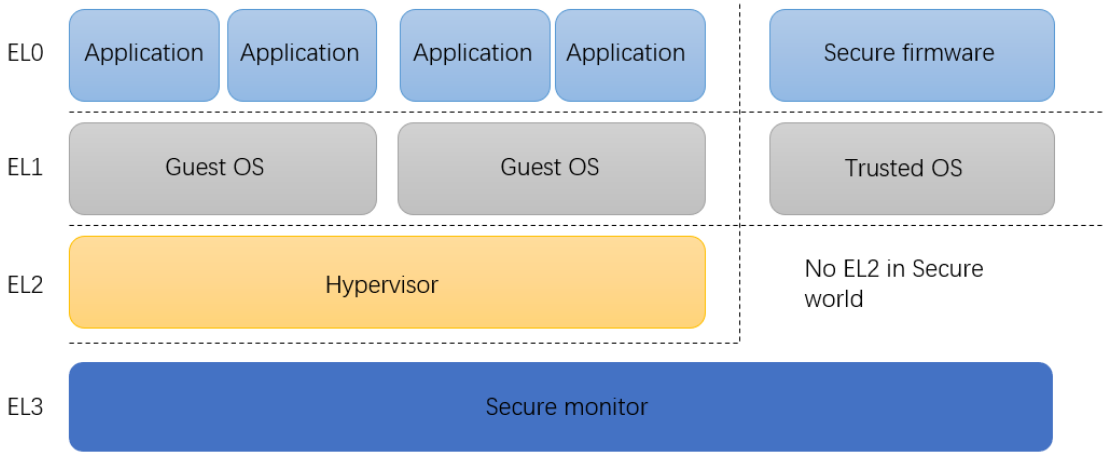


**图 2-3 ARMv8 异常级架构**

**Figure 2-3 ARMv8 Exception Level**

### 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 体系结构一开始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。

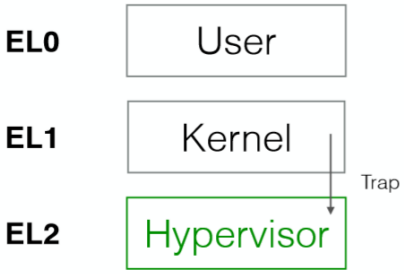


图 2-4 陷入模拟

Figure 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。

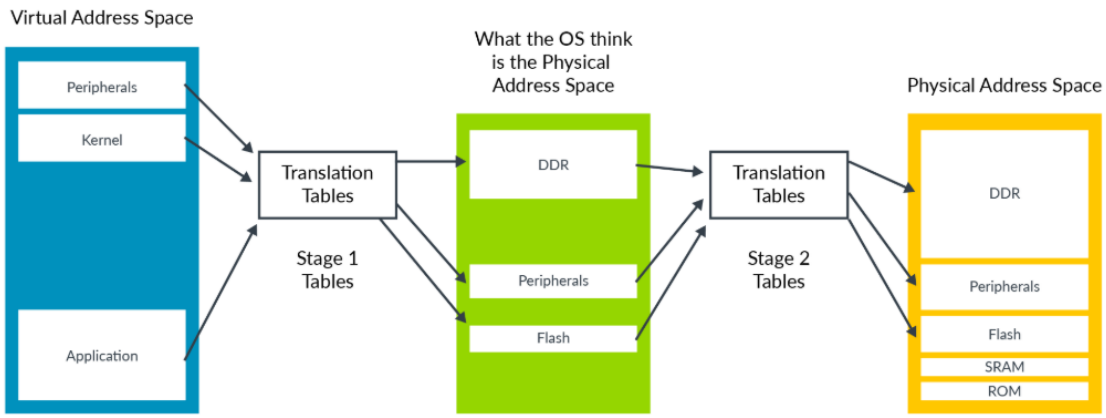

图 2-5 ARMv8 两段内存翻译[6]

Figure 2-5 ARMv8 Stage-2 Translation

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。

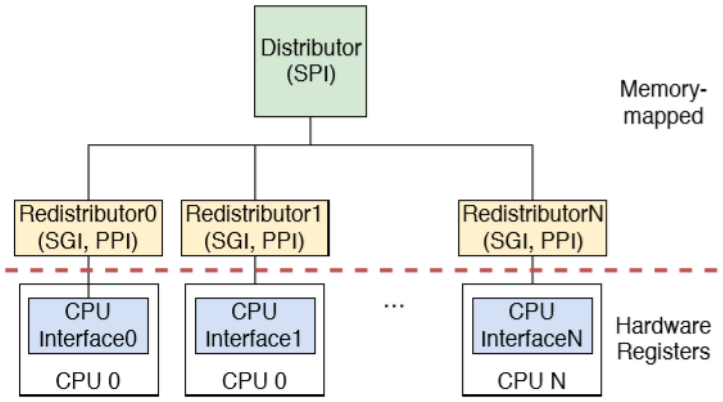


图 2-6 GICv3 架构图

Figure 2-6 GICv3 Architecture

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。

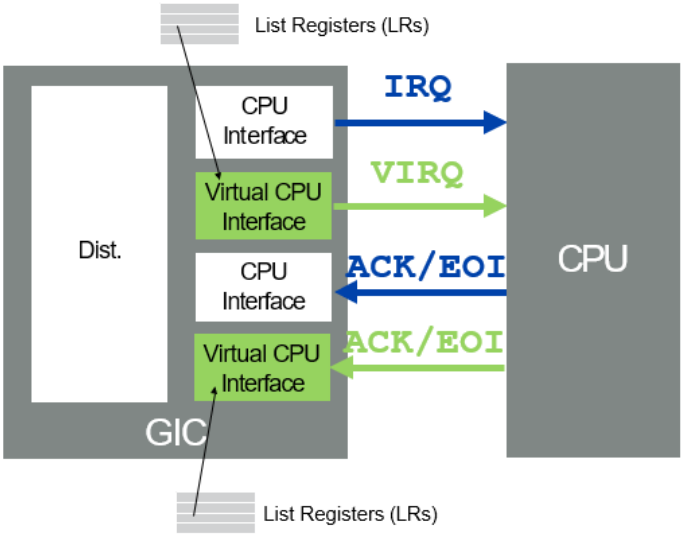


图 2-7 GICv3 虚拟中断[8]

Figure 2-7 GICv3 Virtual Interrupt

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 2-8 所示，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。

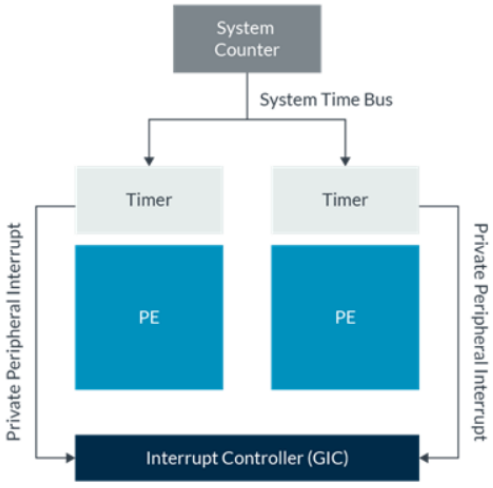


图 2-8 ARM 时钟架构[9]

Figure 2-8 ARM timer

## 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 虚拟化方案对比分析

## 3.1 KVM/ARM 架构分析

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。

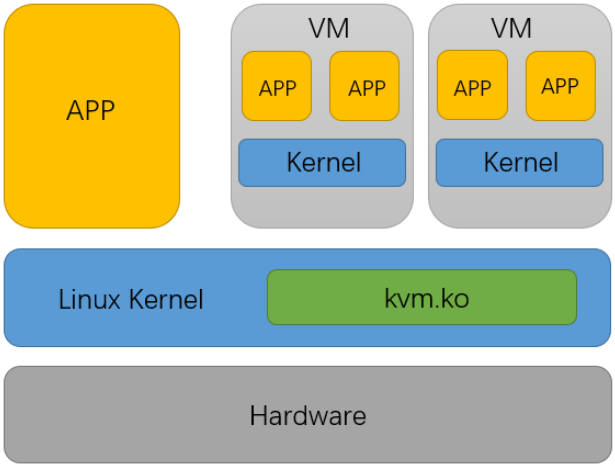


图 3-1 KVM 架构图

Figure 3-1 KVM Architecture

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。

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。

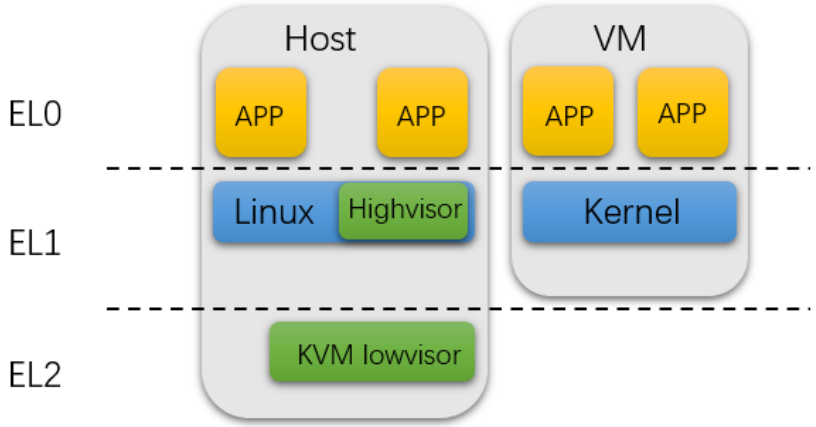


图 3-2 ARMv8 下的 KVM 架构图

Figure 3-2 KVM on ARMv8

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。

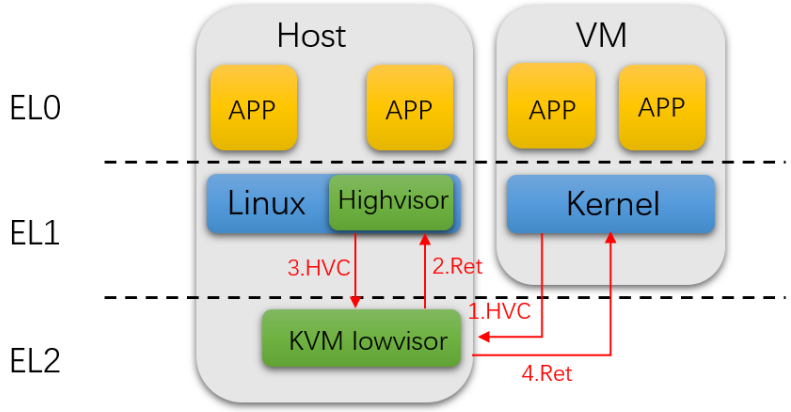


图 3-3 KVM/ARMv8 Hypercall 陷入处理

Figure 3-3 KVM/ARMv8 Trap Hypercall

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。

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。

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 架构分析

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。

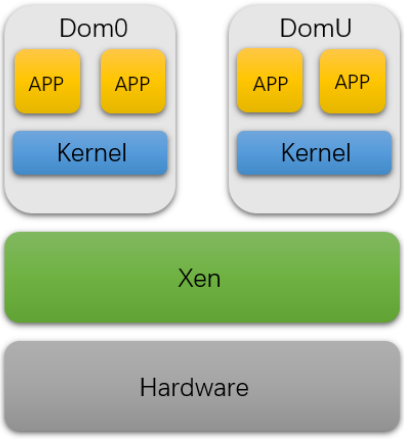


图 3-4 Xen 架构

Figure 3-4 Xen Architecture

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 设备写入时，

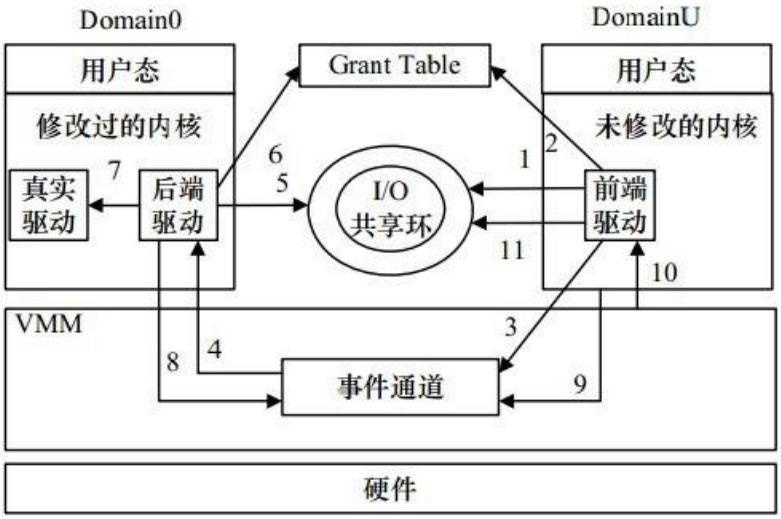


图 3-5 Xen I/O 设备流程图[14]

Figure 3-5 I/O Process on Xen

(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。

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。

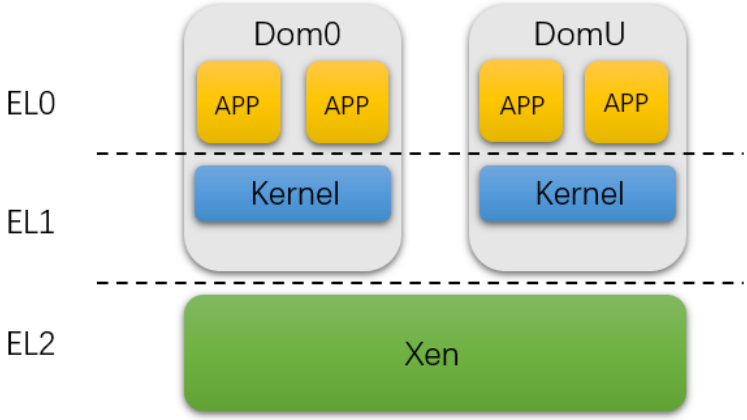


图 3-6 Xen/ARM 架构图

Figure 3-6 Xen/ARM Architecture

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 架构分析

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。

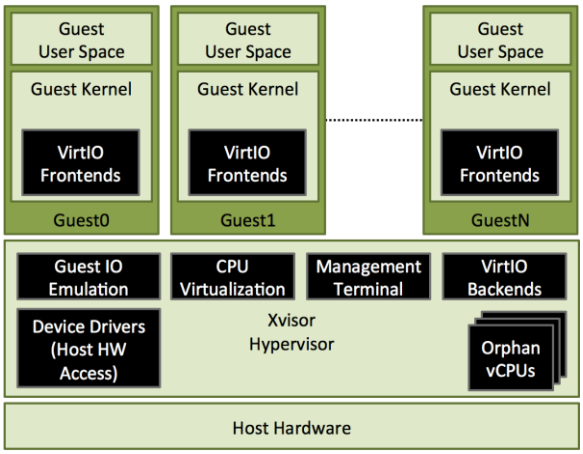


图 3-7 Xvisor 架构[16]

