



北京航空航天大学
BEIHANG UNIVERSITY

Proj351

基于异构多核平台的机器人混合部署系统及实时优化

姓名	年级	联系方式
任航麒	大三	15303488118
肖灿	大三	13370181328
崔翰彧	大三	16633660366

指导老师：王雷

2024 年 5 月 30 日



完成进度

1. 能够在搭载实体开发板的机器人上基于 Linux 系统运行 ROS(ROS1, ROS2 均可)，使其支持小车遥控、建图和导航等的功能。
2. 能够在搭载实体开发板的机器人上运行 Shyper，使其支持启动至少两个虚拟机
3. 能够剥离机器人的实时任务，并能在不同 VM 上运行
4. 了解 ROS1 通讯原理，基于 Shyper 对实时支持进行优化
5. 搭建的基于 Hypervisor 的 ROS 控制框架可以在无人车上部署，并提供实际测试结果验证对 ROS 框架优化的效果
6. 基于现有的 unikernel，搭架 ROS 的运行环境，使其能够支持 ROS 应用（可选拓展）
7. 基于 Rust 自行实现一个支持网络协议栈的 unikernel，并完成上述功能（可选拓展）

初赛阶段完成情况如下：

目标	完成情况	说明
ROS1 运行	完成	运行实验机器人基本功能
Shyper 上启动多 vm	完成	可以启动多个 Linux
剥离机器人实时任务	完成	分布 ROS 应用节点，保证机器人正常运行
了解 ROS1 通信原理	进行中	与重写 ROS 通信机制同步进行
部署 Hypervisor 并测试对	进行中	初步测试了 shyper 的内核



ROS 优化		时延和基于共享内存的通信时延等
基于 unikernel 搭建 ROS 运行环境	进行中	Qemu 上 Unikernel+ROS 运行成功，暂未上板测试
unikernel + 网络协议栈	进行中	学习中

目 录

第一章 绪论	1
1.1 研究背景及意义	1
1.2 课题国内外研究现状	2
1.2.1 ROS 实时性研究	2
1.2.2 混合关键系统研究	5
1.3 研究目标和内容	7
1.3.1 研究目标	7
1.3.2 研究内容	7
1.4 论文主要工作及贡献	8
第二章 相关理论技术分析	10
2.1 ARM Hypervisor	10
2.1.1 Type-1 Hypervisor	10
2.1.2 ARMv8 体系架构及虚拟化拓展	10
2.1.3 ARMv8 内存虚拟化	11
2.2 Linux Kernel	13
2.2.1 Linux Kernel and Kernel Module	13
2.2.2 Linux 设备模型	14
2.3 混合关键系统	15
2.3.1 混合关键系统综述	15
2.3.2 ROS	16



2.4.3 ROS 混合关键系统中的实时性	18
第三章 ROS 混合关键系统设计	19
3.1 机器人硬件系统	19
3.1.1 JetRover 与 RK3588	20
3.1.2 电路与通信接口设计	22
3.2 机器人软件系统	24
3.2.1 Shyper + Linux + Unikernel 架构设计	25
3.2.2 实时性分析	27
3.3 本章小结	31
第四章 虚拟机间的共享内存与消息传递	32
4.1 通信模型	32
4.2 Shyper CLI、Linux Kernel Module	34
4.3 Shyper 内存映射	38
4.4 建立共享内存和通信过程	40
4.4.1 系统启动阶段	40
4.4.2 共享内存建立过程	41
4.6 本章小结	45
第五章 ROS 通信拓展	46
5.1 ROS 通信拓展	46
5.1.1 ROS 原生通信机制	46
5.1.2 共享内存的通信拓展	48
5.2 实时进程在 Unikernel 的适配	52
5.2.1 Unikernel MicroROS	52
5.2.2 ROS 实时进程移植	53
5.3 Linux + Unikernel 下对 ROS 节点的实时调度	56
5.4 本章小结	58
第六章 性能测试	59
6.1 测试平台说明	59



6.2混合关键系统实时性能分析	59
6.2.1 虚拟化后原 VM 的实时性能	59
6.2.2 LMBench	60
6.2.3 不同数据量下的跨 VM 传输时延	61
6.2.4 不同数据量下的响应时延表现	63
6.3 ROS 应用实时性能	64
6.3.1 ROS_Bench	64
6.3.2 实时性性能在多接收节点下的表现	66
6.4 本章小结	67
总结与展望	68
参考文献	70
附录 A 脚本及测试源码	72
附录 B 项目成果展示视频	76



第一章 绪论

1.1 研究背景及意义

机器人系统的当前研究趋势表明了其不断向着智能化和多功能化发展。随着深度学习和强化学习等人工智能技术的飞速进步，机器人系统在感知、决策和执行等方面取得了显著进展。研究人员着重于提高机器人系统的自主性和适应性，使其能够更好地应对复杂、动态的环境。同时，机器人系统的研究也不断涉及到人机交互、情感识别等方面，以提升其与人类的互动体验和合作能力。在硬件方面，机器人系统的传感器、执行器等关键部件也在不断创新和优化，以提高其性能和稳定性。在软件方面，机器人系统的软件控制架构、智能交互、机器视觉、计算实时性等逻辑部件也在不断更迭和升级。

实时性是当前机器人系统研究中的一个重要议题。随着机器人应用场景的不断扩展，对实时性能的需求也日益迫切。研究人员正在探索高效的实时控制算法和系统架构，以确保机器人系统能够在复杂、动态的环境中做出及时的响应和决策。在这一需求中，涉及到传感数据的实时处理、运动规划的实时性优化、以及实时通信和协作等方面的技术挑战。

在当下车载、机器人等嵌入式场景中，Linux 已经得到了广泛的应用，但实际上并不能覆盖所有的如高实时、高可靠、高安全等场合。在这些场景下通常需要优先考虑的是 RTOS（实时操作系统）。但有些应用场景下会既需要 Linux 的管理能力、开发效率、丰富的生态又需要实时操作系统的高实时、高可靠、高安全，在这种场景下的设计通常要考虑在同一嵌入式场景下同时运行 Linux 与 RTOS，并对实时性安全性可靠性要求较高的任务运行在 RTOS 上。现在一种典型的设计和在某些产品化的场景下会采用一颗性能较强的处理器运行 Linux 以支持丰富的功能，同时采用一颗微控制器



/DSP/实时处理器运行实时操作系统负责运行车载的基本传感器信号接收处理以及直接的实时控制，两者之间通过 IO、网络或片外总线等通信。但这种方式存在的问题是两套系统实际是独立存在于系统上的，这种物理存在分离的角度下，由于设备会单独连在某个确定的控制器上就导致很多非通信等设备在片上无法直接共享，同时会受限于具体开发板的情况和开发模式而在二者之间的通信性能很难达到一个理想的效果，且在整个系统的灵活性上与可维护性上存在很大的改进空间。

Hypervisor 作为一种关键的虚拟化技术，在计算机领域的研究和应用中扮演着重要角色。Hypervisor 可以将物理计算机的资源虚拟化成多个独立的虚拟机实例，从而使得多个操作系统和应用程序能够在同一台物理主机上并行运行，提高了硬件资源的利用率和灵活性。通过 Hypervisor 可以灵活地配置 VM 的资源，并可以对片上的设备以统一视角的方式灵活地进行直通/模拟，对混合关键系统场景是一个非常恰当的解决工具。

在机器人场景中，会出现很多对实时性要求较高的情境，如人形机器人步态闭环控制，必须要求在下一步迈出的间隔内将下一步的步态方案计算完才能保证机器人稳定运行。目前 ROS 主要基于 Linux 开发，但 Linux 本身的实时性非常差，如果想获得硬实时性就必须与一些 RTOS 结合使用，然而 ROS 上层的用户代码生态非常庞大，特别是 ROS1，目前 ROS 开发者主要仍在使用 ROS1，且已经产生了很多与系统甚至内核强绑定的非标准应用，因此将这一生态完整兼容且为应用赋予实时性保障是一个巨大难题。

在上述场景下，Hypervisor 可以将上层 Linux 发行版和 ROS1 应用生态完整地通过虚拟化兼容，这样仅需代码的微小改动，便可以将实时的部分剥离出来在新的 VM 上运行，并通过 Hypervisor 下调度以保证其实时性，进一步地，实时性任务通常较为简小且耦合度低，可以使用 RTOS 进一步提高系统资源利用率，增强系统的实时性和系统灵活性。

