Embedded Technoloy and Intelligent Systems 嵌入式技术与智能系统, 2025, 2(3), 161-168
Published Online June 2025 in Hans. https://www.hanspub.org/journal/etis
https://doi.org/10.12677/etis.2025.23013

嵌入式虚拟化技术探索及实践

张云飞[1]，吴春光[2]

[1]麒麟软件有限公司，天津
[2]麒麟软件有限公司，北京

收稿日期：2025年8月12日；录用日期：2025年10月15日；发布日期：2025年10月27日

摘　要

本文介绍了关于嵌入式虚拟化技术的主要类型和特点。针对虚拟化技术在嵌入式领域中应用的主要问题，提出了降低虚拟化损耗和提升系统实时性的方法。并针对嵌入式设备的需求，给出虚拟化下动态调频的技术方案，用于提升系统性能的同时，降低设备功耗。最后列举了两个关于不同虚拟化类型在工业场景下的实际应用。

关键词

虚拟化，实时性，设备直通

Exploration and Practice of Embedded Virtualization Technology

Yunfei Zhang[1], Chunguang Wu[2]

[1]KylinSoft, Tianjin
[2]KylinSoft, Beijing

Received: August 12, 2025; accepted: October 15, 2025; published: October 27, 2025

Abstract

This article introduces the main types and characteristics of embedded virtualization technology. In view of the main problems in the application of virtualization technology in the embedded field, methods to reduce virtualization losses and improve system real-time performance are proposed. In response to the needs of embedded devices, a technical solution for dynamic frequency modulation under virtualization is provided to improve system performance and reduce device power consumption. Finally, two practical applications of different virtualization types in industrial scenarios are listed.

文章引用: 张云飞, 吴春光. 嵌入式虚拟化技术探索及实践[J]. 嵌入式技术与智能系统, 2025, 2(3): 161-168.
DOI: 10.12677/etis.2025.23013

Keywords

Virtualization, Real-Time, Passthrough

1. 引言

虚拟化技术应用较多的是通用的云场景主要以数据中心服务器为依托，虚拟出较多的硬件资源，包括 CPU、显卡、硬盘等资源，帮助企业提高资源利用率、降低成本、提高灵活性和可靠性，从而推动企业更好地适应数字化时代的挑战。这种场景下，对虚拟化的要求主要是灵活应用、降本增效，而不过分关注虚拟化的实时性能。随着物联网、智能制造、自动驾驶等业务的增长，对虚拟化也提出了不同的要求，我们称之为实时虚拟化。在该场景下，我们需要一定的虚拟化能力，但也同样看重实时性能。虚拟化性能或者称为虚拟化损耗是衡量一项虚拟化技术的重要指标。虚拟化是在硬件层之上进行了封装，相比直接基于物理机，必然会损失一部分性能。这对于本就资源贫乏的嵌入式设备来说可能更不容易接受。另外对于一些实时性要求较高的场景，如工控领域，虚拟化损耗将比普通的云场景虚拟化显得尤为重要。

2. 调度机制

在实时性要求较高的情况下，一般都会在虚拟化层之上运行一个实时操作系统作为客户机，通过这个实时操作系统 + 强实时虚拟化来处理实时任务。当一台物理机上并行运行多个虚拟机时，物理机资源的使用率越高，虚拟机性能下降得越剧烈。这就要求实时操作系统 + 强实时虚拟化架构下的实时性指标进一步提高。在这种架构下，Guest OS 的调度器和 Hypervisor 自身的调度器共同构成了两级调度，这里称 Guest OS 对任务的调度为第二级调度，称 Hypervisor 对 VCPU 的调度为第一级调度[1]。两级调度模型抽象了 Guest OS 内部任务在 CPU 上的执行行为，这非常适合用于分析运行于 Hypervisor 上的 RTOS 是否仍然满足实时任务的实时需求。参与调度的主体主要包括客户 OS 中运行的任务 Task、客户 OS 的调度器、VCPU 以及 Hypervisor 的调度器。在两级调度框架下，Guest OS 按照自身的调度算法选取任务使之在 VCPU 上执行，然后 Hypervisor 按照 VCPU 调度算法选取 VCPU 使之能在 CPU 上执行。本文的研究重点是第一级调度，在目前的主流虚拟化软件中主要可以分为静态隔离和动态调度两种方式。