Figure 3-7 Xvisor Architecture

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。

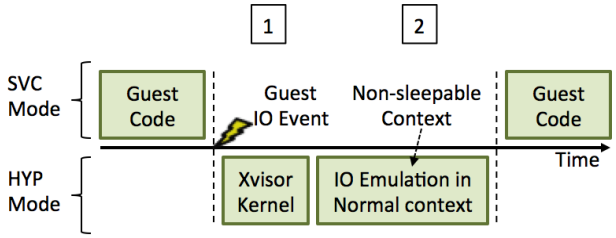


图 3-8 Xivsor 客户机 IO 事件[16]

Figure 3-8 VM IO Processing on Xvisor

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/ARM 架构分析

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。

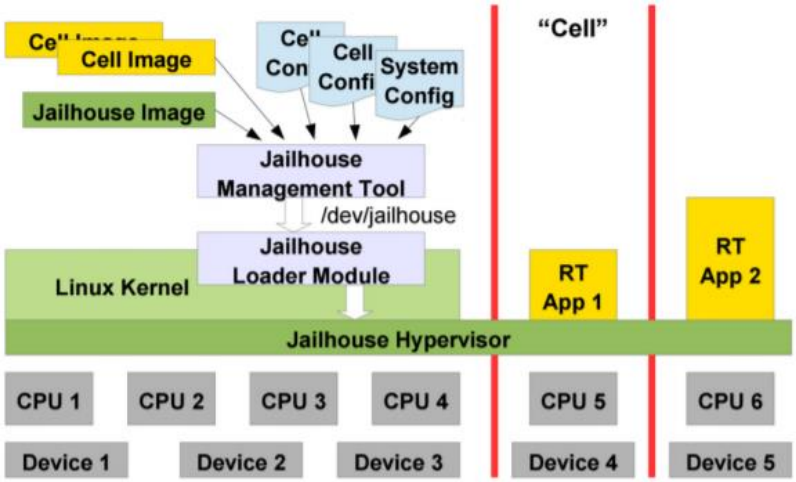


图 3-9 Jailhouse 架构图[19]

Figure 3-9 Jailhouse Architecture

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。

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。

### 3.4.2 Jailhouse 启动流程研究

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。

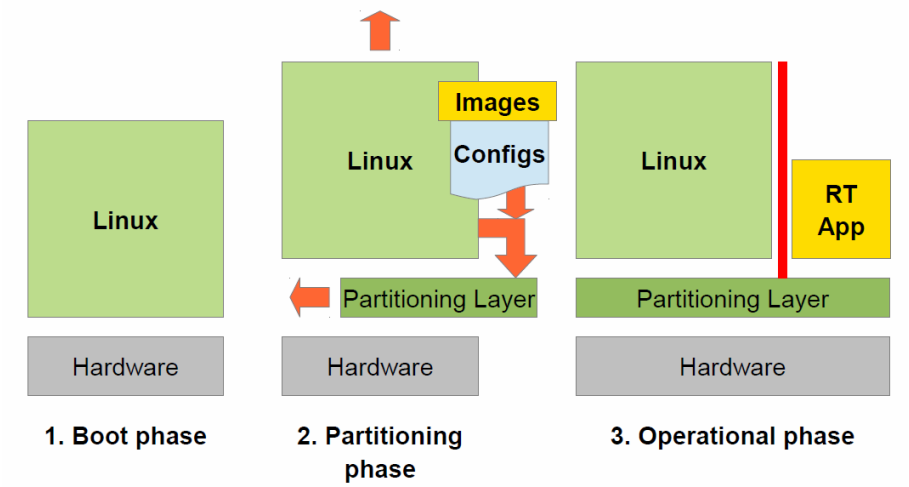


图 3-10 Jailhouse 启动流程

Figure 3-10 Jailhouse Startup Process

## 3.5 本章小结

KVM 和 Xen 都是非常著名的开源虚拟化系统，并且都已经移植到了 ARMv8 架构上。KVM 是 Type2 型虚拟化 hypervisor，作为内核模块运行于宿主机 Linux 上，只实现 CPU 和内存虚拟化，复用 Linux 的功能简化了 KVM 实现。KVM 在 ARMv8 架构下以分离模式运行，虚拟机与 highvisor 之间切换涉及到多次异常级切换。Xen 是 Type1 型的 hypervisor，支持全虚拟化和半虚拟化，在 ARMv8 架构下运行于 EL2 异常级，设备 IO 借助运行于 Dom0 内的 Linux 完成。除了这两者外，还有一些针对 ARMv8 架构的开源虚拟化系统，比如说 Xvisor，一种 Type1 型的完全宏内核 hypervisor，将所有功能都实现在 hypervisor 内，设计和性能都符合车载应用，但其支持的设备驱动有限，而车载硬件有很多各种各样的设备，手动移植这些设备驱动非常麻烦。Jailhouse 是不同于传统虚拟化系统的静态分区虚拟化系统，hypervisor 只是将硬件资源分区，每个区称为 Cell，每个 Cell 内运行一个操作系统或者裸机应用。

# 第四章 方案选型与测试设计

## 4.1 选型准则

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。除此之外，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涉及多次上下文切换，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，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[16]

Table 4-1 Hypercall overhead of KVM and Xen

| 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，本文准备的 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，本文目前也是主要围绕这两个解决方案进行开发。

## 4.3 测试设计与实现

虚拟化技术带来了很多好处和方便，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，既是彼此独立又可交互信息，提高了硬件资源利用率，大幅降低了成本。在一块硬件上同时运行多个操作系统，虚拟化技术可以确保各个系统之间相互隔离、安全，既满足了日益复杂场景下的不同业务需求，又大大提升系统的可靠性和安全性。但是虚拟化技术也带了系统性能开销，如果运行在虚拟化框架上操作系统其性能表现不好，甚至是糟糕，可能无法满足很多应用场景的需求。因此虚拟化方案的性能表现也是评估虚拟化方案非常重要的指标。本文基准测试从图 4-1 黄色部分几个方面来进行虚拟化性能测试。而实时性测试则是主要衡量 GIC 中 PPI 和 SGI 的延迟性能(图 4-2)。

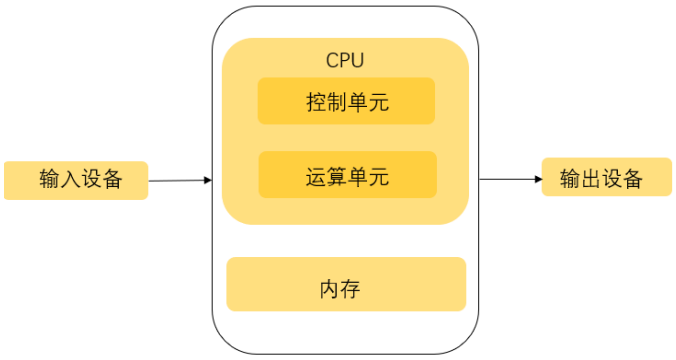


图 4-1 测试框架

**Figure 4-1 Test Architecture**

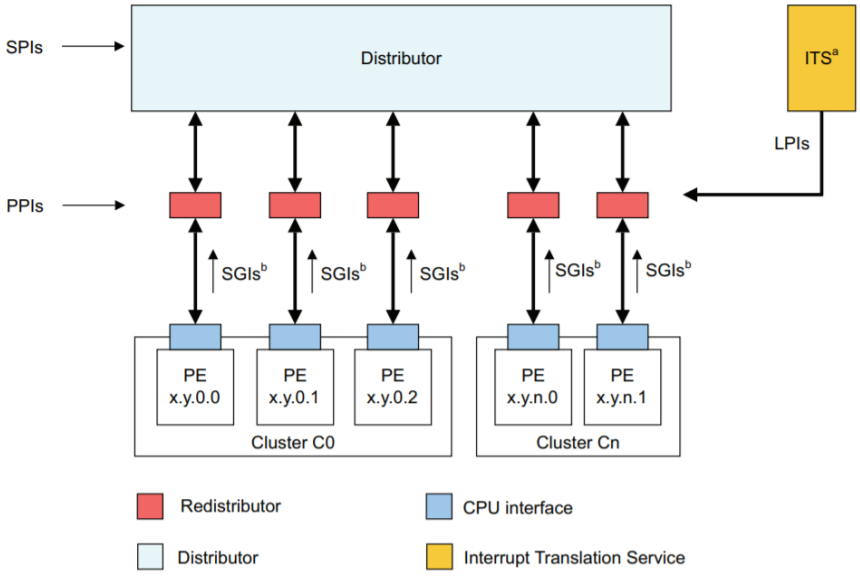


图 4-2 GICv3 架构图[32]

**Figure 4-2 GICv3 Architecture**

### 4.3.1 基准测试设计与实现

虚拟化性能包括了很宽泛的领域，其性能测试应该是一个整体的、全方位、系统性的测试。比如说其性能测试应该涉及各个子系统测试，比如 CPU 性能、内存性能、磁盘性能、系统调用或者一些系统操作等等各方面性能。

(1) CPU 性能测试方法设计，CPU 是计算机组件中最核心的部分，负责每一条指令的载入与执行，其性能直接影响整个操作系统的性能表现。所以 CPU 的性能测试是虚拟化性能测试设计的首要工作之一。本实验中对 CPU 的性能测试主要从整数计算操作的处理效率和浮点数的计算操作性能表现来进行评估。

Dhrystone，Dhrystone 是常见的测量 CPU 运算能力的基准测试程序之一，其对 CPU 的性能测试很具有代表性。Dhrystone 的测试标准也很简单，该测试主要聚焦于整形数操作的处理，关键标准就是在单位时间内运行了多少个 Dhrystone，其测试数据是广泛地从各个软件数据中收集而来，比较具有代表性，该测试输出指标是每秒执行该测试的循环次数(lps loop per second)。本实验总共测试 7 个样本，每个样本测试 10 秒，尽可能排除偶然数据，最后取平均值得到单位时间处理次数。

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，该测试输出指标是每秒执行的百万指令数(MWIPS Millions of Whetstone Instructions Per Second)。与 Dhrystone 类似，在本实验汇总 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。

(2) 文件 IO 测试方法设计，在操作系统中文件是非常重要的组成部分，不仅保存了用户应用的数据和用户程序代码，还保存着操作系统内核的很多信息和关键数据。操作系统运行时，时时刻刻都需要进行文件读写操作，文件读写性能对虚拟化系统的性能影响非常大。尤其现在信息时代产生了了海量数据，读磁盘的读写性能效率要求非常之高。本实验中主要采取 Unixbench File copy 对文件进行测试。Unixbench 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。

除此之外，我自己用 Linux 中的 dd 命令对磁盘也进行了读写性能测试，在 Linux 系统下 dd 命令一般用来进行文件拷贝。在本实验中用 dd 命令来进行磁盘读写性能的测试。如代码 4-1，bs 参数即 block size，每次读写的数据量，count 表示共执行读写操作的次数，if 表示输入文件，of 表示输出文件，一般来说对磁盘的读写会使用内存作为缓冲区，先把数据读到内存上，并没有把数据写到磁盘，所以为了测试磁盘性能必须立即将数据同步到磁盘，加上 oflag=direct, nonblock 参数表示每次读写都会直接写到磁盘，不使用写缓存技术，这样比较接近真实的磁盘性能。/dev/zero 设备可以产生无穷无尽的 0，从该设备读数据不会产生 IO，用来测试磁盘写速度；类似的，/dev/null 设备不产生写磁盘的 IO，可以向该设备无限写内容，可用来测试磁盘读速度。

**代码 4-1 磁盘读写测试**

**Code 4-1 Disk Read-Write Test**

```
1.  dd bs=64k count=4k if=/dev/zero of=test oflag=direct,nonblock
2.  dd bs=64k count=4k if=test of=/dev/null iflag=direct
3.  dd bs=512 count=1000000 if=/dev/zero of=test oflag=direct
4.  dd bs=512 count=1000000 if=test of=/dev/null iflag=direct
```

另外还可以用 dd 对大量小文件进行读写操作来测试磁盘的读写延迟，如代码 4-1 第三、四行分别对磁盘读写延迟做测试，记录执行该指令的时间，用时间除以 count 值得到每次 IO 延迟时间。将每次读写数据设置非常小，比如 512Bytes，进行读写延迟测试。

(3) 内存测试设计，内存也是操作系统性能测试中非常重要的一环，所有的用户程序和系统程序都需要加载到内存中来进行处理，除了程序外，需要处理的数据也需要加载到内存中来。内存作为磁盘和 CPU 之间的缓冲区，其性能表现直接影响到了整个系统的性能。本实验主要从两个方便对内存性能进行测试，内存访问延迟和带宽。该部分测试由我自己设计并编写 C 语言代码进行测试。