1.2 课题国内外研究现状

1.2.1 ROS 实时性研究

目前 ROS 的实时性研究非常多，因为本文的目标是期望可以保留 Linux 原有的

ROS1 生态，因此选取一些与 ROS1 生态兼容的实时性研究工作。目前 ROS 实时性工作主流仍然是使用 Linux 的实时性补丁，或直接更换实时操作系统，且仍然是使用 RT-PREEMPT^[12]、Xenomai3^[13]等实时性补丁居多，由于直接的实时操作系统与本研究各自优劣势面对的场景差异较大，且关联度不高，本文仅介绍 RT-PREEMPT、Xenomai3。

- PREEMPT_RT 的方式是期望将 Linux 内核转变为一个适用于实时性的 Linux 内核（如图 1），它主要对 Linux 做了内核抢占性更改，实时锁优化和调度器改进

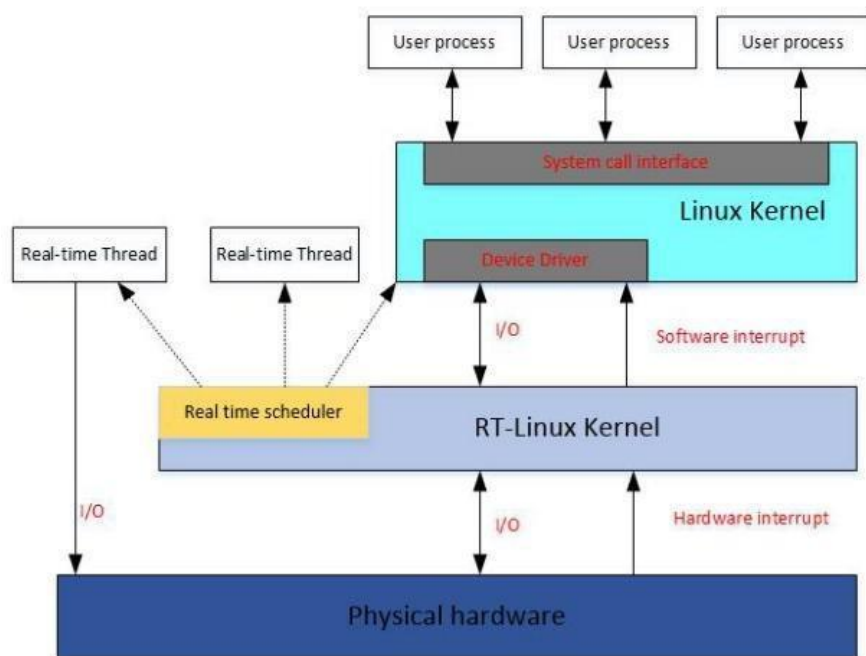


图 1. PREEMPT_RT 架构图

- Xenomai3 是在 Linux 内核上提供了另外一套 API 进行管理和调度，它主要增加了分时调度模块，与 Linux 内核的共享资源管理，硬实时性能保障，并与 Linux 内核做了兼容性改进，同时提供了 POSIX 兼容的 API，使得开发者可以使用标准的 POSIX 接口编写实时应用程序（如图 2-图 5）