2.1. 静态隔离和动态调度

静态隔离是在虚拟机启动前，就已为其分配固定的物理资源(CPU 核心、内存、PCI 设备等)，上线后不随负载变化动态调整。各虚拟机间资源完全独占，互不干扰，适合对性能和实时性有严格保证的场景。静态隔离的性能、安全可靠性较好，所以工业场景很多情况下更适合静态隔离(成本不敏感，性能优先，持续性任务)。但这种方案的缺点是灵活性、可配置性较差，硬件共享实现困难，导致整个系统的资源利用率差。

动态调度可以根据各虚拟机/容器的实时负载，动态地分配或回收 CPU、内存、I/O 带宽等资源，可在同一物理机上运行超过物理资源总量(overcommit)，提高资源利用率，支持虚拟机的热迁移(live migration)、自动伸缩(autoscaling)等高级功能。例如在 AI 应用场景中，通过虚拟化技术，可以动态调整虚拟机的资源配置(如 CPU、内存)，以适应不同 AI 任务的计算需求，实现资源的最优利用。适合成本敏感，间

歇性任务。

2.2. 动态和静态相结合方案

动态和静态相结合方案首先是基于一个动态调度的框架。本方案将 Hypervisor 上的客户机分为两种类型，即实时客户机和非实时客户机，如图 1 所示。其中 Guest1 到 GuestN 是非实时客户机，这个类型的虚拟机更强调资源虚拟化和 VCPU 调度，满足业务场景对虚拟化的要求。RT-guest 是实时虚拟机，这个类型旨在满足硬实时的要求，去处理实时任务。RT-guest 中使用 bind 调度，使 CPU 对应的 VCPU 队列中只有一个 VCPU，这样对于 RT-guest 来说就不存在 VCPU 调度延时。相应的中断处理也可以直接分发到目标 VCPU，进一步降低了因调度导致的中断延时。在这种情况下，客户机中的物理时钟等于虚拟时钟，保证了 RT-guest 中虚拟机高精度时间应用程序的实时性能。

创建两个 CPU 池，目的是将实时客户机和非实时客户机的物理 CPU 隔离开，使两个域不会互相影响。这里假设系统一共有 n + m 个 CPU，非实时客户机使用 CPU 池 Pool-0，包括 CPU0-CPUn-1，实时客户机使用 CPU 池 rtpool，包含 CPUn-CPUn + m。Pool-0 用于客户的多虚拟化业务，可以创建多个虚拟机，这些虚拟机共用一个 CPU 池和一套硬件如(Network Interface、UART、DSIK 等)。rtpool 只用于客户实时场景，并且只能给一个实时客户机使用。在实际情况中 n 远大于 m，甚至某些场景只需要一个 CPU (m = 0)去完成实时控制业务。设置 vcpu 硬件亲和性将 rtpool 中的 CPU 指定 VCPU。

方案的关键在于在一个基于动态调度的框架下如何实现静态隔离的功能。因为在动态调度的框架下 VCPU 的调度是基于时钟的调度器去实现的。以 arm 为例，Hypervisor 通过 arm 的 generic timer 产生时钟中断，同时也为系统提供系统时钟。虚拟化的时钟可以简单分为物理时钟和虚拟时钟，正常情况下虚拟时钟等于物理时钟加偏移，但是在 bind 调度中时钟偏移等于 0，所以这时候物理时钟就是虚拟时钟。bind 调度首先关闭调度器的调度定时器，这样就不会周期性地频繁进入调度器，保证了 RTOS 系统不会被打断。每次 generic timer 产生中断以后不再不需要去维护定时器，而是直接将时钟中断注入到 RT-guest 虚拟机中，为 RTOS 系统提供调度时钟。在 bind 调度在初始化时关掉调度定时器，同时完成 VCPU 队列初始化。系统正常运行以后，CPU 将直接运行队列中的 VCPU 实体，而且没有定时器的打断，CPU 将不再由客户机陷入 Hypervisor 的调度器，大大降低了 Hypervisor 的虚拟化损耗。