**代码 4-2 内存延迟测试算法**

**Code 4-2 Memory Latency Test**

**Algorithm 1** Memory Latency Algorithm

**Input:** *memorySize stride*

```
// put the value of i in register ;
for range = 512; range < memorySize; range = step(range) do
    for i = 1; i < n; i = i + 100 do
        // initialize memory with a circular list pointer;
        // Interval between elements is stride ;
    end for
    for i = 1; i < maxNum; i = i + 100 do
        // unloop access memory 100;
    end for
    // calculate the time of every memory load.
    result = time / maxNum;
end for
```

内存单次访问延迟一般都是纳秒级，单位级非常小，所以需要放大其访问时间，该部分代码设计思路为访问 1 亿次内存，记录其总时间，然后计算单次访问时间。由于存在多级缓存，所以对不同的内存大小进行测试，从 2KB 到 64MB 逐一进行测试，击穿缓存大小，为防止数据 locality 问题，使用较长的内存访问步长，尽量避免该问题。除此之外，将循环中的其它无关变量放在寄存器中，而不放在内存中，寄存器的访问速度远快于内存，可以减小对测试结果的影响。并且将循环尽可能展开，防止循环带来的开销影响测试结果，本实验中按单次展开 100 次循环。

内存带宽测试的设计基于内存延迟设计实现，统计所有访问数据量，记录总延迟，数据量除以时间延迟即可得到内存带宽。

### 4.3.2 其它测试设计与实现

其它系统测试设计，除了 CPU、磁盘和内存外，系统本身的一些操作性能也是衡量虚拟化系统比较关键的指标。比如说系统调用开销、管道性能等等，该部分测试主要使用 Unixbenc3.1.5 进行综合测试。测试结果用 Unixbench 系统跑分计算公式进行跑分计算，得到一个系统性能综合分数。如表 4-2 所示。

**表 4-2 其它系统测试**

**Table 4-2 Other System Tests**

| | |
|---|---|
| Execl Throughput 测试 | 此测试项实际上是每秒对 execl 函数的调用次数 |
| Pipe Throughput(管道吞吐)测试 | 该测试建立一个管道，向管道里写 512 Bytes 数据，然后读出该数据，记录读写 |

| | |
|---|---|
| | 次数。 |
| 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。 |
| 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。 |
| 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。 |
| 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 |

对于车载场景的需求来说，除了操作系统的整体表现以外，其实时性能也是非常重要的，实时性一般指在一定时间内系统的响应速度和能力。本实验中的实时性能指的是从线程创建到 CPU 响应该线程的时间，除了单线程直接响应时间以外，加入多线程干扰，增加 CPU 线程处理压力，测试在多线程干扰下，其响应时间的抖动具有更好的参考意义。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。

Hackbench 是一个针对 Linux 内核调度器的压力测试，它的主要工作是创建多组线程或者进程，每组进程间通过 socket 或者 pipe 进行通信，最后得来一次传输数据和接受数据来回所需要的时间。本实验中设置了 10 组发送者和接收者，每组使用 40 个文件描述符进行通信，每个发送者会发送 100 条 100 bytes 大小的数据，最后统计其总时长。

网络延迟测试，该部分代码由我自己设计编写，主要设计思路是在本地运行两个进程，分别作为服务端(server)和客户端(client)，测试从客户端发送网络包到收到来自服务端的网络包的延迟，服务端和客户端通信使用 TCP 协议。每次发送的网络包大小为 1 Byte，重复发送 100000 次，最后计算得到平均延迟。由于设备限制，在家里无法无法将开发板接上网线，无法进行真正的网络测试，所以所有网络延迟测试都是在本地进行测试。

**代码 4-3 IPI 延迟测试**

**Code 4-3 IPI Latency Test**

**Algorithm 2** Inter Processor Interrupt

```
1: totalTime = 0;
2: for iter = 0; iter < MAXITER; iter++ do
3:     // bind current task with current cpu, prevent preempted by other tasks;
4:     starTime = getTime();
5:     sendipi(handleIpiFunc);
6:     // handleIpiFunc get current time and sub startTime to get diffTime;
7:     // unbind current cpu;
8:     totalTime = totalTime - diffTime;
9: end for
```

核间中断测试(ipi)，核间中断在 ARMv8 架构的 GICv3 中属于 SGI 中断，即软件生成中

断。该类型的中断主要是一个 CPU 向另一个 CPU 发送中断。GICv3 间的中断传输是 Jailhouse 虚拟化框架下为数不多的开销，也是 Xen 的主要开销之一，并且核间中断的延迟对于实时系统来说也是比较重要的性能指标，所以测试 IPI 延迟对于车载系统也是很有必要的。核间中断测试代码由我本人编写，以内核模块的方式实现，通过加载内核模块到 Linux 系统，该内核模块向系统中任一除当前内核模块的 CPU 发送中断，并执行对应的中断函数，测试该过程的时间延迟。重复以上操作 100000 次，累加总时长，再计算得到平均每次 IPI 延迟。代码 4-3 为该算法伪代码。

## 4.4 虚拟化平台实现与工具移植

### 4.4.1 Xen 移植

使用的开发板是 ARMv8 架构的 NXP i.MX 8QuadMax，Xen 使用的是 NXP 官网指定的 Xen 4.11 版本，Linux 版本统一使用 NXP 官网指定的 Linux 4.14.98。具体配置和编译过程如下流程图。Jailhouse 的编译使用的是 github 上 mater 版本。图 4-4 是 Xen 的编译启动过程，图 4-6 是 Jailhouse 的编译启动过程。

本实验采用 NXP 生产的 i.MX 8QuadMax (i.MX 8QM)开发板，NXP 官方文档说明该开发板支持 Xen 和 Jailhouse 虚拟化框架，并且官网有部分关于 Xen 和 Jailhouse 相关的搭建过程[26]文档。

在嵌入式开发的工作中，嵌入式的设备十分的丰富多样，各大厂商退出了各种多样的开发板，Linux 开发人员如果对这些开发板都定制对应的嵌入式 Linux，任务十分繁重且有很多不可预估的风险。所以为了简化嵌入式开发，推出了 yocto 项目，yocto 工具可以为开发人员定制指定开发板的嵌入式 Linux，提供了整套 Linux 编译工具。图 4-1 是 Yocto 编译过程[28]。

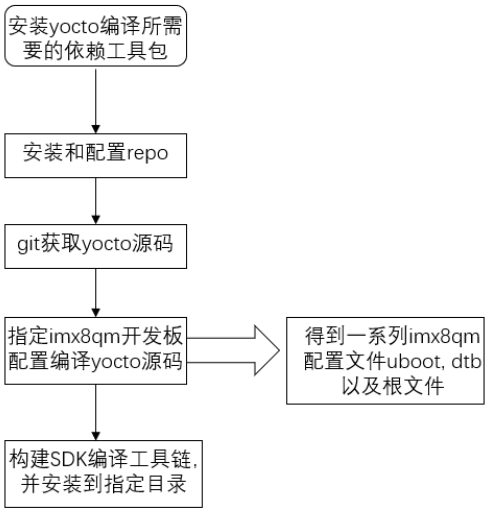


**图 4-3 Yocto 编译过程**

**Figure 4-3 Compile Yocto**

SD 卡准备，SD 卡至少需要 16GB 的空间大小。SD 卡分为三个分区，一个分区为启动所需配置文件分区，第二、三个分区为系统根目录系统。第一个分区 FAT 格式的文件系统，里面包括 xen，Linux Image，imx8qm 对应的设备树文件和 Dom0 的配置文件(fsl-imx8qm-mek-dom0.cfg)，将从官网下载的 spl-imx8qm-xen.bin 拷贝到 FAT 分区，然后将 u-boot-imx8qm-xen-dom0.imx 文件烧写入 SD 卡偏移 32bytes 处；第二个分区为 EXT4 格式的 Dom0 下的 Linux 根文件系统，该区域的根文件系统用 NXP 官网提供的根文件系统，将其烧录进 SD 卡即可。fdisk 建立 SD 卡的第三个分区，用 linux fsck.ext4 命令可以直接将第三分区格式化为

EXT4 文件系统，该区域作为 DomU 的根文件系统(rootfs)。然后用 yocto 生成的 SDK 工具链编译 Linux 源码，并且将编译好的 Linux 镜像拷贝到 SD 卡 FAT 分区。最后编写 DomU 的配置文件，其中主要包括 CPU、内存、系统镜像和设备，系统镜像使用与 Dom0 相同的 Linux 镜像，内存和 CPU 也保持和 Dom0 相同，设备编写用 NXP 官方给的配置文件即可。图 4-2 是 SD 卡启动盘制作过程。

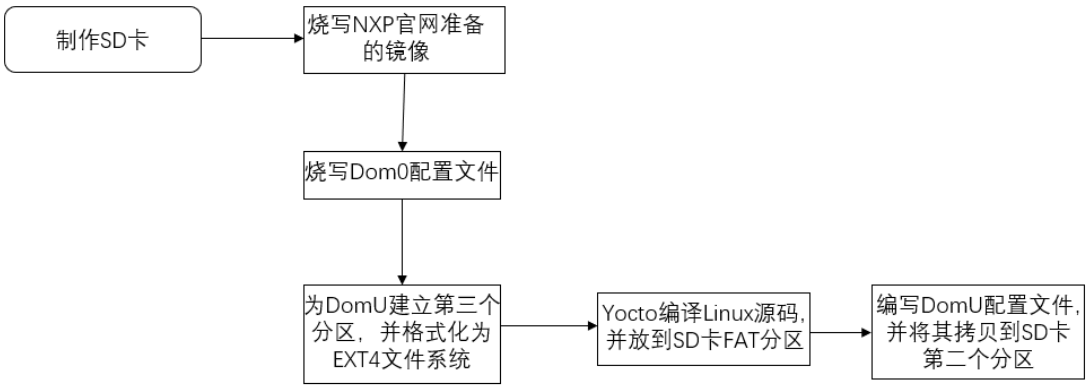


**图 4-4 启动盘制作**

**Figure 4-4 Create Startup Disk**

以上步骤准备就绪后，开始操作硬件启动开发板，首先如图 4-3 连接开发板电源和 USB 调试接口，将 USB 接口连接到本地 Linux 端。然后安装下载 minicom 工具，配置 minicom 接入 ttyUSB0，开发板的输出信息将会通过 USB 接口传输到本地 Linux 系统，方便开发人员进行调试和命令操作。然后用针拨开发板的模式拨片，将开发板设置为从 SD 卡启动。然后插入 SD 卡，上电启动开发板，所有启动信息将从 USB 接口输出到 Linux 终端。

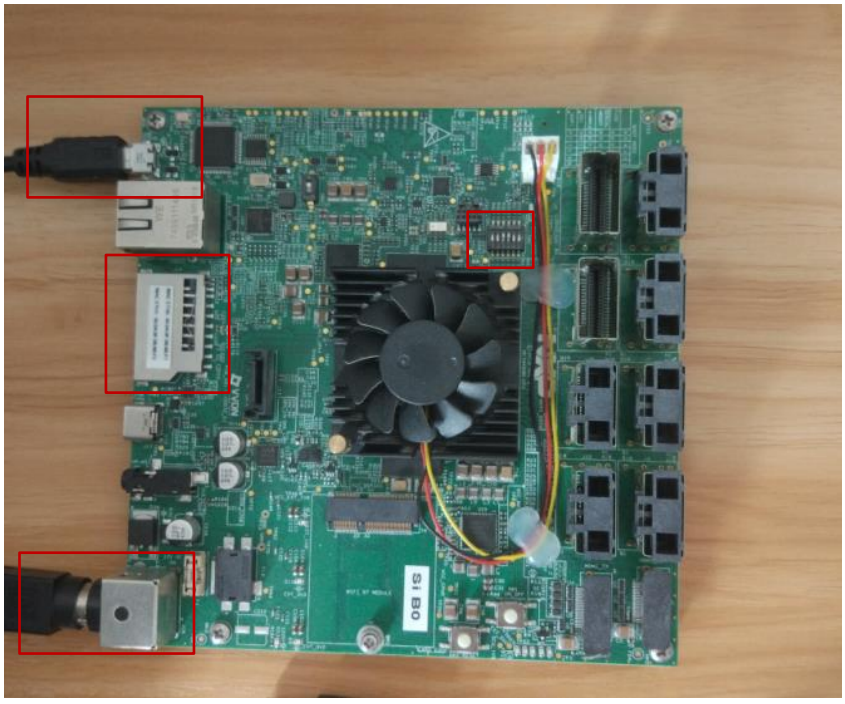

**图 4-5 开发板连接**

**Figure 4-5 Connection of Board**

在系统启动时快速按任意键，将进入 U-boot 阶段，在此阶段输入指定命令操作去运行启动 Xen 和 Dom0，并且进入 Dom0 控制台，Dom0 账户是 root，没有密码可以直接进入。在 Dom0 可以用提前放在 Dom0 根目录下的 DomU 配置文件启动 DomU。到此，Xen、Dom0 和 DomU