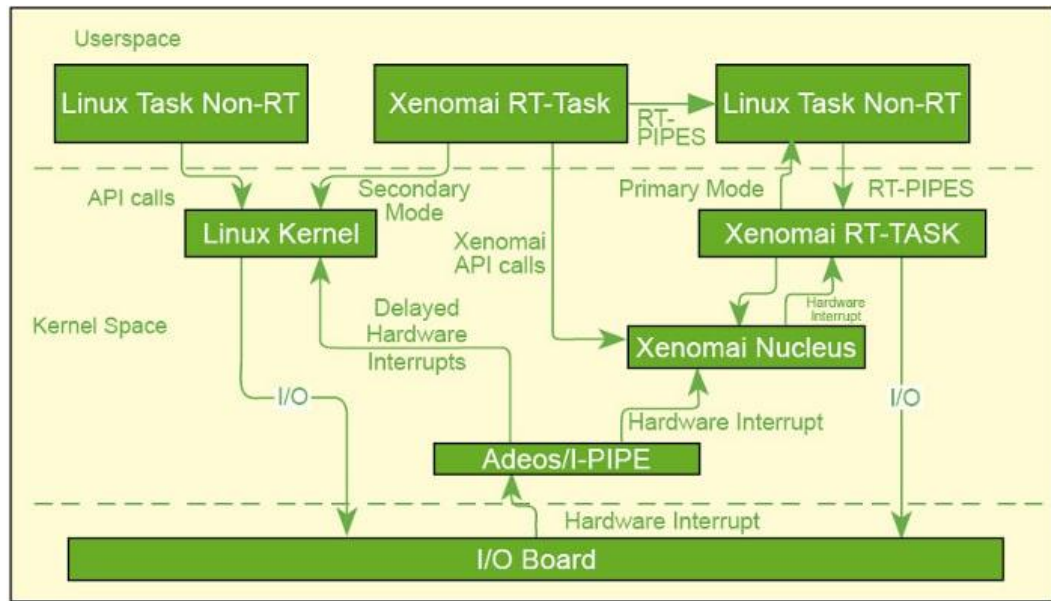


图 2. Xenomai3 架构图

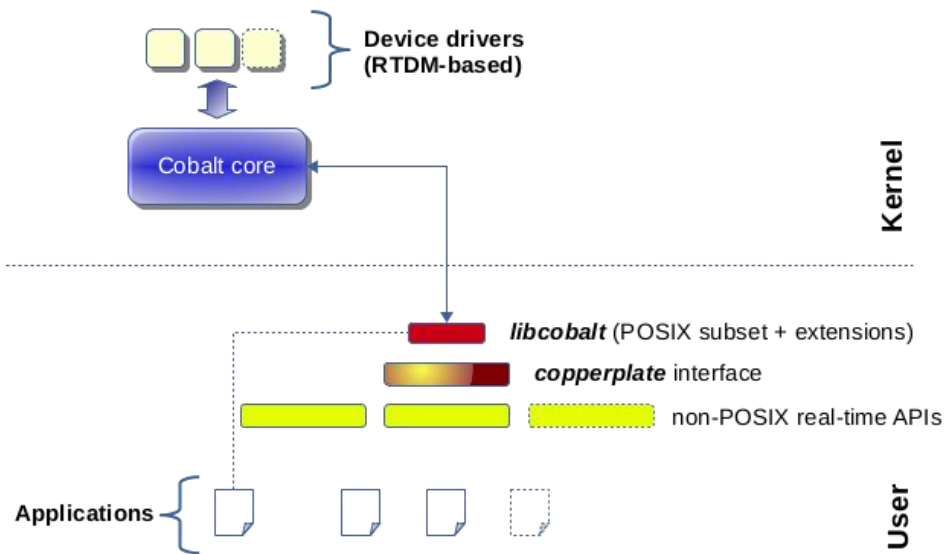


图 3. Xenomai3 Cobalt Core POSIX 兼容性设计

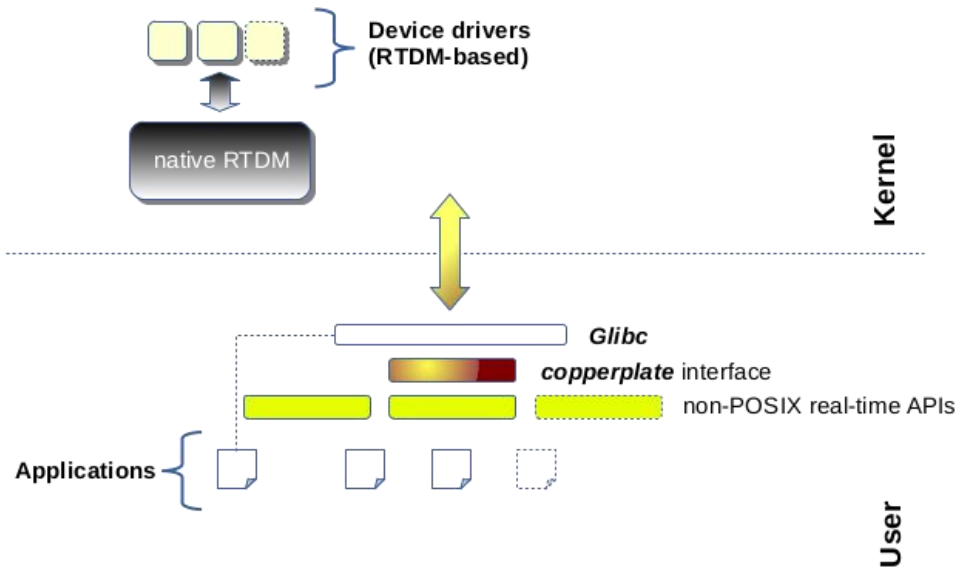


图 4. Xenomai3 Native RTDM POSIX 兼容性设计

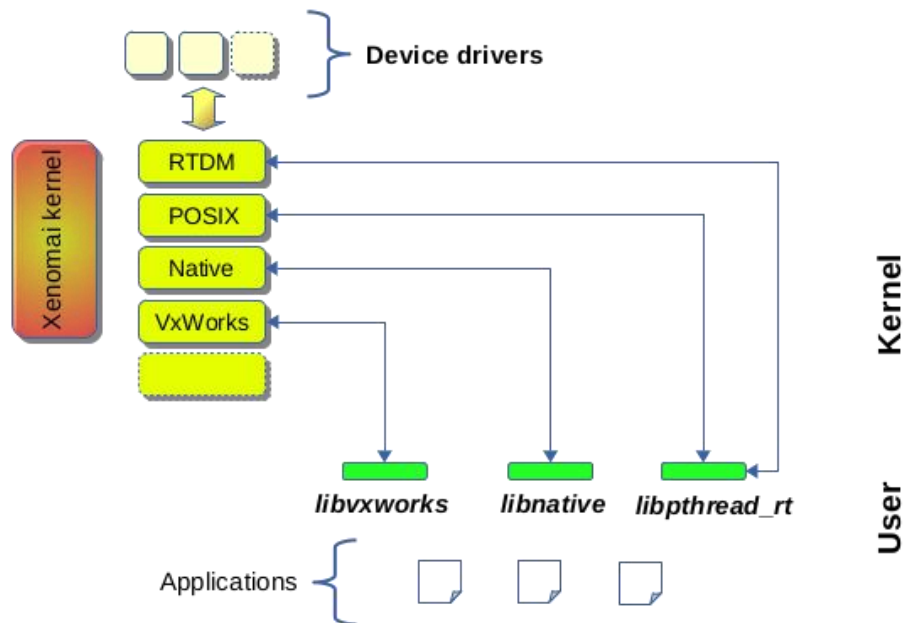


图 5. Xenomai3 lib 对应关系

1.2.2 混合关键系统研究

当前混合关键系统的研究主要集中在 Hypervisor 的实时性 baseline 的优化，和上层的混合关键系统中的 SRTVM（Software Real-time Virtual Machine）与 HRTVM（Hardware Real-time Virtual Machine）选型与软件生态的研究工作。

对于现今在 Hypervisor 中增加实时性工作的方案主要为增加调度接口和实时调度算法，优化抢占过程。但通过这种方式实现的实时性通常还需要考虑虚拟化 I/O、增加的调度延迟、共享缓存争用等问题。在 Lee 等人的工作中^[1]，使用了软实时应用程序的知识以增强响应性，并考虑了缓存亲和性，证明了缓存管理对于软实时任务的影响，并进行了负载均衡，减小了缓存抖动。在 Sundar Rajan 等人的工作中^[2]，讨论了虚拟化在当前智能汽车领域的实时应用，并讨论了系统启动、中断和陷阱处理、以及促进输入输出访问等虚拟化中的重要方面，并在最后评估和比较了虚拟化系统在核加载、中断和任务定时参数方面的性能。在 Casini 等人的工作中^[3]，作者详细介绍了三种 I/O 虚拟化技术的建模，针对不同的度量标准提供了每种技术的分析界限，并量化了不同延迟源的贡献。在 Jiang 等人的工作中^[4]，根据该团队在先前工作提出的实时 I/O 虚拟化管理程序（VCDC）、I/O 控制器（GPIOCP）和内存互连（BlueTree）开发了 BlueVisor，使 CPU、内存和 I/O 的虚拟化具有可预测性，并具有快速的中断处理程序和虚拟机间通信功能。在 Xi 等人的工作中^[5]，提出了首个针对 Xen 的实时调度框架，RT-Xen 通过实例化了一套固定优先级服务器（可推迟服务器、周期服务器、轮询服务器和间歇服务器），该文章广泛的实验结果展示了在 RT-Xen 中使用固定优先级层次化实时调度的可行性、效率和功效。

对于 Hypervisor + HRTVM 架构当下的研究并不多，但已有相关方向的工作，并有提出该架构的优化方向。

在 Yang 等人的工作中^[14]，建立了一个 Linux 与 RTOS 混合的 Hypervisor，将部分实时应用调入到 RTOS 中运行。虽然目前只支持同时运行一个 Linux 与一个 RTOS，但已经有类似的雏形了。在 Lupu 等人的工作中^[15]，作者优化了在 Hypervisor + Unikernel 架构下，Unikernel 对于 POSIX 应用特别是 fork() 相关支持的问题，并为其研究的 Unikernel Nephele 拓展在 Xen 平台上，同时为基于 Unikernel 的虚拟机提供了自动扩展能力，将 Nephele 的实例化速度提高了 8 倍，并且相比于单独启动的 Unikernel，可以在相同硬件上运行 3 倍更多的活跃 Unikernel 虚拟机。

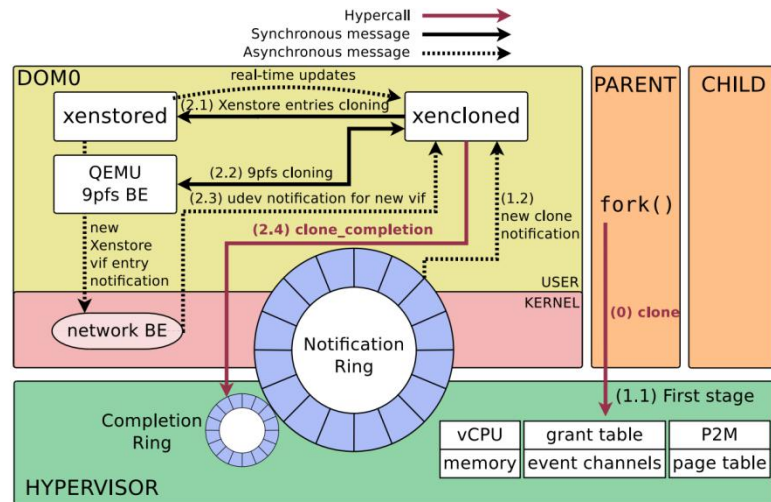
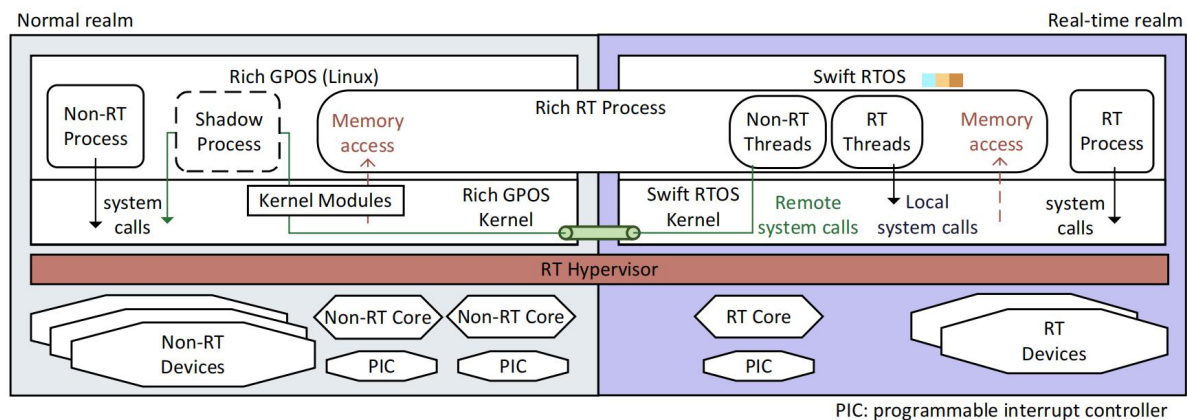