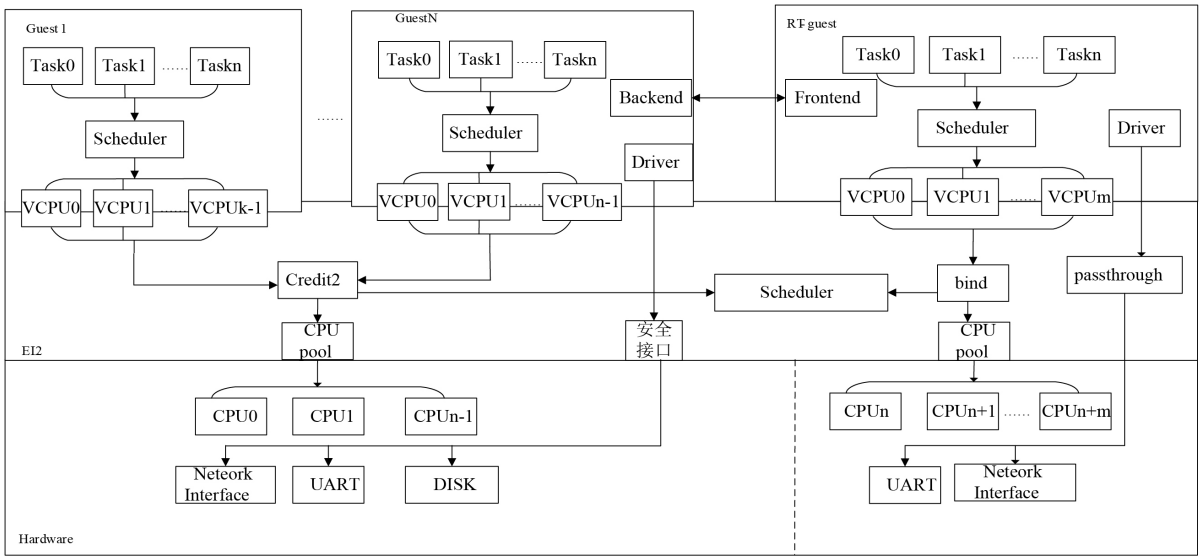

Figure 1. Diagram of the dynamic scheduling + static isolation
图 1. 动态调度 + 静态隔离图

3. ARM 处理器架构下设备直通

设备直通(Passthrough)是一项重要的虚拟化技术，它允许虚拟机(VM)直接访问物理主机上的硬件设备，而不是通过虚拟化层(Hypervisor)。这可以提供更接近物理机性能的虚拟机性能，并在一些场景下提高了虚拟化环境的灵活性。通过虚拟化管理工具或者命令行工具，管理员可以将物理设备分配给特定的虚拟机。这将设备从主机操作系统的控制下移交给虚拟机。在虚拟机生命周期结束或者管理员的操作下，设备可以被释放回主机操作系统控制。这通常涉及对设备的重置和重新分配。设备直通的一个主要优势是它可以提供接近本地性能的虚拟机性能，因为虚拟机可以直接访问硬件资源，而无需通过虚拟化层的中介。

在支持设备直通的系统中，通常需要启用 IOMMU [2]。IOMMU 是一种硬件设备，其主要功能是管理和映射设备的直接内存访问(DMA)。DMA 允许外部设备(如网络适配器、图形卡等)直接访问系统内存，以提高数据传输速度。IOMMU 的作用就是在系统内存和外部设备之间创建一个映射，以提供更好的内存管理和安全性，它为虚拟机提供一个虚拟化的地址空间，隔离虚拟机和其他虚拟机对设备的访问。软件基本都是基于 IOMMU 硬件来实现设备直通功能的。但是 IOMMU 作为一种高级硬件，并不是在所有平台上都支持。尤其是以硬件资源紧缺的嵌入式平台为甚，嵌入式系统中硬件资源有限的情况是相当普遍的，这可能包括有限的处理能力、内存容量、存储空间和其他资源。这种资源的紧张性可能由于多种原因引起，包括功耗要求、成本约束、尺寸限制以及应用场景。基于这种情况，使得虚拟化的设备直通技术在嵌入式场景下难以应用落地。而嵌入式场景又是一个对性能和实时性要求极高的领域，所以这种场景下迫切的需要一种无需硬件 IOMMU 支持的设备直通虚拟化技术。