全部启动完成。

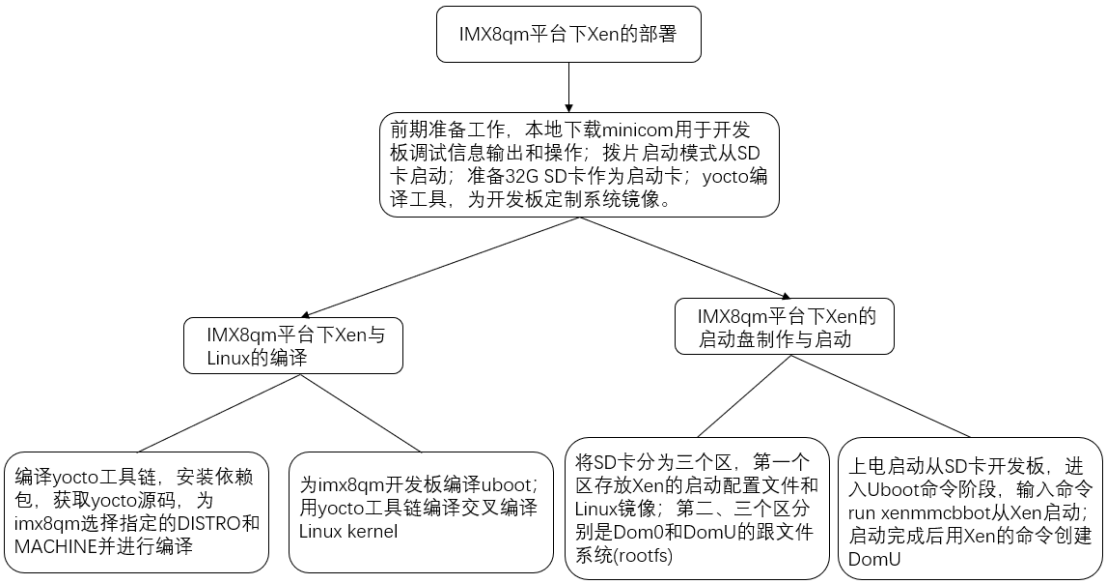


**图 4-6 Xen 部署启动流程**

**Figure 4-6 Deployment Process of Xen**

### 4.4.2 Jailhouse 移植

与 Xen 类似，Jailhouse 的编译和启动也需要用 yocto 工具定制专门的系统。首先需要用 yocto 为 root cell 编译 Linux，Linux 源码需要用 imx 指定版本的源码，然后用 yocto 生成的工具链按默认配置编译 Linux；接下来编译 Jailhouse，Jailhouse 也需要从 NXP 官网去获取，使用指定的版本分支，根据编译好的 Linux 设置好一系列参数后用 yocto 工具链进行编译。编译完成后会生成四类文件，.cell 文件及 Cell 的配置文件，它描述为某个客户机分配的硬件资源，包括 CPU 信息、物理内存、外设等等；然后是 jailhouse.bin，这是 jailhouse 的可执行文件，主要负责管理 Cell 的创建和销毁；还生成了一些可以裸机运行的应用：gic-demo.bin、ivshmem-demo.bin，可用来测试 jailhouse 能否正产运行；最后一类文件是 linux-loader.bin，负责为 Root cell 加载和引导 Linux 系统。下图是 Jailhouse 的编译过程。

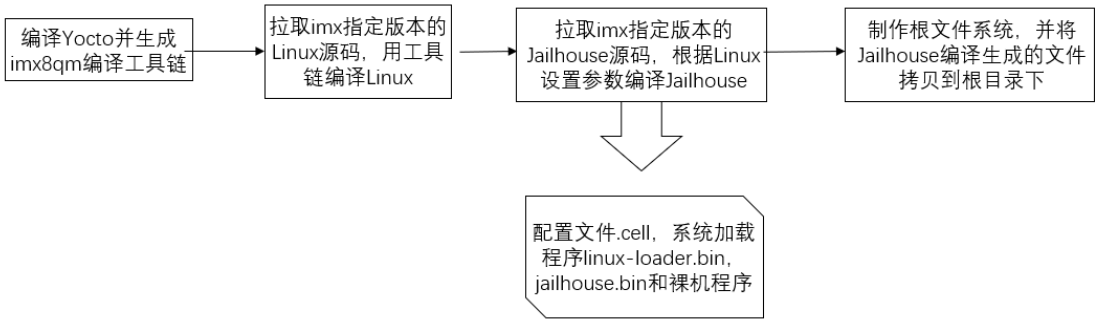


**图 4-7 Jailhouse 编译流程**

**Figure 4-7 Compile Jailhouse**

以上步骤编译完成后，与 Xen 类似，将开发板连接好，将拨片设置为从 SD 卡启动模式。在本地 Linux 系统上启动 minicom 并接收来自 USBtty0 的调试信息。上电启动开发板，快速按空格键进入 U-boot 阶段，输入命令操作将 Jailhouse 代码加载到 Linux 指定内存区域，然后启动系统。Linux 启动完成后，加载 jailhouse.ko 内核模块，然后根据.cell 配置文件启动

jailhouse，使 jailhouse 运行在 hyp 模式，Linux 运行在 Root cell 内，到此，jailhouse 启动完成。

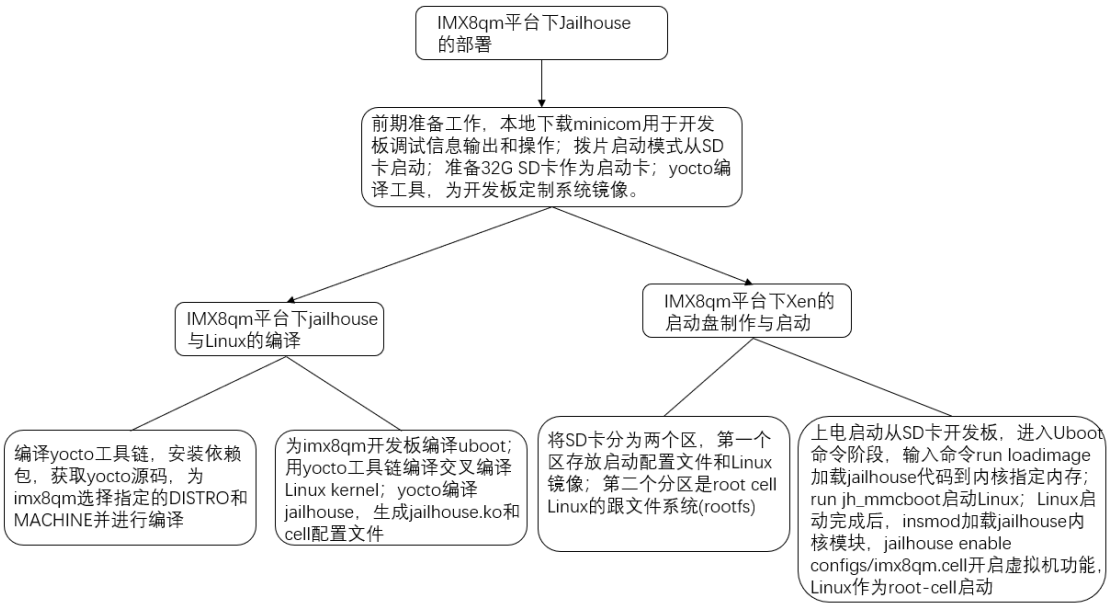


**图 4-8 Jailhouse 部署启动流程**

**Figure 4-8 Deployment Process of Jailhouse**

### 4.4.2 工具移植

对于所有用到的测试工具，包括 unixbench 和 cyclictest，和自己写的代码都需要用 aarch64-linux-gnu 进行交叉编译然后移植到 Linux 对应的根文件系统。对于系统基准测试，Unixbench 其自动化测试脚本是用 perl 语言写的，而自己手动编译的 Linux 内核并没有 perl 模块，所以需要移植 perl 模块到实验环境下。

- 下载 unixbench 代码到本地 linux 系统
- 安装交叉编译器 aarch64-linux-gnu
- 修改 unibench 的 Makefile 文件，CC=aarch64-linux-gnu-gcc
- 修改 Run 脚本文件，将 preChecks()注释掉，因为该函数下的 system("make all") 在程序运行时重新编译 unixbench，而我自己手动编译的 Linux 并不具备编译所需要的环境。
- 用读卡器将 SD 卡连接到本地 Linux USB 接口，挂载 SD 卡第二个分区/dev/sdb2 的 Linux 根目录到本地目录。
- 拷贝编译好的 unixbench 到根文件目录，搜索 ARM64 Linux 根文件系统所有相关的 perl 模块，并将其一一移植到 SD 卡根目录的指定目录下。

除此之外，还有 cyclictest 和我自己的测试代码也需要进行交叉编译才可以在 ARM64 Linux 系统上运行。

- 安装依赖包，libnuma-dev
- aarch64 交叉编译 cyclictest 和我的代码文件
- 用读卡器将 SD 卡连接到本地 Linux USB 接口，挂载 SD 卡第二个分区/dev/sdb2 的 Linux 根目录到本地目录。
- 拷贝可执行文件到 SD 卡跟目录下。

# 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**

**Table 5-1 Experimental environment**

| | |
|---|---|
| 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 |

## 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。

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**

**Table 5-2 Dhrystone Test**

| 测试环境 | Result(lps) | Index |
|---|---|---|
| Linux | 13586135.5 | 1164.2 |
| Jailhouse Root Cell | 13584726.8 | 1164.1 |
| Xen Dom0 | 13582018.6 | 1163.8 |
| Xen DomU | 13583103.8 | 1163.9 |

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

**Table 5-3 Double-Precision Whetstone Test**

| 测试环境 | Result(MWIPS) | Index |
|---|---|---|
| Linux | 2431.4 | 442.1 |

| | | |
|---|---|---|
| Jailhouse Root Cell | 2427.9 | 441.4 |
| Xen Dom0 | 2428.9 | 441.6 |
| Xen DomU | 2428.8 | 441.6 |

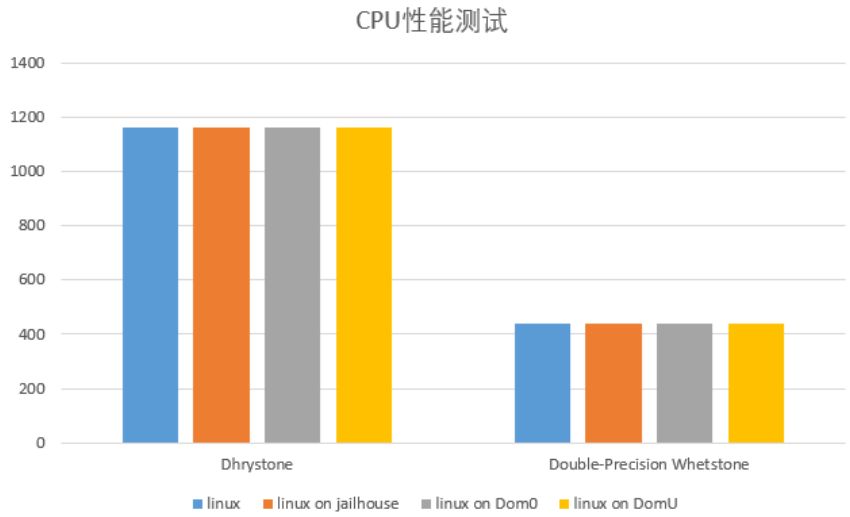


**图 5-1 CPU 性能测试图**

**Figure 5-1 CPU Performance**

从测试结果可以看出，不论是 Dhrystone 还是 Whetstone，四种实验环境下的测试结果几乎没有区别；对于 Jailhouse 来说，其 CPU 实直接分配给 Root cell 内的 Linux 系统的，没有任何虚拟化，所以没有性能开销；而对 Xen 来说，CPU 性能开销也很低，另外，我在编写 Dom0 和 DomU 配置文件时，将 vcpu 绑定到了固定的物理 CPU 上，所以几乎没有 CPU 性能开销。

### 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**

**Table 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 5-5 File copy 256 bufsize 500 maxblocks**

**Table 5-5 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 5-6 File copy 4096 bufsize 8000 maxblocks**

**Table 5-6 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 文件复制性能测试**

**Figure 5-2 File Copy Performance**

Unixbench 的 File copy 测试的文件 IO，Xen 开销比较小，考虑到可能是因为数据量不够答没有体现出差距。所以我自己用 dd 命令进行的磁盘 IO 进行了测试，测试结果如下，分别测试了磁盘读写带宽和磁盘读写延迟。读写带宽分为每次读写数据 32KB 和 64KB，重复进行 4000 次；延迟测试设计为每次读写 512Bytes，重复读写 1 万次。以上所有读写都跳过缓存进行以确保获取真实的磁盘性能测试数据。

**表 5-7 磁盘写带宽**

**Figure 5-7 Disk Write Bandwidth**

| 测试环境 | 32K(M/s) | 64K(M/s) |
|---|---|---|
| Linux | 13.42 | 16.72 |
| Jailhouse Root Cell | 12.94 | 16.6 |
| Xen Dom0 | 12.175 | 15.54 |
| Xen DomU | 11.22 | 14.12 |

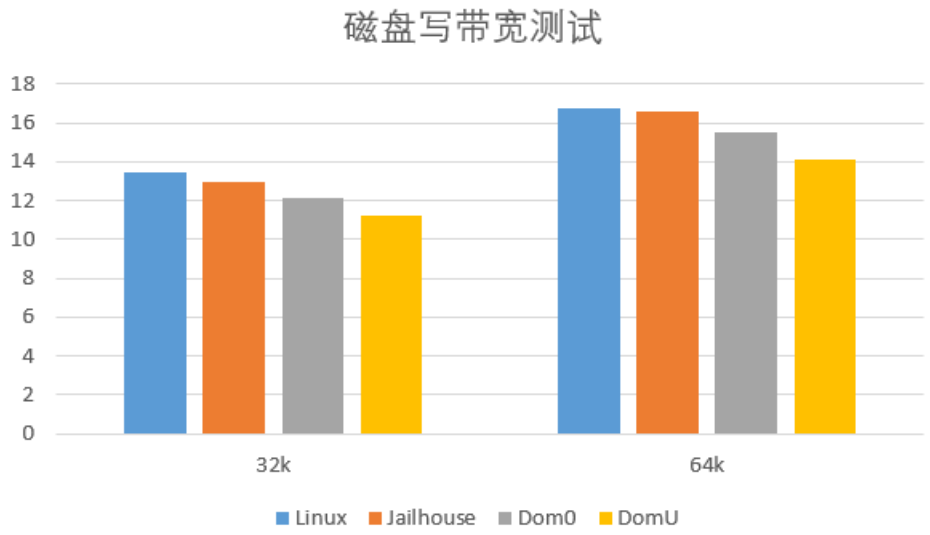


**图 5-3 磁盘写带宽测试**

**Figure 5-3 Disk Write Bandwidth**

**表 5-8 磁盘读带宽**

**Table 5-8 Disk Read Bandwidth**

| 测试环境 | 32K(M/s) | 64K(M/s) |
|---|---|---|
| Linux | 45.3 | 58.83 |
| Jailhouse Root Cell | 45.2 | 58.82 |
| Xen Dom0 | 45.6 | 58.84 |
| Xen DomU | 40.78 | 54.06 |