图 6. Unikernel fork 设计



PIC: programmable interrupt controller

图 7. RTOS Mixed 设计

1.3 研究目标和内容

1.3.1 研究目标

在 Shyper 与 Unikernel 的基础上进行迭代开发，移植 ROS 后进一步改写、适配 ROS 的用户应用，使其具有与普通 ROS 相类似的运行能力，并可通过常见的 ROS 应用，如建图导航，物体跟踪等测试其实时性改善效果。进一步地，基于实时性理论分析该混合关键型系统的实时性，并与实际情况对比分析实现效果。然后基于项目开发过程和实现方案给出该研究的评价和应用前景。

1.3.2 研究内容



整个混合关键系统可以分为 Hypervisor、Linux、Unikernel、ROS 的四部分工作。

Hypervisor 部分的工作是需要增加基于共享内存部分的话题调用，以及 ROS 进程注册时，新 VM 的启动，并基于 POSIX 接口实现一套共享内存申请调用，以及锁和信号量的基本同步机制。进一步地，基于 GVM 的启动和其运行的进程情况，加入基于 MLFQ 的实时调度算法。

Linux 部分 (MVM) 需要补充内核模块的功能，将需要跨 VM 的 HVC 调用补充一个内核调用接口。同时需要在 MVM 上维护一个共享内存部分的 ROS 话题机制，并补充一个将 ROS 应用打包成 Unikernel 线程的方式，并自动生成其配置文件的脚本。

Unikernel 部分需要实现的首先是 ROS 软件的移植，将 ROS 用户应用剥离出来的实时部分改写 to Unikernel 上，并为 Unikernel 补充部分 HVC。同时 Unikernel 当前在 Shyper 中的输出不能与 MVM 保证同时回显到终端，需要移植 virtio-console 保证 Unikernel 部分的 ROS 应用支持调试。对于共享内存通信，Unikernel 中也需要补充类 POSIX 的方式，完成与其他 VM 间的通信和同步。进一步，为了能够更好地完成在 Linux 的开发和在 Unikernel 的实时线程切换，需要为 Unikernel 补充一个执行线程切换的消息通信机制。

ROS 部分的工作主要是需要额外提供一套基于共享内存实现的通信接口和同步机制，也因此需要与原 ROS 共同维护原有的消息架构。在此基础上，将机器人用户应用的实时部分剥离出来，并将实时部分移植到 Unikernel 上。

1.4 论文主要工作及贡献

本文的主要贡献如下：

- 提出了 Hypervisor + RTOS 应用于改善 ROS 实时性的应用范式，并使用 Shyper + Unikernel 实现一套范式系统
- 完成 MicroROS 在 Unikernel 上的移植，并增加将 ROS 进程单独封装成一个 Unikernel 线程的方式
- 基于当前系统提供一套增加实时启动的 ROS 运行方式，动态配置 GVM 的配置项
- 剥离 ROS 基础应用中的实时部分，并在 Unikernel 上完成适配



- 基于 POSIX 实现了 MVM（第一台虚拟机——Linux）和 GVM（Unikernel）之间的共享内存通信，以进一步改善该情境下的通信性能

第二章 相关理论技术分析

本章首先介绍在本文中的关键系统组件，包括 Hypervisor、Linux Kernel Module、Linux 与 RTOS 设备子系统、以及 ROS 混合关键系统，并简易说明了系统的实时性方案。

2.1 ARM Hypervisor

2.1.1 Type-1 Hypervisor

Type-1 型 Hypervisor（如图 8），也被称为裸机（bare-metal）Hypervisor，是一种直接运行在物理硬件上的虚拟化软件。与 Type-2 型 Hypervisor 不同，Type-1 型 Hypervisor 不依赖操作系统，而是作为最底层的软件运行在物理服务器上，控制和管理硬件资源的分配并提供虚拟化服务。它提供了一个虚拟化层，通过这个层可以运行多个虚拟机（VM），每个虚拟机都有自己的操作系统和应用程序。这种架构使得 Hypervisor 可以高效地管理硬件资源，并提供良好的性能和安全性。

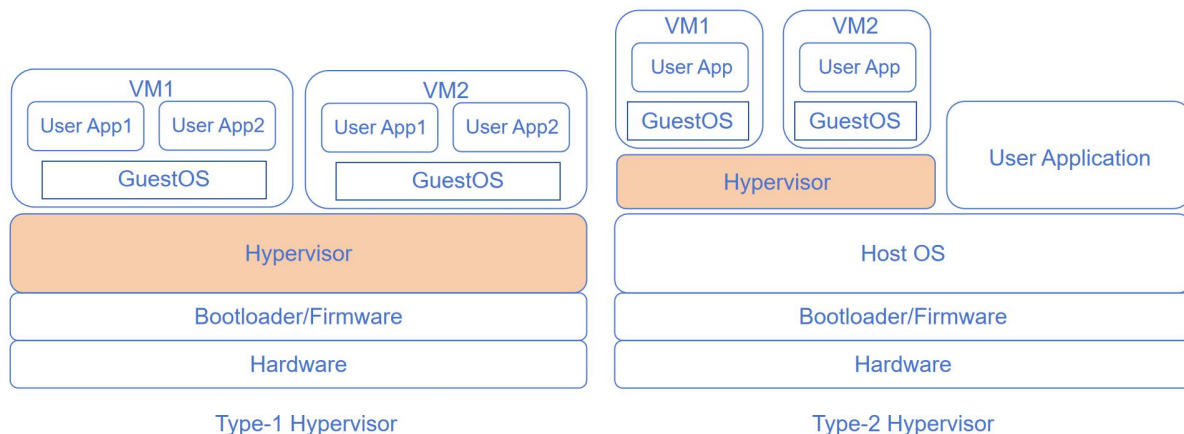


图 8. Type-1 Hypervisor

2.1.2 ARMv8 体系架构及虚拟化拓展

先进各个体系结构在虚拟化方面都会有对应的虚拟化拓展以支持更高性能、更高安全、更强功能的虚拟化设计。在本项目中使用的体系架构为 ARMv8，ARMv8 在虚拟化支持上主要有以下几点：



- 异常级别：ARMv8 引入了四个异常级别（EL0 到 EL3），EL0 指用户模式（User mode），用于运行应用程序；EL1 指内核模式（Kernel mode），用于运行操作系统内核；EL2 指虚拟化管理模式（Hypervisor mode），用于运行虚拟机管理程序；EL3 指安全监控模式（Secure Monitor mode），用于处理安全相关的操作。
- 虚拟化状态寄存器：虚拟化处理器状态寄存器 VMPIDR_EL2、虚拟化控制寄存器 HCR_EL2 等
- 硬件支持的二级地址地址转换：包括 Stage2 TLB、拓展后含 VMID 标记位的页表、二级页表翻译的基址寄存器 VTTBR_EL2
- 中断虚拟化支持：ARMv8 的虚拟化扩展允许虚拟机管理程序虚拟化中断控制器，使得每个虚拟机可以拥有独立的中断控制器视图；同时 ARMv8 架构的中断可以根据配置路由到虚拟机管理程序或直接交付给虚拟机
- 安全虚拟化支持：ARMv8 架构支持 TrustZone 技术，将系统划分为安全世界（Secure World）和非安全世界（Non-Secure World），并允许虚拟机管理程序在安全世界中运行，提供更高的安全性；安全异常级别（Secure EL2）：引入了一个专门用于安全虚拟化的异常级别，进一步增强了安全性；改进的 MPU 和 MMU：增强了对内存保护和管理的支持，提供更细粒度的内存访问控制和更高效的内存转换机制。

2.1.3 ARMv8 内存虚拟化

本文的在虚拟化方面的工作主要集中在内存虚拟化，在 ARMv8 内存虚拟化下，主要包括以下几点：

- 两层地址转换机制
 - 第一层地址转换（Stage 1 Address Translation）
 - ◆ 操作系统（OS）管理：第一层地址转换由操作系统管理，将虚拟地址（Virtual Address, VA）转换为中间物理地址（Intermediate Physical Address, IPA）。