在虚拟化的传统方案(IOMMU 支持)如图 2 所示 IOMMU 场景下，CPU 的 MMU 地址映射分为两个阶段 Stage1 和 Stage2。Stage1 基于操作系统页表将 CPU 的虚拟地址 VA，转换为中间物理地址 IPA，Stage2 基于 Hypervisor 的页表将 IPA 转换为真正的物理地址 PA [3]。Hypervisor 本身是一套全虚拟化平台，操作系统认为自己是运行在一个真实的硬件中，它并不能感知到 IPA 的存在。对于操作系统来说，它认为的 PA 其实是虚拟化环境下的 IPA。CPU 通过 DMA engine (DMA 控制器驱动)去启动 DMA 去完成一次数据传输。以数据传输方向为内存到设备(网卡、显卡等设备)为例，CPU 将物理内存地址 PA (上文中提高，其实是虚拟化下的 IPA)、设备物理地址、数据大小等内容告诉 DMA，DMA 按照这些配置去完成一次数据传输。当 DMA 进行内存访问时，IOMMU 中已经被 Hypervisor 配置了同样的页表，也就是说 IOMMU 的 IPA- > PA 与 MMU Stage2 的 IPA- > PA 相同，所以 DMA 就可以访问到正确的物理地址。

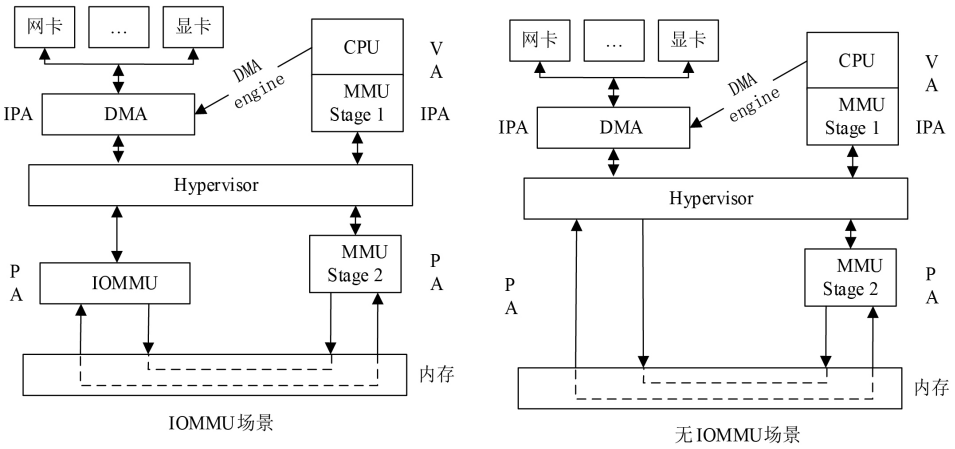

Figure 2. Diagram of memory pass-through under virtualization
图 2. 虚拟化下内存直通原理图

第一步是在创建虚拟机之前，首先通过 Hypervisor 的内存管理器为虚拟机分配静态连续内存。这样指示 Hypervisor 直接从其堆空间中分配一段静态内存，而不是在虚拟机启动以后通过写时复制的手段为虚拟机动态扩充运行内存。在这段静态连续内存上我们可以直接分配 DMA 内存缓冲区。第二步，一般动态调度类型的 Hypervisor 问题在于它的内存是动态分配的，客户机运行的地址空间一般是固定的，所以正常来说 IPA 不等于 PA。DMA 的内存访问必须将 IPA 转换为 PA 才能找到正确的数据。本文首先在客户机的数据结构中找到 Stage2 的页表。同时我们已经在第一步中得到了静态连续内存的起始物理页帧号 mfn，大小为 2^order 个页面。这里修改 gfn (客户机页帧号)等于 mfn，循环建立 mfn 到 gfn 的页表项，最终使虚拟化下的 IPA 等于 PA。如图 2 所示无 IOMMU 场景，DMA 直接使用 DMA engine 配置的内存源地址 IPA 在内存中寻址。但这时经过映射以后 IPA 已经等于 PA，所以 DMA 就可以使用 PA 找到内存中的正确数据。