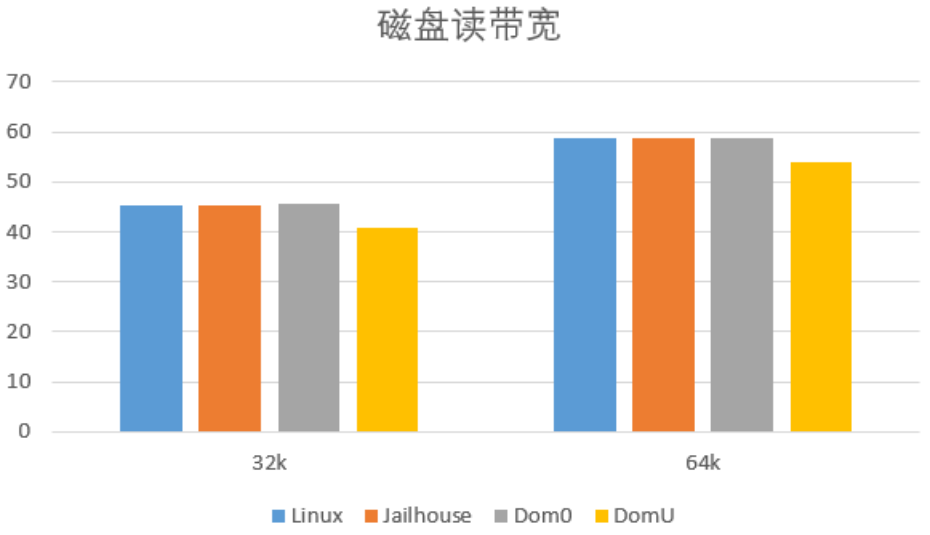


**图 5-4 磁盘读带宽测试**

**Figure 5-4 Disk Read Bandwidth**

**表 5-9 磁盘写延迟**

**Table 5-9 Disk Read-Write Latency**

| 测试环境 | 写延迟(ms) | 读延迟(us) |
|---|---|---|
| Linux | 3.5 | 276.3 |

| | | |
|---|---|---|
| Jailhouse Root Cell | 3.56 | 280.6 |
| Xen Dom0 | 3.71 | 282.02 |
| Xen DomU | 3.95 | 319.8 |

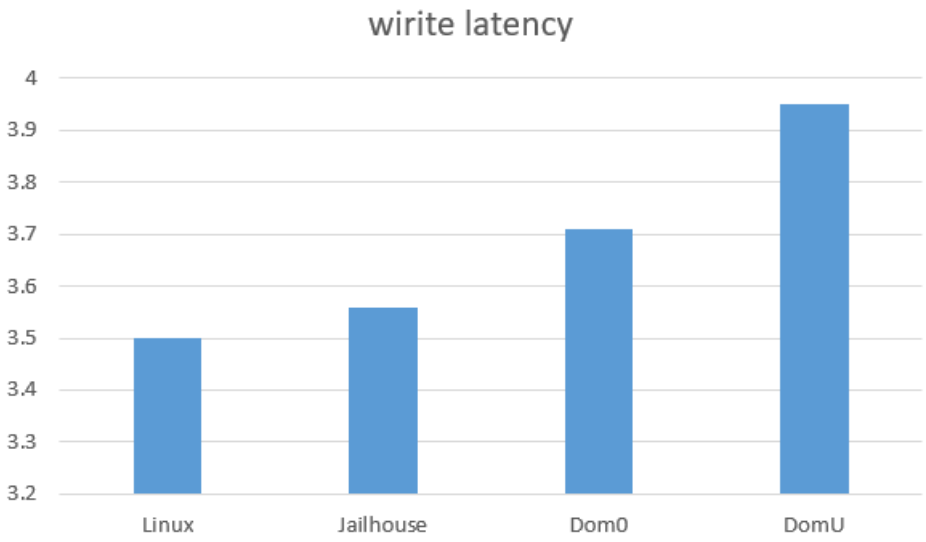


**图 5-5 磁盘写延迟**

**Figure Disk Write Latency**

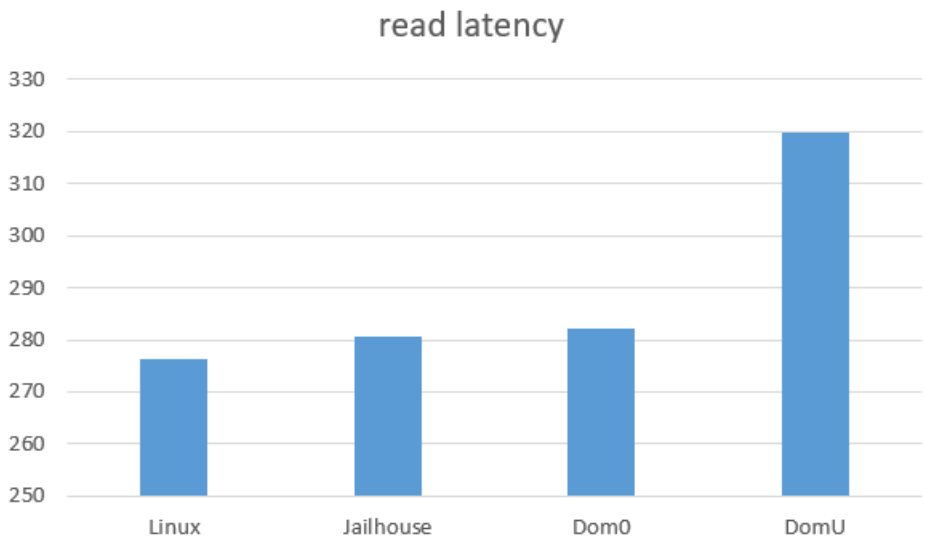


**图 5-5 磁盘读延迟**

**Figure 5-5 Disk Read Latency**

从测试结果来看，Jailhouse 文件 IO 性能几乎没有开销；而 Xen 的 Dom0 文件 IO 有一定的性能开销；当 Unixbench File copy 测试数据量比较小时，文件 IO 性能开销很小，当数据量增大时才开始体现出性能损耗，通过直接测试文件 IO 读写延迟能看出 DomU 的延迟比另外三种实验环境都高，这是因为 DomU 的 IO 操作需要借助 Dom0 来完成，所以对于 DomU 来说，文件读写延迟比较高，而 Dom0 的读写延迟开销几乎没有，因为 Dom0 自己有物理设备驱动，可直接进行文件读写。

### 5.2.3 内存测试

该部分代码由我自己设计和编写进行测试，测试结果显示四种环境下的内存不论是单次访问延迟还是内存访问带宽都差别不大。我又用了比较著名的 benchmark lmbench 进行内存延迟和带宽重新进行测试，发现结果与我自己设计的代码相似。

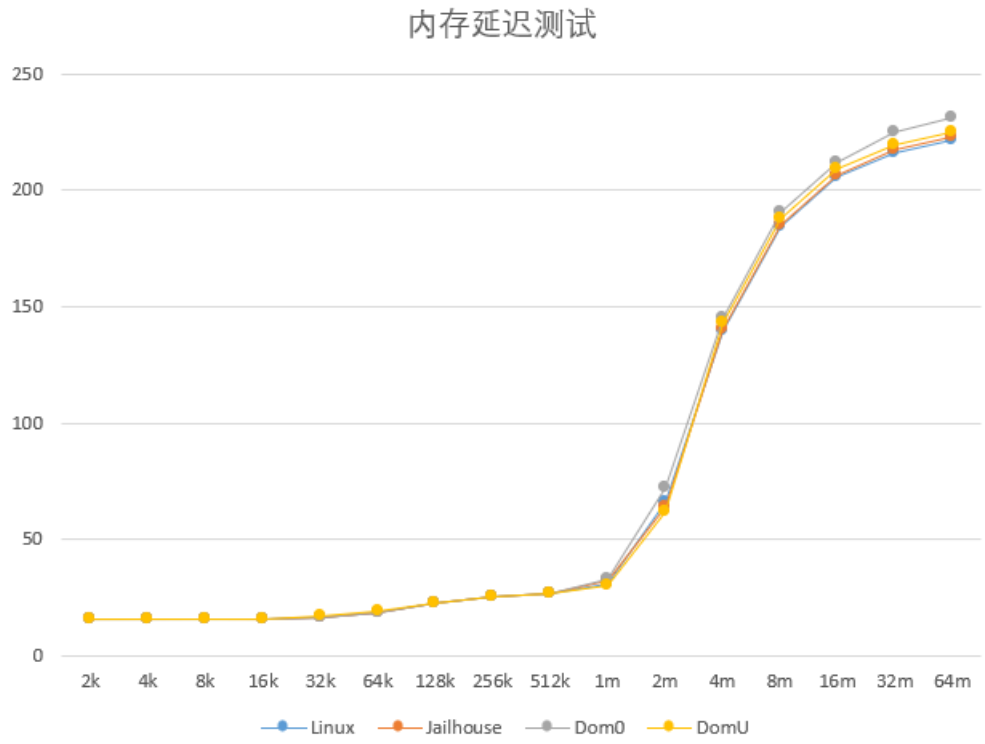


图 5-6 随机访问延迟

Figure 5-6 Random Memory Access Latency

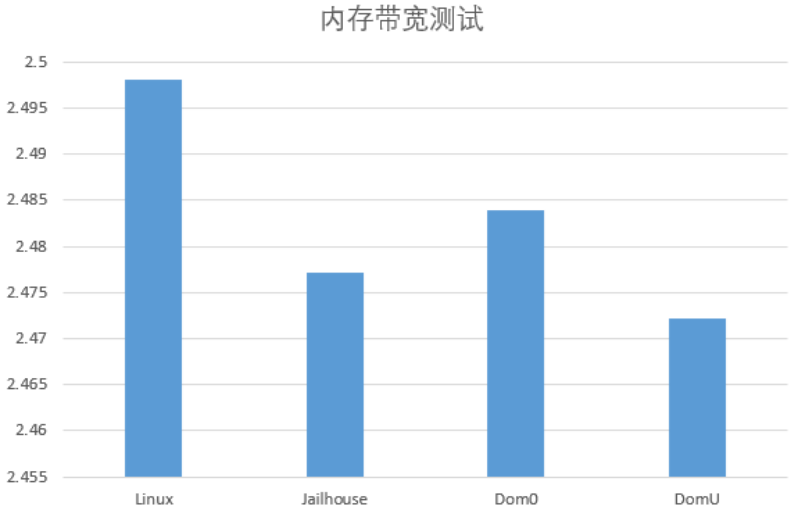


图 5-7 内存带宽测试

Figure 5-7 Memory Bandwidth

Jailhouse 和 Xen 的内存性能都很接近裸机运行的 Linux，Xen 和 Jialhouse 的内存性能开销大概在都在 1%左右。值得注意的是当测试内存大小比较小时，四种环境延迟几乎一样，当测试内存大小开始超过 32KB 时，内存访问延迟开始上升，尤其是当测试内存超过 1MB 的时候，内存延迟开始大幅度上升，并且内存访问延迟开始出现差别。这是由于缓存影响造成的，IMX8qm 开发板有三级缓存，L1 和 L2 缓存大小都为 32K，L3 级缓存为 1MB，所以

当测试内存小于 32K 时内存延迟几乎没有变化，因为内存访问几乎都是从 L1 级缓存读取，速度非常快；当测试内存突破 1MB 时，大量数据开始从内存读取，所以延迟急剧上升。

### 5.2.4 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 5-10 Execl Throughput 测试数据**

**Table 5-10 Execl Throughput Test**

| 测试环境 | Result(lps) | Index |
|---|---|---|
| Linux | 2383.6 | 554.3 |
| Jailhouse Root Cell | 2459.2 | 571.9 |
| Xen Dom0 | 2652.5 | 616.9 |
| Xen DomU | 2604.7 | 605.8 |

**表 5-11 Pipe Throughput 测试数据**

**Table 5-11 Pipe Throughput Test**

| 测试环境 | Result(lps) | Index |
|---|---|---|
| Linux | 640537.9 | 514.9 |
| Jailhouse Root Cell | 643075.3 | 516.9 |
| Xen Dom0 | 642503.6 | 516.5 |
| Xen DomU | 640081.4 | 514.5 |

**表 5-12 Pipe-based Context Switching 测试数据**

**Table 5-12 Pipe-based Context Switching Test**

| 测试环境 | Result(lps) | Index |
|---|---|---|
| Linux | 109379.6 | 273.4 |
| Jailhouse Root Cell | 103648.5 | 259.1 |

| | | |
|---|---|---|
| Xen Dom0 | 105723.2 | 264.3 |
| Xen DomU | 104354.7 | 260.9 |

**表 5-13 Process Creation 测试数据**

**Table 5-13 Process Creation Test**

| 测试环境 | Result(lps) | Index |
|---|---|---|
| Linux | 4060.8 | 322.3 |
| Jailhouse Root Cell | 3774.3 | 299.6 |
| Xen Dom0 | 4844.7 | 384.5 |
| Xen DomU | 4732.9 | 375.6 |

**表 5-14 Shell Scripts 测试数据**

**Table 5-14 Shell Scripts Test**

| 测试环境 | Result(lpm) | Index |
|---|---|---|
| Linux | 3923.9 | 925.4 |
| Jailhouse Root Cell | 3776.0 | 890.6 |
| Xen Dom0 | 3166.9 | 746.9 |
| Xen DomU | 3056.9 | 721.0 |

**表 5-15 System Call Overhead 测试数据**

**Table 5-15 System Call Overhead Test**

| 测试环境 | Result(lps) | Index |
|---|---|---|
| Linux | 653300.1 | 435.5 |
| Jailhouse Root Cell | 656856.2 | 437.9 |
| Xen Dom0 | 658855.9 | 439.2 |
| Xen DomU | 658673.7 | 439.1 |