- ◆ 页表 (Page Tables) : 操作系统使用页表来管理虚拟地址到中间物理地址的映射。页表存储在内存中, 并由内存管理单元 (MMU) 使用。
- 第二层地址转换 (Stage 2 Address Translation)
 - ◆ 虚拟机管理程序 (Hypervisor) 管理: 第二层地址转换由虚拟机管理程序管理, 将中间物理地址 (IPA) 转换为实际物理地址 (Physical Address, PA) 。
 - ◆ 二级页表 (Stage 2 Page Tables) : 虚拟机管理程序使用二级页表来管理中间物理地址到物理地址的映射。这一层增加了一层抽象, 允许虚拟机管理程序控制虚拟机的内存访问。
- 地址转换单元 (Translation Lookaside Buffer, TLB)
 - 两级 TLB: ARMv8 架构支持两级 TLB, 分别缓存第一层和第二层地址转换结果, 从而减少内存访问延迟。
 - TLB 刷新 (TLB Flush) : 在虚拟机切换时, 需要刷新 TLB, 以确保不同虚拟机的内存访问不会互相干扰。
- 虚拟化支持的内存管理单元 (MMU)
 - 扩展 MMU 功能: ARMv8 的 MMU 扩展了对虚拟化的支持, 能够处理两层地址转换。MMU 负责将虚拟地址通过两层页表转换为物理地址。
 - 内存域 (Memory Domains) 和访问权限: MMU 还支持内存域和访问权限控制, 使得每个虚拟机可以有不同的内存访问权限, 增强了内存安全性。
- 地址空间控制
 - 虚拟地址空间标识符 (Virtual Address Space Identifier, ASID) : 每个进程或虚拟机可以分配一个唯一的 ASID, 用于标识其地址空间。ASID 使得同一虚拟地址在不同的虚拟机或进程中可以映射到不同的物理地址, 从而支持快速上下文切换。
 - 物理地址空间标识符 (Physical Address Space Identifier, PASID) : 类似于 ASID, 但用于物理地址空间的区分。

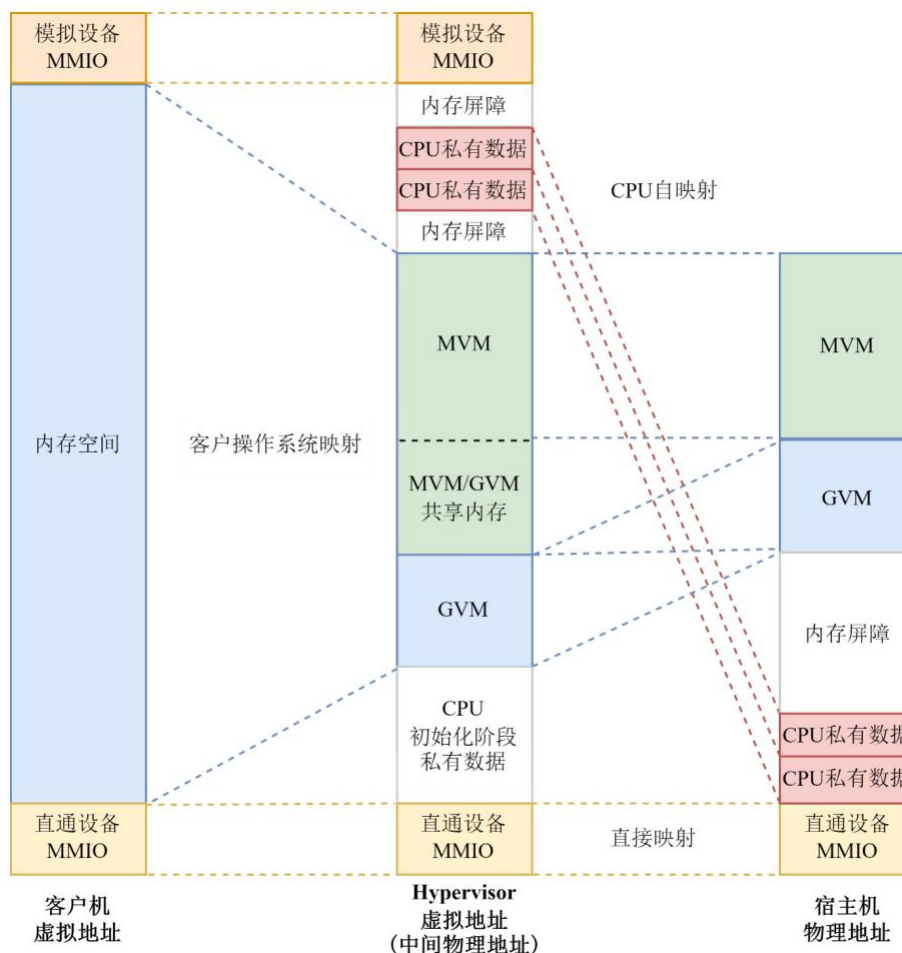


图 9. Shyper 地址映射设计

2.2 Linux Kernel

2.2.1 Linux Kernel and Kernel Module

Linux Kernel (Linux 内核) 是整个 Linux 操作系统的核心部分，负责管理系统资源并提供底层服务，使用户应用程序能够安全高效地运行。Kernel Module (内核模块) 是可以动态加载到内核中以扩展其功能的独立代码段。内核作为一个常驻内存的程序，处理系统的基本操作，而内核模块允许在运行时添加新功能或驱动程序，无需重新编译或重启内核。

通过内核模块对 Linux 内核进行开发是一个非常高效且便捷的方式，Linux 内核模块可以在系统运行时动态加载和卸载 (如图 10)，这提供了系统的灵活性并节省内存，同时内核提供了许多接口和 API，供内核模块调用，使模块能够与内核的其它部分交

互。在 Linux 为内核模块提供的交互方式下，内核模块的开发视角会是一段独立的代码，可以独立编译、测试和调试，但它们依赖于内核的某些功能和 API，需要与当前运行的内核版本兼容，实际在编译内核模块的时候会进入到相应版本内核的源码目录中进行构建。

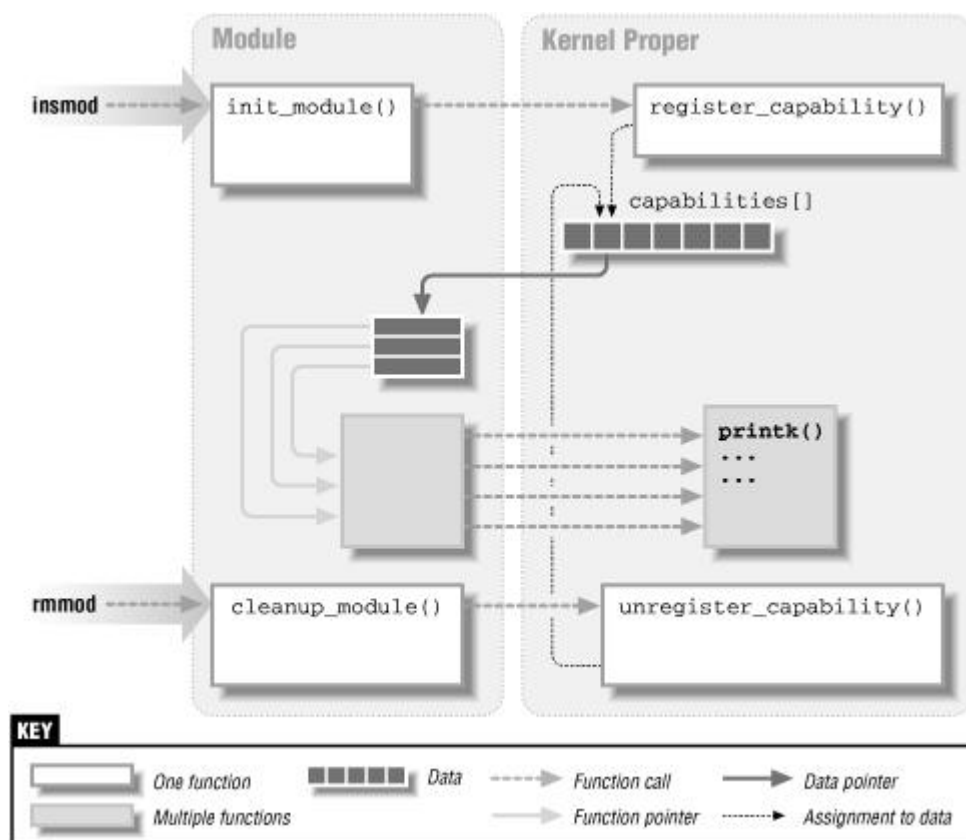


图 10. Linux Kernel Module 装载过程

2.2.2 Linux 设备模型

Linux 的设备模型是内核中用于管理和操作设备的框架。它提供了一种标准化的方法来表示和操作系统中的硬件设备，使得内核代码可以更加模块化和易于维护。在 Linux 对于设备在内核中的视角主要包括：