4. 虚拟化下调频功能

CPU 调频允许动态调整 CPU 频率，从而在性能和节能方面都带来了好处。调频技术对于嵌入式设备来说主要为了解决以下关键问题：一是性能提升，CPU 动态调频使得在必要时可以提高 CPU 频率。这对于“重负载”用例特别有帮助，其中应用程序需要更多的处理能力。通过使用更高的频率，任务可以更快地完成，从而提升整个系统的性能。在加快引导过程，增加 CPU 频率可以加快引导速度，使系统更快地变为可操作状态。这在需要快速启动时间的设备中尤为有益。第二是节能，CPU 动态调频的一个主要优势是能够在系统负载较轻时降低 CPU 频率。在嵌入式设备的嵌入式领域尤为关键。通过降低频率，设备的能耗可以减少。在某些情况下，CPU 动态调频可以帮助防止 CPU 过热。性能优化和功耗效率的结合，使得 CPU 动态调频成为嵌入式系统中的一个有价值的功能。它与资源受限的设备要求相吻合，在性能和能耗之间取得合适的平衡至关重要。

但是在设备引入虚拟化技术以后，CPU 调频就变得大不相同[4]。如图 3 所示，调频的依据是 CPU 的负载计算，也就是说 CPU 负载大，则需要较高的频率以提高 CPU 的算力和实时性。当 CPU 的负载较小时需要降低 CPU 的频率，以减少能量的消耗。开启虚拟化以后，会在系统之上虚拟出多个客户机，不同的客户机分处不同的域，并且相互隔离。对于特权客户机来说，它拥有整个系统的硬件资源权限，这其中就包括 CPU 调频驱动及相应的硬件权限。但特权客户机操作系统只能计算自身虚拟机的负载情况，并不能感知其它虚拟机的负载。如果让特权客户机直接去进行 CPU 动态调频的话，就不能做出正确的决策，影响其它客户机的运行。另外还有一种思路是在 Hypervisor 中实现 CPU 动态调频功能。因为 Hypervisor 了解每个客户机和物理 CPU 的运行情况，所以它来做 CPU 调频的决策是比较正确的。但问题主要是像 TYPE-I 型 Hypervisor 基本上都具有轻量化的特点。它是处在硬件和操作系统之间薄薄的一层。而如果 Hypervisor 要去完成整个 CPU 调频功能的话，需要集成大量的 CPU 调频驱动。因为不同芯片的 CPU 调频驱动千差万别，而为了适配不同的硬件，其代码量是相当巨大，而且考虑到后续的维护，也有很多的工作量。将大量的驱动代码移植到 Hypervisor 中，显然不是 TYPE-I 型虚拟机的初衷。本文的思路是尽量地使用客户机操作系统的现有功能。以 Linux 系统为例，现已形成了比较完善的 CPU 调频算法，并基于 Linux 系统的强生态特性，可以适配绝大部分的 CPU 调频驱动。如果要使用操作系统自己的 CPU 调频系统，最重要的是要解决特权客户机无法感知其它客户机和实际 CPU 负载的问题。

首先是关键指令探测，以 Ondemand (Linux 系统负载动态调整频率策略)调频策略为例。通过选定的 governor (调频控制策略)去调用 od_dbs_update 函数去执行定期的负载检测和频率调整。其中 od_dbs_update 调用 od_update 函数，在 od_update 中调用真正的负载计算函数 dbs_update，然后根据负载计算结果去调用调频驱动，最终改变 CPU 的硬件频率。所以我们这里需要打桩的函数是 dbs_update。使用内核提