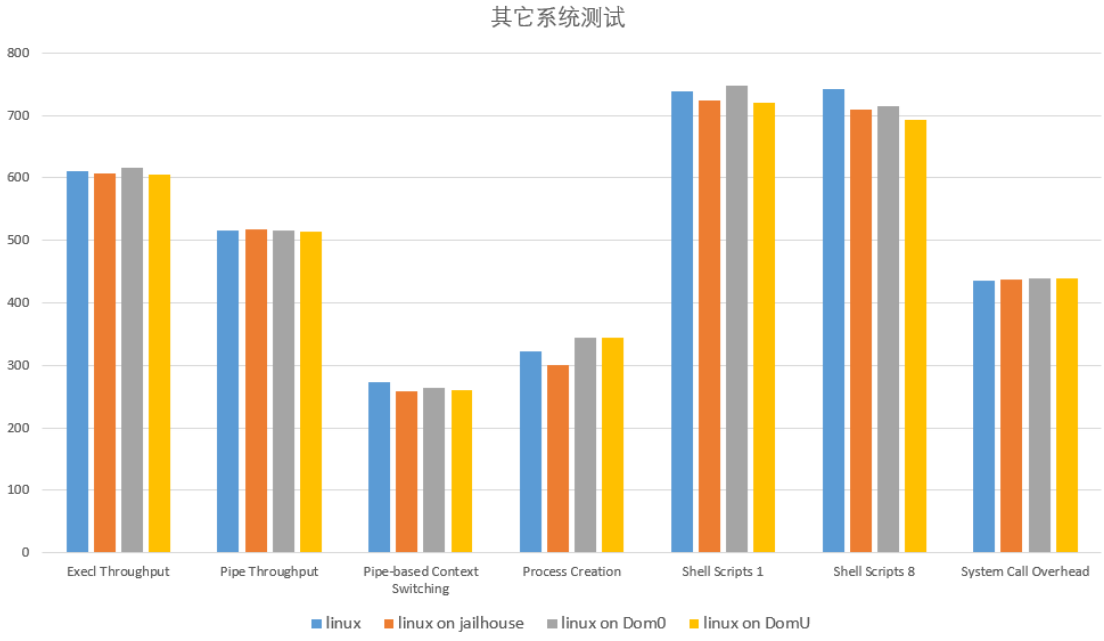


**图 5-8 各项测试数据对比**

**Figure 5-8 Other System Test**

以上几项测试大多和系统相关，大部分操作都可以在虚拟机内由虚拟机的内核完成，不需要借助 hypervisor 完成，所以这几项测试性能开销都不大，有部分测试甚至 Xen 表现出了比裸机运行的 Linux 更好的性能。所以对于可以直接由虚拟机系统完成的工作，其性能变现都非常好。

### 5.2.5 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

$$product = \sum_1^n(\log(Count0) - \log\frac{Time}{TimeBase}) \tag{5-1}$$

进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。

$$score = \exp(\frac{product}{2}) \tag{5-2}$$

**表 5-16 系统跑分数据**

**Table 5-16 System Score**

| 测试环境 | 分数 |
|---|---|
| Linux | 612.9 |
| Jailhouse Root Cell | 603.4 |
| Xen Dom0 | 580.7 |
| Xen DomU | 575.3 |