- 设备 (Device): 表示实际的硬件设备，比如硬盘、网卡等
- 驱动 (Driver): 操作和管理设备的代码，实现了设备的具体操作逻辑
- 总线 (Bus): 连接设备和驱动通道，比如 PCI、USB 等
- 类 (Class): 用于分组相似功能的设备，例如块设备、字符设备、MISC 设备等

- 设备文件 (Device File): 用户空间与设备交互的接口, 通常位于 `/dev` 目录下

MISC 设备 (Miscellaneous Devices) 是一种字符设备, 专门用于那些没有专门类别的小设备。MISC 设备通过一个简单的机制来分配设备号和设备文件, 适用于那些不需要复杂配置的设备。同时 MISC 设备较为灵活, 在 Linux 系统内, 所有 MISC 设备共享一个主设备号 10, 次设备号会在 MISC 设备通过 `misc_register` 注册时由内核自动分配。

MISC 设备的管理主要通过 `misc_register` 和 `misc_deregister` 两个函数来完成, 对于 MISC 设备的定义需要声明一个次设备号分配、设备名和文件操作, 文件操作即对用户态下对于设备操作的主要方式, 在 MISC 设备成功 `register` 后用户可以通过 `/dev` 目录下的设备文件通过标准文件访问接口来访问 MISC 设备 (如 `read`, `write`, `mmap`, `ioctl` 等), 对应的标准操作即会被在 MISC 设备定义 `file_operations` 的时候触发对应的自定义设备请求处理函数。

2.3 混合关键系统

2.3.1 混合关键系统综述

混合关键系统 (Mixed Criticality Systems) 是一种嵌入式系统设计范例, 旨在将不同安全性和实时性需求的软件组件整合到同一硬件平台上。在这种系统中, 软件组件被划分为不同的关键性级别, 每个级别对实时性和安全性有不同的要求。MCS 的目标是在同一硬件平台上有效地管理和满足这些不同级别的需求, 以降低成本并提高系统效率。

MCS 的核心思想是结合高安全性和高实时性需求的组件与功能, 例如 Linux 的丰富功能和实时操作系统的高实时性能力。这种系统通常采用一颗处理器运行 Linux 来处理复杂的功能, 同时运行实时操作系统来处理对实时性要求较高的任务, 如实时控制或信号处理。两者通过 I/O、网络或片外总线通信, 实现功能的协同工作。



图 13. 混合关键系统示意图

MCS 的发展受益于硬件技术的进步，特别是单核能力的提升、多核和异构多核处理器的出现以及虚拟化技术的发展。这些技术使得在同一个片上系统（SoC）中部署多个操作系统成为可能。同时，应用需求的变化，如物联网、智能化和功能安全等，也推动了 MCS 的发展，因为单一操作系统往往难以满足所有需求。

2.3.2 ROS

ROS，全称为 Robot Operating System，是一个灵活的框架和工具集，用于构建机器人应用程序。尽管名称中含有“操作系统”，但 ROS 并非传统意义上的操作系统，而是一种软件平台，提供了一系列库和工具，用于帮助开发者创建机器人软件。

ROS 的设计理念是模块化和分布式的，它允许开发者将机器人系统分解成相互独立的模块，称为节点，这些节点可以在不同的计算机上运行，通过消息传递进行通信。这种设计使得 ROS 非常适合于复杂的机器人系统，可以简化开发和测试过程，提高代码的重用性。

目前 ROS 主要有 ROS1 与 ROS2 两套框架，在机器人领域均广泛使用，但它们各自有不同的特点和应用场景。

ROS1 目前仍然是许多机器人项目的首选，因为它已经被广泛应用并且有大量的社区支持。ROS1 基于 C++ 和 Python 编程语言，采用了中间件通信库 ROS Master 来管理节点之间的通信。然而，ROS1 在一些方面存在一些限制，例如对实时性的支持较差，

通信机制相对较为脆弱等。

ROS2 是 ROS 的下一代版本，旨在解决 ROS1 存在的一些问题，并引入了许多新特性。ROS2 采用了更为灵活和可靠的通信机制，例如 DDS (Data Distribution Service)，提供了更好的实时性和安全性。此外，ROS2 还引入了一些其他功能，如多语言支持、分布式节点发现、更好的 QoS (Quality of Service) 支持等。由于这些特性，ROS2 逐渐在一些新项目中得到应用，特别是对于需要更高实时性、可靠性和安全性的应用。本项目主要是对于 ROS1 的软件生态支持并在兼容软件生态的基础上做的实时性拓展，但实际也可以收益于 ROS2，但这里主要关注 ROS1。

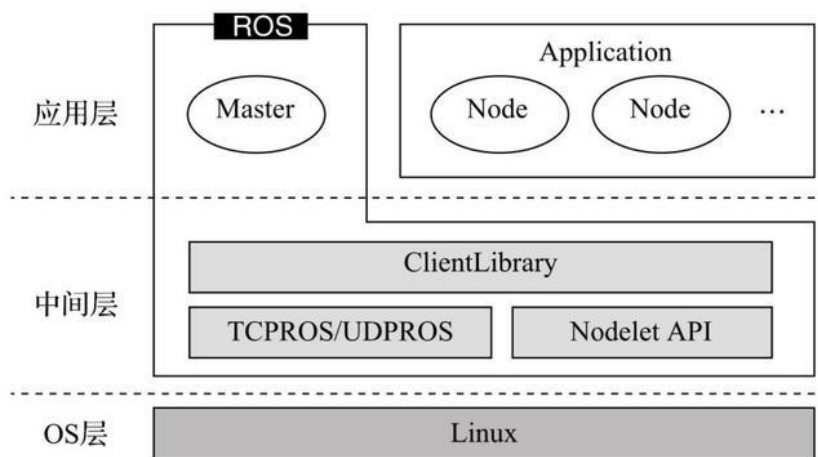


图 14. ROS1 架构图

如图 14，在 ROS1 的架构中：

应用层：ROS1 提供了一系列的工具、库和算法，用于开发各种类型的机器人应用程序，包括传感器数据处理、运动控制、导航、机器人视觉等。

中间层：ROS1 的中间层主要由 ROS Master、节点 (Nodes)、话题 (Topics)、服务 (Services) 和参数服务器 (Parameter Server) 等组成。ROS Master 负责节点之间的注册和通信管理，节点通过发布和订阅话题进行消息传递，通过服务进行请求和响应式通信，同时可以使用参数服务器进行参数配置和共享。

操作系统层：ROS1 可以运行在多种操作系统上，包括 Ubuntu、Debian、Fedora 等，通常使用基于 Linux 的发行版。ROS1 的核心功能依赖于 Linux 的一些特性和工具，如文件系统、进程管理、网络通信等。

2.4.3 ROS 混合关键系统中的实时性

在一个 ROS 应用的混合关键系统中同时存在实时和非实时（非严格实时）的部分。在这样的系统中，实时性场景通常涉及需要在严格的时间约束下执行的任务，比如机器人控制、传感器数据采集和处理等。这些任务需要在规定的时间内完成，否则可能导致系统故障或性能下降。

下面是一些 ROS 混合关键系统中的实时性场景的例子：

机器人控制：在 ROS 中，机器人控制是一个典型的实时性场景。例如，一个自主移动机器人需要在实时更新的传感器数据基础上做出快速反应，以避免障碍物或在导航中保持稳定的姿态。

传感器数据采集和处理：许多机器人系统使用各种传感器（如激光雷达、摄像头、惯性测量单元等）来感知环境。这些传感器通常以高频率生成数据，需要在规定的时间内进行采集和处理。例如，一个需要实时生成地图的 SLAM（同时定位与地图构建）系统就是一个实时性场景的例子。

通信和协作：在 ROS 中，不同的节点通过 ROS 话题、服务或行为进行通信和协作。在某些应用中，这些通信可能需要在实时性场景下完成，以确保及时的数据交换和响应。例如，在多机器人系统中，机器人之间可能需要在实时性约束下共享位置信息或执行协同任务。

第三章 ROS 混合关键系统设计

本章主要为对机器人整体架构的软硬件设计，以及混合关键系统模型框架的说明。

3.1 机器人硬件系统

当下机器人系统的硬件设计对于机器人的整体稳定性与系统开发潜力上起决定性作用，一个机器人在完成某一确定任务的执行效果与硬件所能达到的理想性能息息相关。目前机器人的硬件系统主要包括控制系统、外设单元、堆叠设计、行进方式等。