供的 kallsyms_lookup_name 函数找到 dbs_update 的内存地址，该函数是利用内核中 kallsyms 功能。内核将系统中用到的函数符号和地址存储到 proc 文件系统中去，然后通过 kallsyms_lookup_name 函数就可以找到内核符号和地址的对应关系。然后进行负载计算重定向，在该地址上进行指令替换，将函数替换为 ARMv8 的 HVC 指令，通过该函数可以陷入 EL2 模式下的 Hypervisor 代码。由于 Hypervisor 中掌握了真正的 CPU 硬件资源，所以我们可以在这里实现实际物理 CPU 的负载计算。Hypervisor 中的负载计算模块将计算结果在虚拟化层返回(通过 eret 指令返回之前的模式)的时候传递给特权客户机。特权客户机根据计算结果调用调频驱动，实现 CPU 的动态调频。Ondemand 会根据采样频率周期性地去调节 CPU 的频率，而每次调频都会触发到 Hypervisor 中去。这样虽然是借用特权客户机的调频系统，但其实际的决策参数(负载)是由 Hypervisor 提供的，保证了 dom0 能客观地完成调频工作。最后 Hypervisor 中负载计算模块的实现。在 Hypervisor 中为每个物理 CPU 维护了一个 VCPU 的队列，VCPU 按照优先级递减的顺序依次排列在队列中。同时，每个物理 CPU 还有一个 IDLE VCPU，它是优先级最低的 VCPU 任务，在 CPU 空闲时调度，负责对 CPU 状态的一些信息统计和数据收集工作。在这些信息中包含了 IDLE VCPU 的运行时间的统计。IDLE VCPU 的运行时间就是整个物理 CPU 的空闲时间。则负载率计算如下：

$$\text{负载率} = \frac{\text{采样周期} - \text{IDEL VCPU运行时间}}{\text{采样周期}}$$

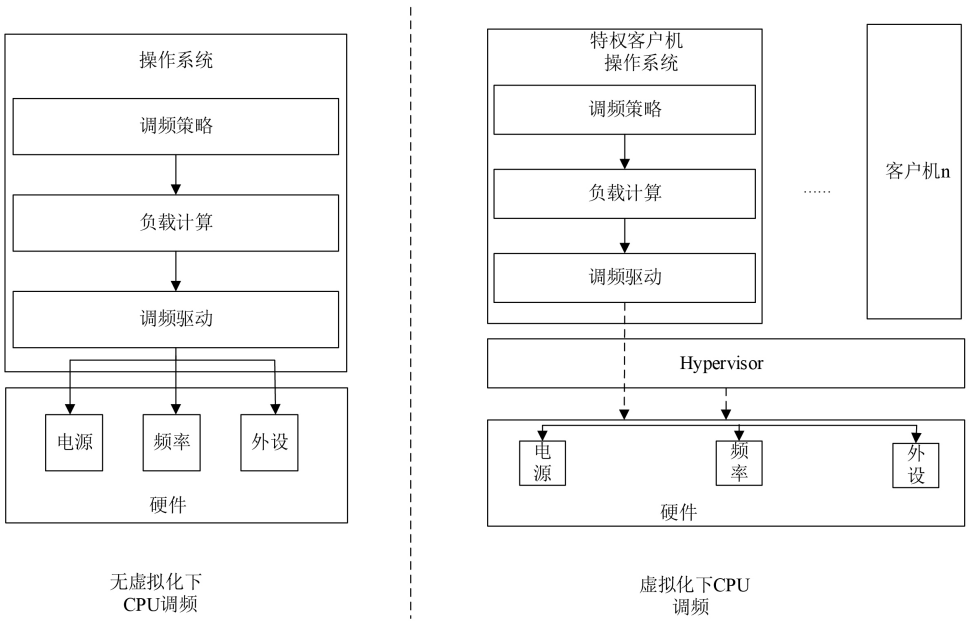

Figure 3. Diagram of CPU frequency modulation after virtualization
图 3. 虚拟化以后 CPU 调频对比图

5. 解决方案及应用案例

5.1. 采用静态隔离虚拟化的电力通信 TTU

一般来说电力网关、数据采集器(DPU)、TTU 等设备为了采集更多的设备信息就会设计多路高速串口。例如现场中遇到很多 TTU 都设计为 32 路串口或者更多。而这些串口通过总线与外部通信的波特率很高，最快的达到 2M bps。这就给 CPU 和操作系统的实时性带来很大的挑战，极易发生丢包等问题。另外这些终端和网关设备除了高速通信以外，还要负责网络通信、人机交互、本地遥测、日志存储等其它

复杂功能。传统方案一般使用实时 Linux 系统作为这些终端设备的操作系统。但是 Linux 本身的实时性较差，即使是经过实时补丁修正以后也不具备硬实时的能力。尤其是当系统负荷较重时，对实时性的影响更加严重。