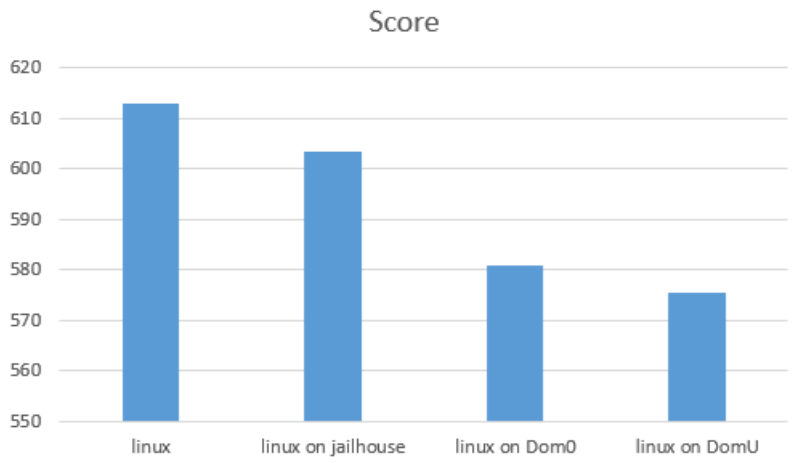


**图 5-9 系统跑分统计**

**Figure 5-9 System Score**

## 5.3 实时性测试

### 5.3.1 单线程延迟测试

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。

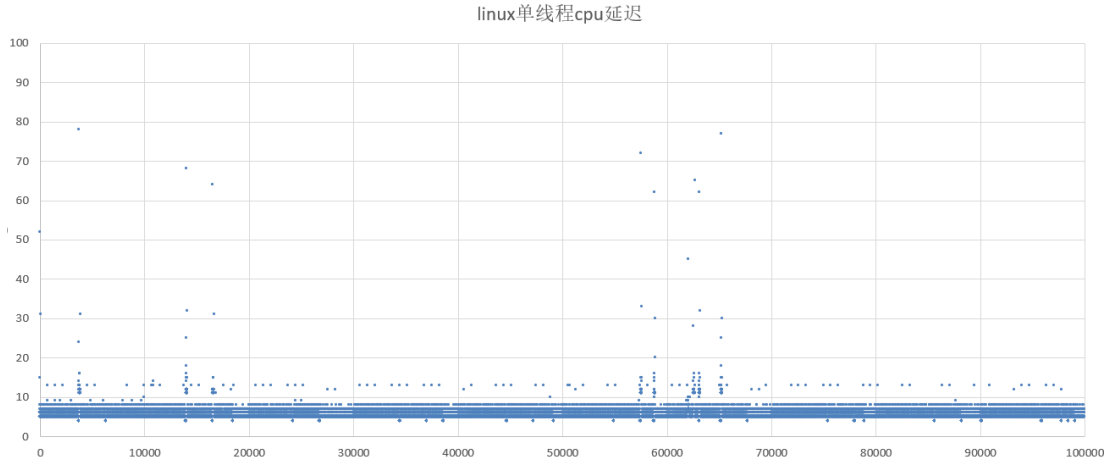


**图 5-10 Linux 单线程延迟**

**Figure 5-10 Linux Sigle Thread Latency**

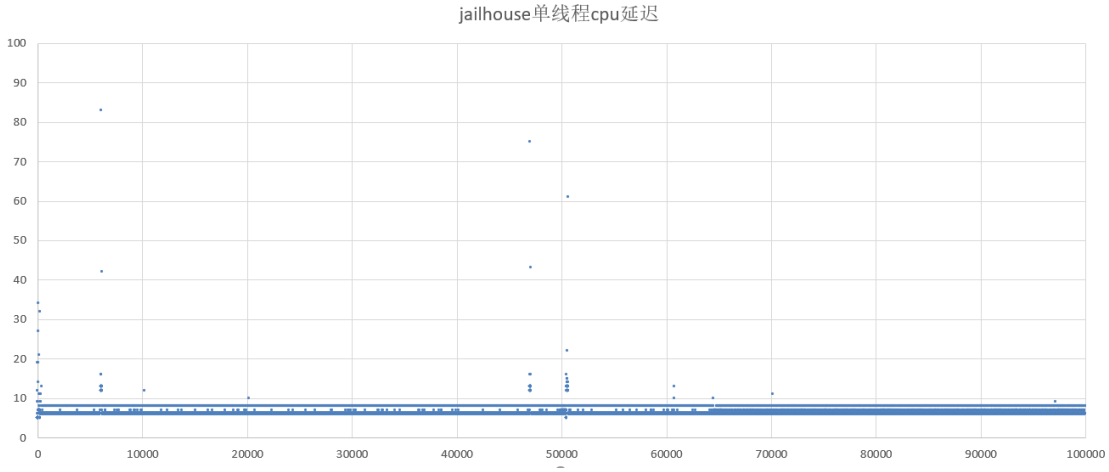


**图 5-11 Jailhouse 单线程延迟**

**Figure 5-11 Jailhouse Single Thread Latency**

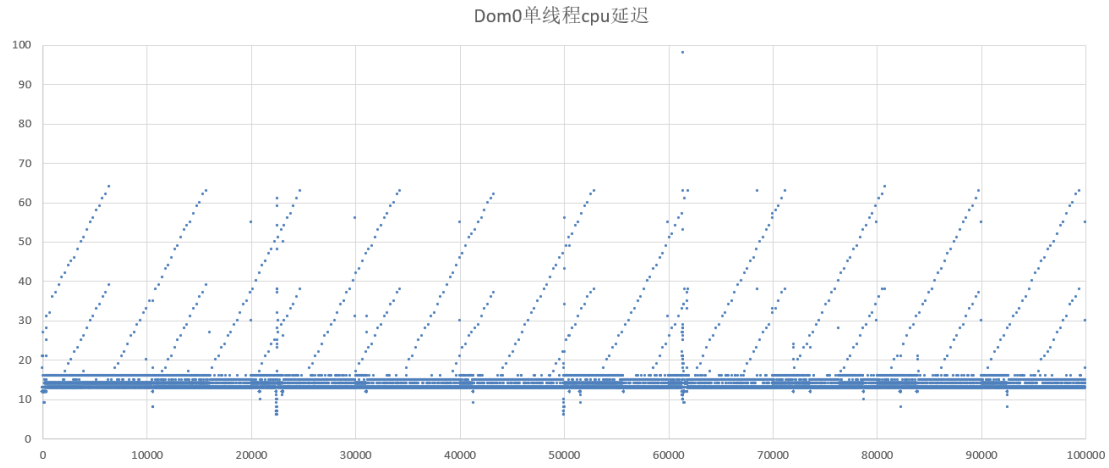


**图 5-12 Dom0 单线程延迟**

**Figure 5-12 Dom0 Single Thread Latency**

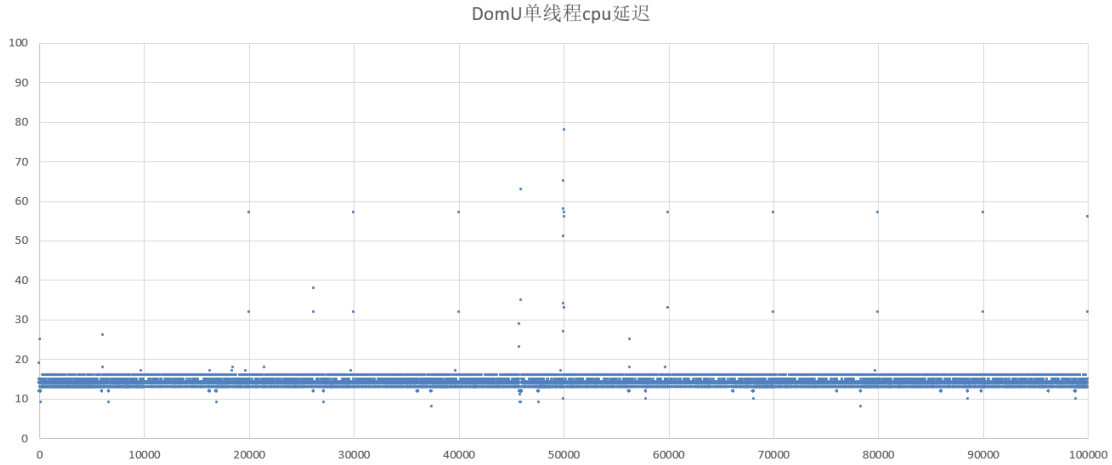


**图 5-13 DomU 单线程延迟**

**Figure 5-13 DomU Single Thread Latency**

### 5.3.2 多线程延迟测试

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。

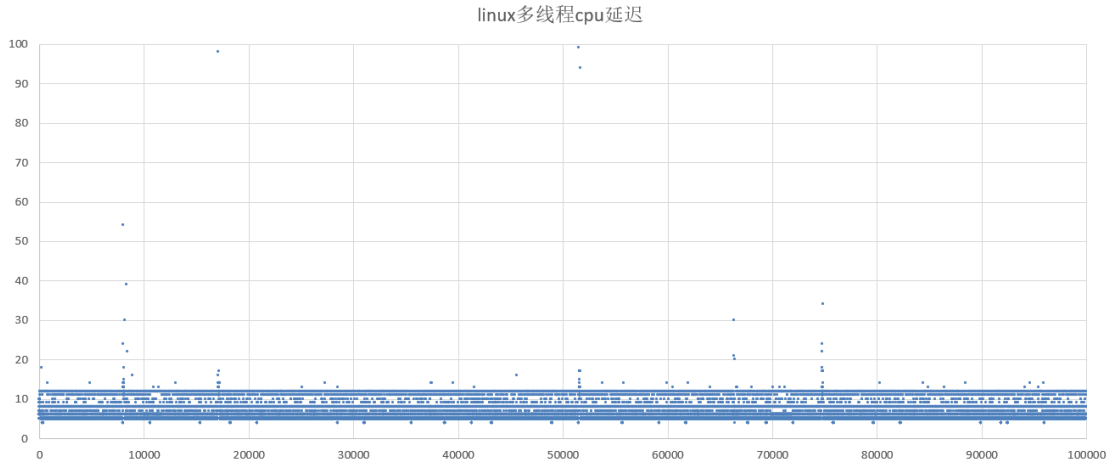


**图 5-14 Linux 多线程干扰测试**

**Figure 5-14 Linux Muti-thread Latency**

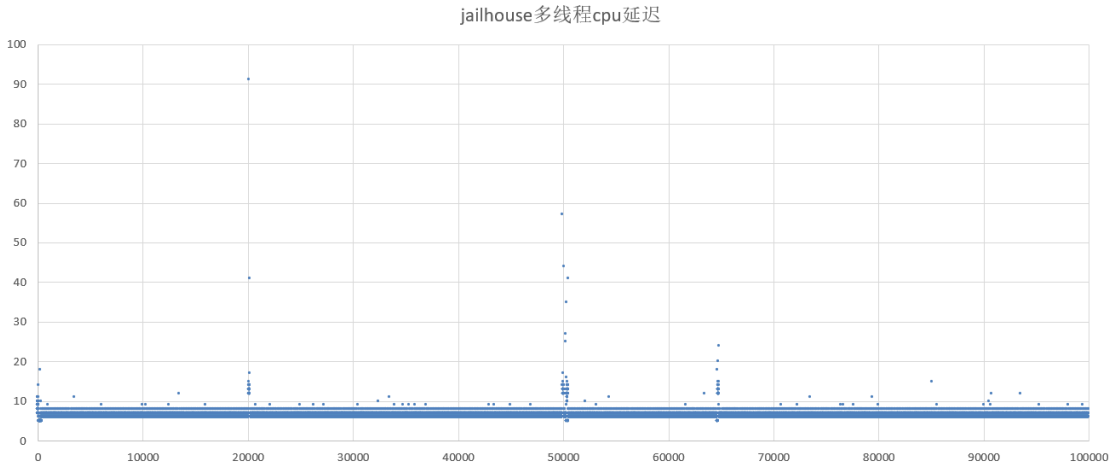

**图 5-15 Jailhouse 多线程干扰测试**

**Figure 5-15 Jailhouse Muti-thread Latency**

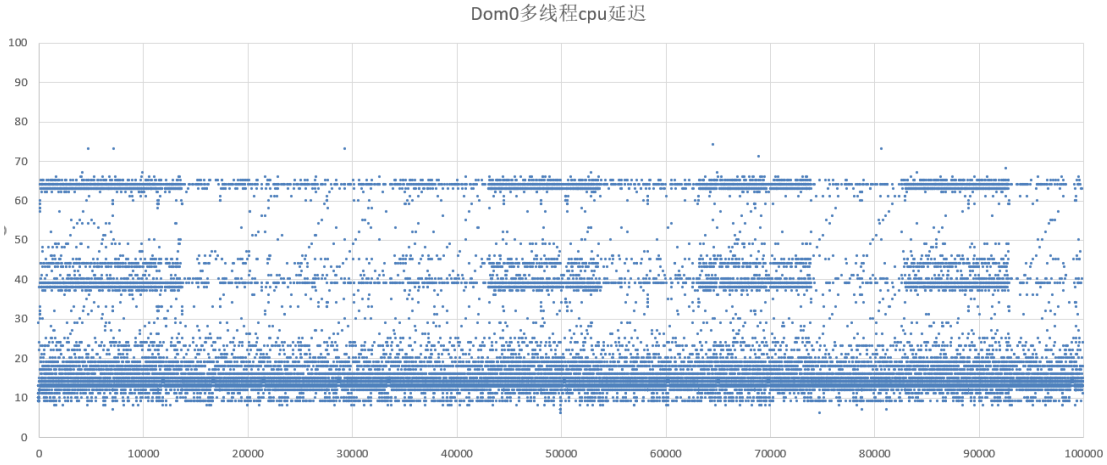


**图 5-16 Dom0 多线程干扰测试**

**Figure 5-16 Dom0 Muti-thread Latency**

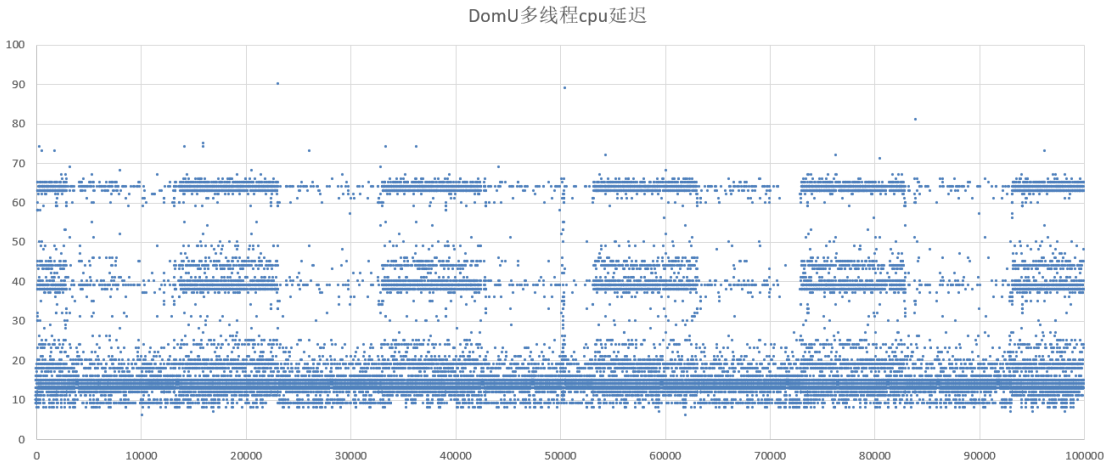


**图 5-17 DomU 多线程干扰测试**

**Figure 5-17 DomU Muti-thread Latency**

对以上八组数据进行均值处理，得到四种环境下的平均响应延迟，以及多线程干扰下，CPU 响应线程的平均延迟，得到图 5-18 统计图对比。

**表 5-17 平均延迟数据**

**Table 5-17 Average Latency**

| 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 |

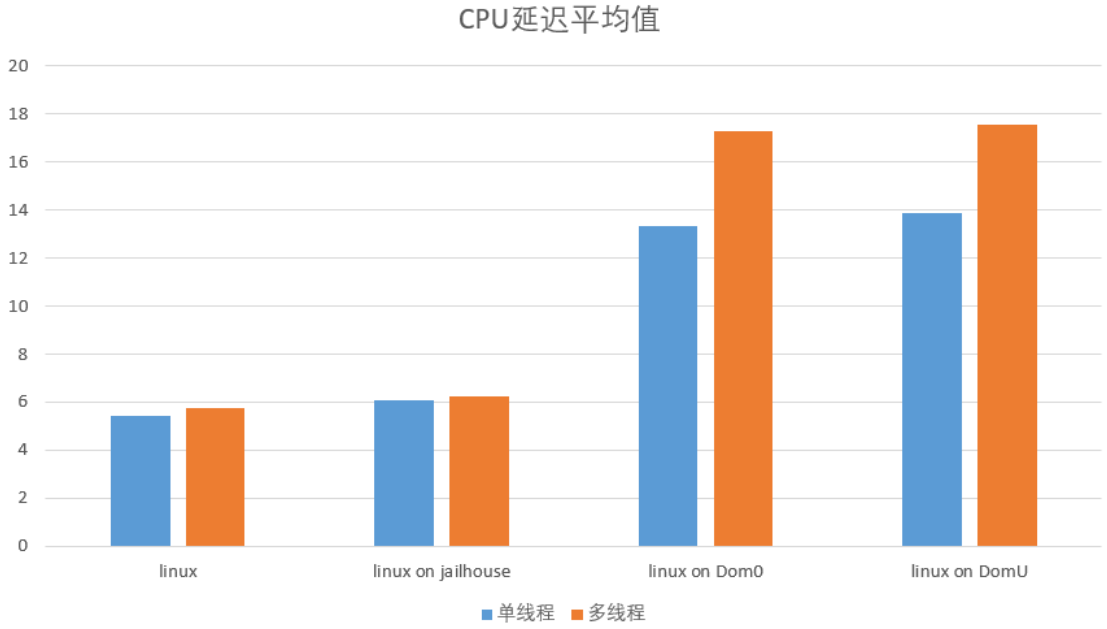


**图 5-18 平均延迟统计图**

**Figure 5-18 Average Latency**

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。

本文认为内核延迟开销主要来源于时钟中断。线程的响应延迟应该主要来自于三个方面：L=L1+L2+I，I 表示更高优先级的任务抢占式干扰导致延迟；L1 指的是线程调度延迟；L2 指的是中断生成延迟，如硬件时钟中断延迟。首先应该不是因为 vcpu 的调度问题导致的线程响应延迟，因为在进行单线程测试时，Xen 的延迟也比 Linux 和 Jailhouse 高很多，单线程不需要频繁进行线程调度，并且我在写 Xen 的配置文件时将 Dom0 和 DomU 的 vCPU 都绑定在了固定的物理 CPU 上，所以其调度开销应该不会比裸机有较大的差距，但是仍然有较大的延迟开销。延迟应该是主要来自于时钟，ARMv8 为每一个 CPU 提供了一个虚拟时钟和三个物理时钟，用来定时用，除此之外还有一个全局计数器。Xen 虚拟机使用虚拟时钟，全局计数器将计数值分发到各个 CPU 的时钟，当定时时间到时后触发中断，时钟中断属于 PPI 类型的中断，对于 Xen 来说，当虚拟机运行时产生时钟中断，退出虚拟机到 EL2 的 Xen，Xen 处理该中断后把中断注入到虚拟机，在进入到虚拟机，虚拟机收到中断。通过阅读 Jailhouse 源码可知，EL2 级的 Jailhouse hypervisor 只负责处理 0-15 号的 SGI(Software Generated Interrupt)中断和 25 号中断 maintenance interrupt，maintenance interrupt 专门设计用来通知 hypervisor 必须处理的事件。而时钟中断属于 PPI 类型的中断，该类中断在 Jailhouse 架构下，可直接由 Guest 虚拟机处理。

### 5.3.3 网络延迟测试

网络延迟测试，主要设计思路是在本地运行两个进程，分别作为服务端(server)和客户端(client)，测试从客户端发送网络包到收到来自服务端的网络包的延迟，服务端和客户端通信使用 TCP 协议。每次发送的网络包大小为 1 Byte，重复发送 100000 次，最后计算得到平均

延迟。下图为网络延迟测试。

**表 5-18 网络延迟测试**

**Talbe 5-18 Network Latency**

| 测试环境 | 网络延迟(us) |
|---|---|
| Linux | 63.7859 |
| Jailhouse Root Cell | 64.8369 |
| Xen Dom0 | 63.1034 |
| Xen DomU | 63.1133 |

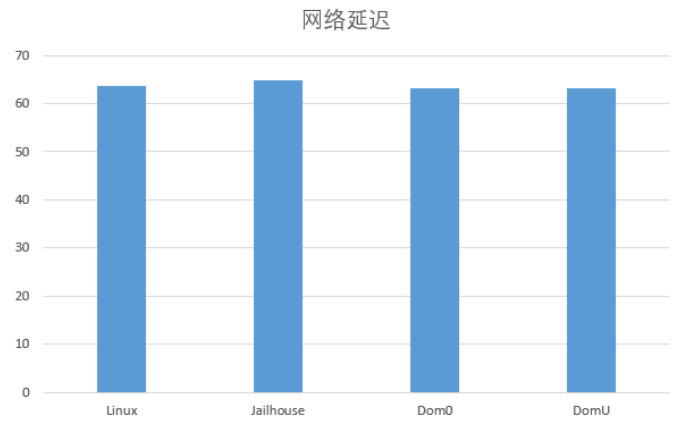


**图 5-19 网络延迟测试**

**Figure 5-19 Network Latency Test**

网络延迟测试结果与本文的预期结果不太符合，理论来说，Xen Dom0 的网络包传输延迟应该很明显地比另外三种环境高，因为网络属于 IO 操作，而 DomU 的 IO 操作需要借助 Dom0 完成，所以应该需要更长的事件来完成网络 IO 操作。但是对于 TCP 发送网络包延迟测试，四种环境几乎一样，Xen 几乎没有开销，理论来说 Xen 的 DomU 延迟性能会差很多，因为 DomU 网络 IO 需要通过 Dom0 内的驱动去进行处理，这是一个比较长的处理路径，但测试显示四种环境差不多一样的延迟。初步分析是因为本次测试是在系统本地进行测试的，由于设备限制暂时没有条件为 imx8qm 开发板插上网络并配置网络，所以网络测试使用的是 127.0.0.1(localhost)主机地址，对于本机地址，网络传输不需要通过真实的网卡进行传输，即网络回环(loop-back)，系统发现是回环地址直接进行，报文从应用层通过 socket 接口到传输层，然后到网络层，发现不用走中断不用继续向下发送给链接层，直接返回传输层，然后传回应用程序。

### 5.3.4 核间中断延迟测试

主要设计思路是利用 Linux 内核模块，将内核模块加载到内核中，当模块向另一个 CPU 发送 IPI 时，将模块与当前 CPU 绑定，记录当前时间，然后向除了当前 CPU 之外的任一 CPU 发送 IPI，接收者 CPU 收到 IPI 后记录时间返回给发送者 CPU 收到 IPI 的时间，两者时间相减得到 IPI 发送延迟时间；重复发送 100000 次 IPI，得到总时间，计算平均时间。