在控制单元上，目前主要可以分为上位机和下位机。

- 上位机通常是指用于进行复杂计算和控制任务的计算机系统。它通常具有强大的处理能力和丰富的软件资源。上位机的主要功能包括：复杂的路径规划、任务分配和全局决策；处理来自下位机和某些高频传感器的数据，并进行分析和决策；提供一个较为友好的人机交互界面和开发环境等，方便用户操作和监控机器人状态。

目前典型的上位机包括个人 PC 计算机、工业 PC 机、树莓派、Nvidia Jetson 系列开发板等以及其他高性能嵌入式系统。并且在现代机器人中，可能上位机可能还会利用云端的资源，借助云端的运算来处理大量数据和复杂的算法。



图 15. 上位机图样

- 下位机通常负责直接与某些实时性较高的传感器执行器等外设交互，比如 IMU、驱动板、蜂鸣器等，下位机在运行状态下会接受相应的传感器信号直接决策，或将

消息转发到上位机，具体的主要功能包括：实时控制，如直接处理运动控制，部分对实时性较高的传感器相应以及温湿度传感器等数据的初步处理；执行上位机发送的特定控制机器人运动的指令；完成某些特定的通信，如串口、CAN 总线、以太网、I2C、SPI、蓝牙等，因为多数场景下的通信需求不同，而上位机作为 PC 在很多嵌入式场景下的外设比较固定，在做测试阶段或在某些特定场景有相应的通信需求通常会通过下位机提供的丰富接口来满足。

目前典型的下位机主要包括 MCU（如 ARM Cortex 系列）、单片机（如 AVR、PIC）、以及专用的运动控制器（如 DSP 或 FPGA）。

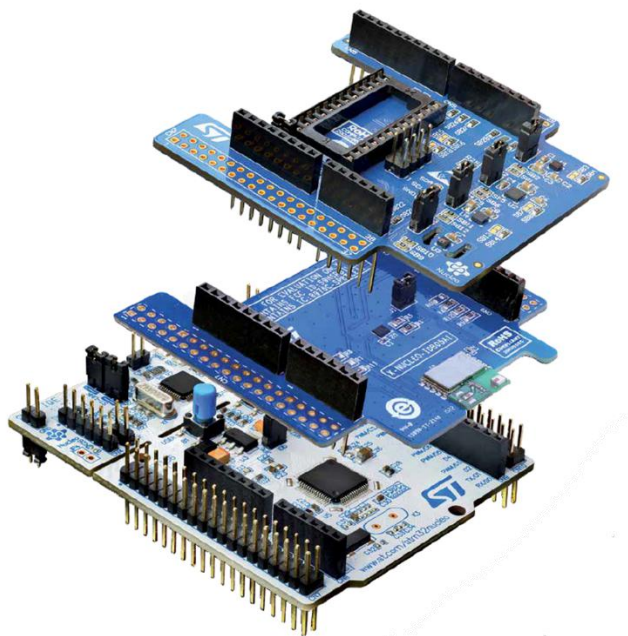


图 16. 下位机图样

3.1.1 JetRover 与 RK3588

目前市面上有很多 ROS 开发小车套件，如幻尔、冰达等等，由于实时性需要在较为复杂的使用场景下才可以体现，并且不同的应用场景下对于机器人的实时性任务的区分标准不一，因此为了可以满足更高效的测试需求，我们在项目中选用了 JetRover 机器人套件，如图 17。

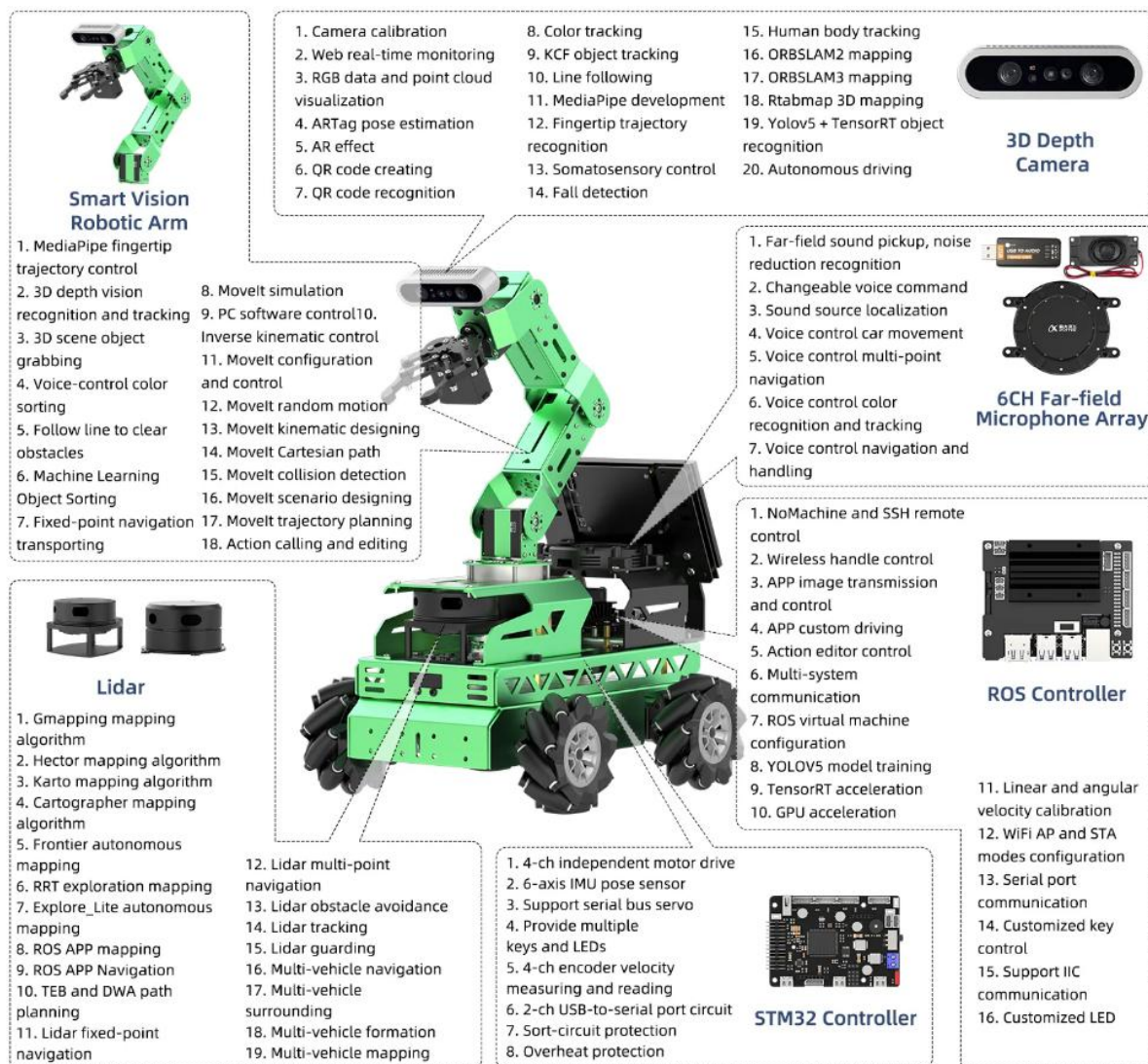


图 17. JetRover 硬件组件图

JetRover 是幻尔机器人的一款面向教育机器人场景的产品，它具有一个 6 自由度配有 3D 深度相机的智能视觉机械臂，一个 30m 内有效建图的激光雷达，一个 stm32 集成了驱动板的控制器和一块 Jetson Nano 开发板。

由于 Jetson Nano 已经是 2019 年推出的产品，目前在嵌入式开发板场景下已经不是一个高性能的解决方案，并且为了响应国产号召，我们在项目中选用了瑞芯微的 RK3588。RK3588（如图 18）是 Rockchip 公司推出的一款高性能系统级芯片（SoC），它采用 8 核架构，包括 4 个 Cortex-A76 大核和四个 Cortex-A55 小核，提供高效能和能效使用的平衡。同时它具备一个 Mali-G610 MP4 GPU 和一个 NPU，支持高性能图形处理，支持 8K 视频解码和 8K 显示输出，支持多路 4K 视频输入和输出，拥有高达

6 TOPS 的 AI 算力，并具备高达 32GB 的 LPDDR4X/LPDDR5 内存，并且支持 eMMC5.1 和 UFS 3.1 存储；且提供了丰富的 IO 接口，包括 USB3.1、PCIe3.0、SATA3.0 等，便于拓展和外设链接，适合一些嵌入式边缘计算、机器学习推理、图形处理等应用以及高端视频监控和多媒体处理设备需求。