对于这样实时性要求严苛且不涉及硬件资源的灵活调度的场景下，特别适合静态隔离虚拟化。它可以解决电力通信场景下，多路高速串口通信实时性不足的问题。通过改进设备的操作系统，使其具备 Linux 系统丰富生态和强大功能，同时具备更强的实时性，保证通信的安全可靠。

通过静态隔离虚拟化将 Linux 系统和 RTOS 系统混合部署到 SOC 上并将系统分为功能域和实时域。功能域运行 Linux 系统，负责设备常规的非实时业务，实时域运行 RTOS 系统进行高速串口的采集。RTOS 系统通过中断 + 轮询的方式处理大量的串口数据，并将串口数据按照串口 ID 写到共享内存的对应通道。当共享内存通道中的数据达到一定阈值以后触发 sgi 中断，通知 Linux 侧读取串口数据。这样做最大的好处是阻止了 Linux 系统因访问串口设备而造成的频繁的系统陷入以及频繁的中断造成的异常切换，这将大大减少了操作系统的损耗，增强系统实时性和减少 CPU 的占有率。Linux 侧通过编写一个虚拟串口驱动，将共享内存通道作为虚拟串口的设备层，通过平台总线设备驱动封装以后变成一个 Linux 系统的正常串口。这样原来的 Linux 侧业务就能无缝衔接过来。

5.2. 采用动态调度虚拟化的基站冗余备份

基站冗余备份是移动通信系统中确保服务可靠性和连续性的关键技术。传统方案是双机热备， 即采用两台或多台硬件设备，处于主备模式。主基站负责正常业务处理，备用基站在主基站出现故障时接管其任务。使用嵌入式虚拟化技术，将基站单元分离为不同的虚拟机。在一个虚拟机出现问题时，可以快速切换到备份的虚拟机。在正常情况下，备用虚拟机没有太多的 CPU 负载。这时可以通过调度虚拟化为其分配较少的 CPU 和内存资源。只有发生主备切换时才会被调度更多的硬件资源。

主备切换允许几百毫秒甚至秒级的延时，使用动态调度虚拟化在进行实时增强以后，完全可以达到这个指标。同时允许硬件资源动态调整，备份虚拟机只需要占用很少的资源就可以完成信息同步工作，提高了硬件利用率。

6. 总结和展望

1) 未来需要一套可在静态隔离与动态调度之间平滑切换的策略框架，避免粗粒度模式切换带来的抖动与延时。通过 tickless (NO_HZ) [5]和实时守护功能，在静态隔离时关闭调度时钟，减少周期性唤醒与 OS 抖动；在动态调度时再按需恢复时钟与抢占；实时性守护则以周期性验证 vCPU 周期/配额达成率，配合迟滞(hysteresis)逻辑抑制“频繁摆动”。

2) GICv4/v4.1 直注：利用 vPE 映射与 doorbell 机制，将外设 LPI 直接注入目标 vCPU，绕开 Hypervisor 的中断射线与 LR 编程开销，显著降低注入延迟与 VMEXIT 次数[6]。

参考文献

[1] 吕孟军. Xvisor 虚拟机管理器[D]: [硕士学位论文]. 合肥: 中国科学技术大学, 2023.

[2] 黄宇飞. ARM 平台下的虚拟化实现及应用[D]: [硕士学位论文]. 武汉: 华中科技大学, 2019.

[3] Arm System Memory Management Unit Architecture Specification SMMU Architecture Version 3. https://developer.arm.com/documentation/ihi0070/latest/

[4] Approach for CPUFreq in Xen on ARM. https://static.sched.com/hosted_files/xensummit18/ec/xen_summit_cpufreq_on_arm.pdf?_gl=1*1ae-owoq*_gcl_au*NTM4ODcyNjAwLjE3NjA5MzEyNTA.*FPAU*NTM4ODcyNjAwLjE3NjA5MzEyNTA

[5] Linux Kernel Documentation—NO_HZ: Reducing Scheduling-Clock Ticks.

https://docs.kernel.org/timers/no_hz.html

[6] GICv4.—Direct Injection of Virtual Interrupts.
https://developer.arm.com/documentation/107627/0102/GICv4-1---Direct-injection-of-virtual-interrupts