**表 5-19 IPI 延迟数据**

**Table 5-19 IPI Latency Data**

| 测试环境 | IPI 延迟(ns) |
|---|---|
| Linux | 1854.40525 |
| Jailhouse Root Cell | 2321.57125 |

| | |
|---|---|
| Xen Dom0 | 4253.2655 |
| Xen DomU | 4190.34575 |

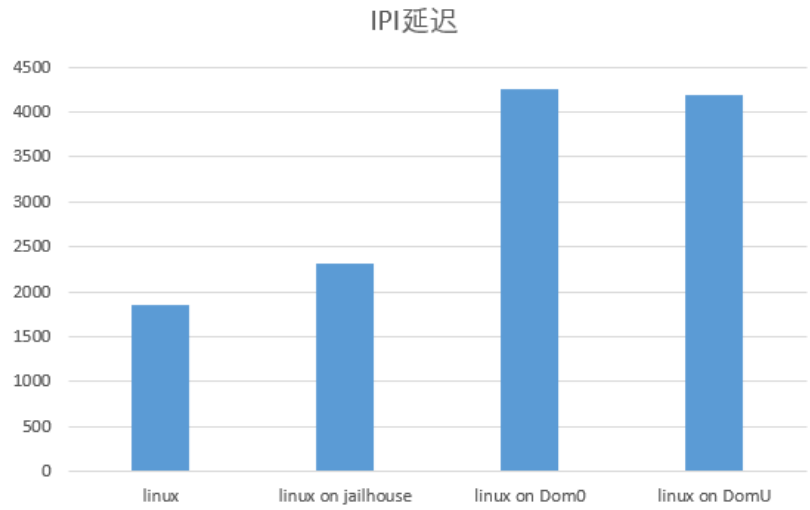


**图 5-20 IPI 延迟测试**

**Figure 5-20 IPI Latency Test**

实验结果显示，Jailhouse 的 IPI 延迟相比裸机 Linux 有较高的延迟，而 Xen 的 Dom0 和 DomU 延迟损耗更高，两倍与裸机 Linux。在 GICv3 的架构下，通常来说中断被分为四种类型，分别是 PPI，SGI，SPI 和 LPI，其中 LPI 是基于消息的中断，GICv3 新增加的一种中断，暂不做讨论。除 LPI 外的三种中断，通过阅读 Jailhouse 源码可知，Jailhouse 会接管 SGI 和 SPI，以及 PPI 中的第 25 号 maintenance interrupt。IPI 就属于 SGI，即软中断。所以当一个 CPU 向另一个 CPU 发送，需要从当前 EL1 级的 Cell 退出到 EL2 级的 Jailhouse Hypervisor 进行处理，Jailhouse 通过写 ICC_SGI1R_EL1 寄存器，直接向接收者 CPU 发送 SGI 中断，所以在 Jailhouse 虚拟化框架下，发送 IPI 中断相较于裸机的开销就是从 EL1 到 EL2 的上下文切换的开销，由于 EL1 和 EL2 有自己独立的一套寄存器，所以上下文切换只涉及到 30 个通用寄存器(GP Register)的状态保存和恢复，所以延迟开销较小。而对于 Xen 来说，Xen 为每一个虚拟机都虚拟的 GIC 中断控制器(vGIC),Xen 通过操作 List Register 来注入虚拟中断，List Register 是 GICv3 专门中断虚拟化设计的寄存器，发送给虚拟机的中断写在 List Register 寄存器内，重新进入虚拟机后从这些寄存器内读出中断，并通过 List Register 返回状态。当中断超过 List Register 的容量时，Hypervisor 将虚拟中断放在内存上，所以 List Register 相当于一个缓冲区，加快中断的处理。但是除此之外还要额外维护一个虚拟的 GIC 状态，所以比 Jailhouse 有更大的开销。

## 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 1.55%，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，但也不是很大，都在 5%左右。尤其在 CPU 运算性能方面，其虚拟化性能损耗几乎没有，可以由操作系统完成的性能测试，其损耗几乎没有，甚至有的测试 xen 比 native 性能更好。内存测试稍有损耗，由于两段地址翻译的硬件支持，内存性能损耗非常低。值得注意的是，当测试内存大小不超过 32KB 时，内存访问延迟几乎没有变化，当超过 32K 时内存访问延迟开始增加，尤其当测试内存大小超过 1MB 时，内存延迟急剧上升。这是由于缓存影响造成的，IMX8qm 开发板有三级缓存，L1 和 L2 缓存

大小都为 32K，L3 级缓存为 1MB，所以当测试内存小于 32K 时内存延迟几乎没有变化，因为内存访问几乎都是从 L1 级缓存读取，速度非常快；当测试内存突破 1MB 时，大量数据开始从内存读取，所以延迟急剧上升。最大的开销在于文件的读写性能，尤其对于 DomU，因为文件 IO 需要借助 Dom0 完成，处理路径很长。

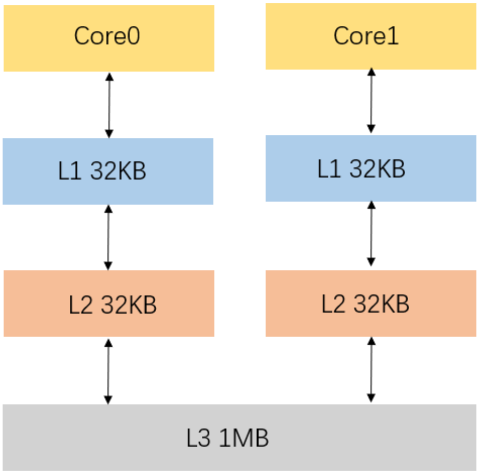


**图 5-20 I.MX8qm 三级缓存**

Figure 5-20 I.MX8qm Cache

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，其综合性能损耗几乎没有，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 有较小的性能损耗，但其实时延迟测试表现并不算好，延迟抖动也比较大。

# 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 性能损耗非常小，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，性能总体损耗也比较小，除了 IO 操作和内存，其它性能测试几乎都接近裸机性；但实时延迟也比较高，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。而 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 上的系统的性能，用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；目前与我们合作的小米团队也正是在做这方面的工作。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] Varanasi, Prashant, and Gernot Heiser. Hardware-supported virtualization on ARM[J]. In Proceedings of the Second Asia-Pacific Workshop on Systems, pp. 1-5. 2011.
[7] Elisei, Alexandru, and Mihai Carabas. bhyvearm64: Generic Interrupt Controller Version 3 Virtualization[R]. 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] Rossier, Daniel. EmbeddedXEN: A Revisited Architecture of the XEN hypervisor to support ARM-based embedded virtualization[J]. White paper, Switzerland. 2012.
[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] Lublin, Uri, Yaniv Kamay, Dor Laor, and Anthony Liguori. KVM: the Linux virtual machine monitor[J]. 2007.
[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] Grisenthwaite, Richard. Armv8 technology preview[J]. In IEEE Conference. 2011.
[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] Smith, Ben, Rick Grehan, Tom Yager, and D. C. Niemi. Byte-unixbench: A Unix benchmark suite[R]. Technical report. 2011.
[23] Gu, Zonghua, and Qingling Zhao. A state-of-the-art survey on real-time issues in embedded systems virtualization[J]. Journal of Software Engineering and Applications 5(4):277-290. 2012.
[24] Toumassian, Sebouh, Rico Werner, and Axel Sikora. Performance measurements for hypervisors on embedded arm processors[J]. In 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp. 851-855. IEEE, 2016.
[25] NXP Semiconductors. i.MX Linux Reference Manual, Rev. L5.4.3_1.0.0[E]. 2020.
[26] NXP Semiconductors. i.MX Virtualization User's Guide[E]. 2019.
[27] NXP Semiconductors. i.MX Linux® User's Guide[E]. 2019.
[28] NXP Semiconductors. i.MX Yocto Project User's Guide[E]. 2019.
[29] Valentine Sinitsyn. Get to know Jailhouse[R]. 2015.
[30] Bugnion, Edouard, Jason Nieh, and Dan Tsafrir. Hardware and software support for virtualization[M]. Synthesis Lectures on Computer Architecture 12, no. 1 (2017): 1-206.
[31] ARM Limited or its affiliates. CoreLink™ GIC-400 Generic Interrupt Controller[M]. 2012.
[32] ARM Limited or its affiliates. ARM® Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and version 4.0[M]. 2016.
[33] Andre Przywara. ARM: New (Xen) VGIC design document[R]. 2017.

# 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带领人，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在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

With the development of Internet of things, 5g, artificial intelligence and other technologies, the traditional automobile industry is rapidly turning to intelligent, electric and networked. Under the trend of intelligent, electric and networked automobile, the proportion of hardware cost and software cost of automobile is increasing rapidly. The number of software in cars is growing exponentially. The driving force of this development is cheaper and stronger hardware and the demand for new function innovation. However, the specific constraints of the automotive industry and the requirements of specific areas require a special solution, and bring various challenges to Automotive Software Engineering. The first software came into the automotive field about 30 years ago. 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 system software of advanced automobile, and more than 2000 independent functions are realized or controlled by the software. 50-70% of software / hardware system development cost is software cost. Hardware is becoming more and more a commodity, while software determines the function, so it becomes the dominant factor. Virtualization is the key technology to share a large number of resources effectively, safely and reliably. The traditional virtualization is mainly to allocate physical generic computing resources to multiple groups of virtual machines, and abstract these resources. Each virtual machine can run different operating systems, and the virtual machine can access the virtual hardware resources assigned to it. Each system shares hardware resources, which is independent and interactive. It not only meets the different business requirements in increasingly complex scenarios, but also improves the use efficiency of hardware resources and greatly reduces the cost. More importantly, the isolation ability between different operating systems of virtualization can greatly improve the reliability and security of the system. Virtualization technology has become one of the key technologies of the next generation of intelligent networked automotive system platform software.

Considering the requirement of low power consumption, ARM architecture is commonly used in automotive hardware platform. This paper studies the hardware platform and open source virtualization scheme based on ARM64-v8 architecture, and tries to build type-1 hypervisor Based on the real-time response and all aspects of system performance. We need to run more than two heterogeneous operating systems on the virtualization platform. According to the demand, we select the virtualization scheme suitable for automotive application from the existing open-source virtualization scheme of ARMv8, compare its performance and analyze it, so as to do a good job of preliminary technical pre research for the industrialization of automotive virtualization platform in the future.

Virtualization technology is undoubtedly one of the most popular computer system software technologies in the past decade. From the time when this technology was put forward to now, virtualization technology has emerged different implementation methods, such as complete virtualization, hardware assisted virtualization and similar virtualization. Armv8 architecture is widely used in

today's automotive hardware platform. Armv8 architecture is the largest architecture change of arm company in recent decades. It introduces completely different exception level EL0-EL3, specially designed EL2 exception level for virtualization, and uses hardware such as GICv3, SMMUv3 and stage-2 translation to do a good auxiliary support for virtualization.

KVM and Xen are very famous open source virtualization systems, and they have been ported to armv8 architecture. KVM is a type 2 virtualization hypervisor, which runs on the host system Linux as a kernel module, only realizes the virtualization of CPU and memory. KVM reuses the function of Linux which simplifies the implementation of KVM. KVM runs in a split- mode on the ARMv8 architecture, and the switch between virtual machine and high advisor involves multiple exception level switches. Xen is a type 1 hypervisor, which supports full virtualization and para-virtualization. It runs in EL2 exception level on armv8 architecture. Device IO is implemented by a modified Linux running in dom0. In addition to these two hypervisor, there are some open-source virtualization systems for ARMv8 architecture, such as Xvisor, a type 1 full monolithic kernel hypervisor, which implements all functions in the hypervisor, and the design and performance are in line with the automotive application, but the device drivers it supports are limited, and the automotive hardware has many kinds of devices, so it is very difficult to manually transplant these device drivers. Jailhouse is a static partitioning virtualization system different from the traditional virtualization system. The hypervisor only partitions the hardware resources. Each partition is called cell, and each cell runs an operating system or bare metal application.

Generally speaking, the KVM virtualization overhead of type 2 is higher than that of type 1 Xen, which is difficult to meet the real-time requirements of some software in the car. In addition, KVM runs in split mode on ARMv8. The switch between the hypervisor and the virtual machine involves multiple context switches. The performance overhead on ARMv8 is huge. It can be seen from the experimental results that the overhead of KVM hypercall is much larger than that of Xen on the ARMv8 architecture, which is not suitable for the embedded virtualization scheme. In response to the KVM problem, the operating system team of Columbia University published a paper related to this problem at the system conference ATC in 2017, which proposed a VHE (virtualization hosted extension) architecture, which has been officially adopted by ARM in the armv8.1 architecture. VHE proposes a design to extend EL2 hardware so that Linux can run at EL2 exception level, which can reduce a lot of unnecessary context switching. But unfortunately, ARMv8.1 architecture is only a design of architecture so far, and it has not been really implemented, and there is no real production of ARMv8.1 Architecture Board. Xvisor is very suitable for automotive virtualization scheme in design. Different from Xen and KVM, it is the hypervisor of type 1 full macro kernel, which implements all IO devices in hypervisor layer. The hypervisor directly processes IO events and then returns them to the virtual machine. This design is better than Xen in terms of IO device performance. Xen's IO needs to be processed through Linux in dom0, with a lot of unnecessary overhead. But there is a fatal disadvantage of Xvisor. The arm board supported by Xvisor is very limited. The IMX.8qm device prepared for this project is not supported by Xvisor, and the devices supported by Xvisor are also very limited. Although Xvisor is compatible with Linux header files and is convenient to port device drivers from Linux, porting a large number of device drivers also requires a lot of work.

To sum up, Xen and jailhouse are more suitable solutions for automotive virtualization in the current virtualization framework supporting armv8 architecture, and the project is mainly developed around these two solutions.

According to my experimental results, jailhouse is more able to meet the needs of automotive application. But it is the static partitioning hypervisor, it can not dynamically schedule hardware resources according to the workload and priority. Xen can set priority for virtual machine, and dynamically schedule virtual machine according to the priority, which can meet the needs of more hardware resources in automotive system in case of emergency of an application needs. Besides, Xen is more mature in technology. General virtualization engineers are also familiar with Xen, and it is easier to develop under Xen framework than jailhouse. If Xen can be modified to run the system in para-virtualization mode, optimize the performance of the system running on Xen, protect the real-time application with scheduling algorithm, and reduce its latency, then Xen will be a better automotive virtualization solution.

All in all, Xen and jailhouse are more suitable solutions for automotive virtualization. If some changes can be made to Xen to optimize its performance and latency, Xen may be more suitable for automotive scenes. The application of virtualization in automotive field is a very new research direction, and it is also very suitable for automotive system. It can greatly reduce the cost of automotive hardware and improve the utilization rate of hardware resources. Nowadays, there is almost no research in this field in industry and academia. In this paper, based on the characteristics and requirements of automotive system software, the performance of all aspects of the system is comprehensively considered, and the existing system is compared and analyzed The virtualization scheme under the open-source armv8 framework will do a good job in the preliminary technical pre research for the industrialization of automotive virtualization platform in the future.