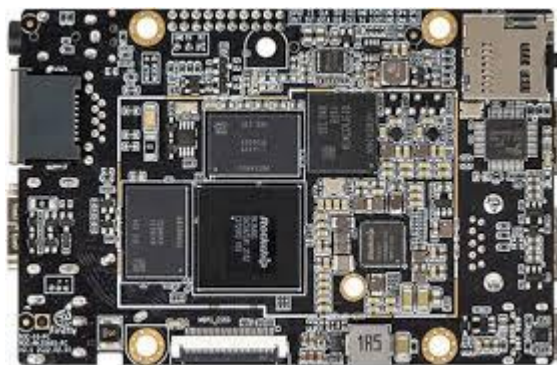


图 18. RK3588 开发板

3.1.2 电路与通信接口设计

JetRover 原有的供电电路是以 3 节 3.7V 的锂电池串联，电源直接接入下位机，而后再经由下位机的降压转换至 5V 后由 USB Type-C 通到 Jetson Nano 控制板（如图 19），所需电压为 5V，最大电流为 3A，最大功率为 15W。



图 19. JetRover 电路连接示意图

由于更改主控板后的 RK3588 所需的工作电压为 12V，最大电流为 3A，最大功率为 36W，在这样的工作功率下，并不适配于原有的 11.1V 供电模式，原有锂电池的放电功率最多只能达到 24W，而在电源无法保持稳压的场景下会直接导致 CPU 硬件计算错误，在运行高负载程序时会因硬件计算问题而导致很多的系统异常甚至自动关机。

为了能到正常的工作电压和工作电流，需要更换更高输出电压及更高放电电流的电源系统，同时为了方便日常开发，在本项目中为日常开发与脱机运行提供了两套电源方案。

- 方案一通过 3S LiPo 航模电池（12.6V）分两路，一路直接由 XT60 连接在降压模块上，经降压模块降压至 11.1V 后通过 DC 2.5mm 直接连接到下位机；另一路直接通过 DC 2.5mm 直接连接到 RK3588（如图 20）。

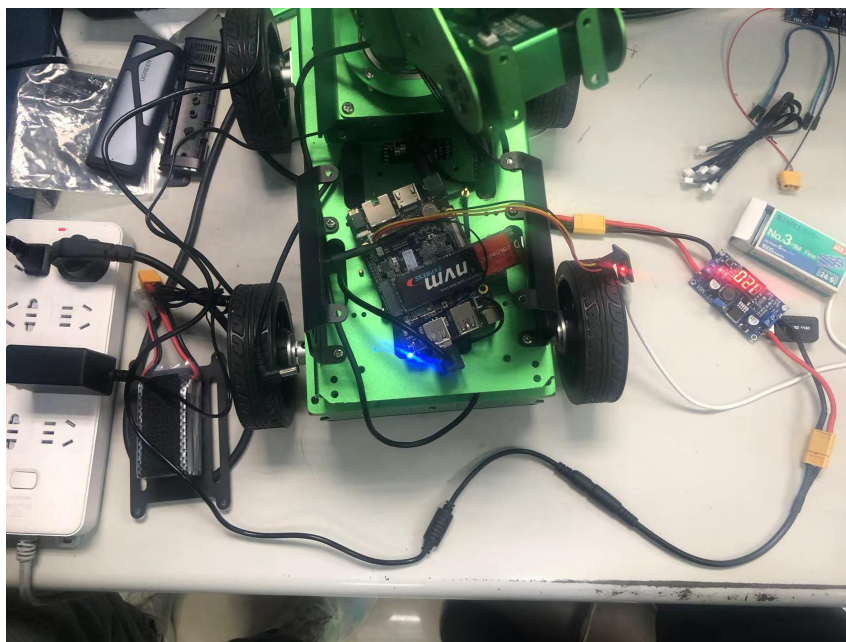


图 20. 方案一电路连接图

- 方案二直插电源对输出的 12V 5A 的 DC 分两路，一路直接由 XT60 连接在降压模块上，经降压模块降压至 11.1V 后通过 DC 2.5mm 直接连接到下位机；另一路直接通过 DC 2.5mm 直接连接到 RK3588（如图 21）。

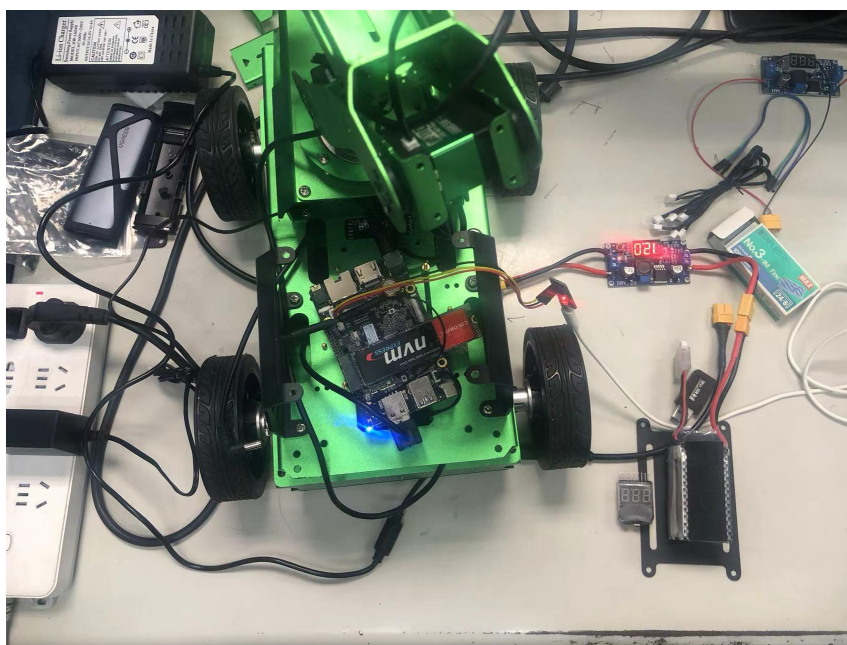


图 21. 方案二电路连接图

3.2 机器人软件系统

机器人软件系统环境如以下表 1 所示：



表 1. 机器人软件系统环境

条目	类型	版本
Ubuntu	操作系统	20.04
ROS	Linux 应用	Noetic
Linux 内核	操作系统内核	5.10.163
激光雷达驱动	Rplidar Driver	Rplidar-a3
深度相机驱动	Depth-Camera Driver	Astra-SDK
RVIZ	交互平台	RVIZ 1.14
Gazebo	仿真平台	Gazebo 11

3.2.1 Shyper + Linux + Unikernel 架构设计

机器人操作系统 ROS 是一个用于机器人软件开发的灵活框架，自 2007 年发布以来，特别是 ROS1，已经在 Linux 平台上建立了庞大的软件生态。这些软件生态包括了大量的开源包，涵盖了导航、控制、感知、仿真等各个方面，特别是导航包（navigation stack）、视觉处理包（vision packages）、移动机器人仿真包（gazebo）等，这些工具大大加速了机器人开发的进程，并已建立起了一个非常活跃的社区。而随着机器人应用的复杂性和对可靠性的要求不断提高，特别是在工业、医疗和自动驾驶等领域，对系统的实时性需求变得越来越重要。实时性要求系统能够在严格的时间限制内完成任务，以确保机器人操作的安全和精确。然而，ROS1 在标准 Linux 内核上的运行面临以下挑战：

- 非实时调度：标准的 Linux 内核使用的是公平调度器（CFS），它旨在确保系统中的每个任务都能公平地获得 CPU 时间片，这对于实时任务来说是不够的
- 中断处理延迟：Linux 内核并不是为实时中断处理设计的，在高负载或复杂系统中，可能会出现较长的中断延迟，这对于需要快速响应的实时系统是不可接受的
- 内核延迟：标准 Linux 内核的内核态和用户态之间的切换，以及内核自身的一些非实时操作（如内存管理、I/O 处理等）都会增加系统的延迟

目前已有的工作如 Preempt-RT 与 Xenomai3 等对 Linux Kernel Patch 的模式下所实现的实时性仍然是软实时，并不能达到硬实时的要求标准，对于一些超过规定时间无

法接受的任务在 Patch 模式下可以达到的效果有限。本文使用 Shyper + RTLinux + Unikernel 的方式从 Hypervisor 角度达到硬实时的目的，具体架构如图 22 所示。

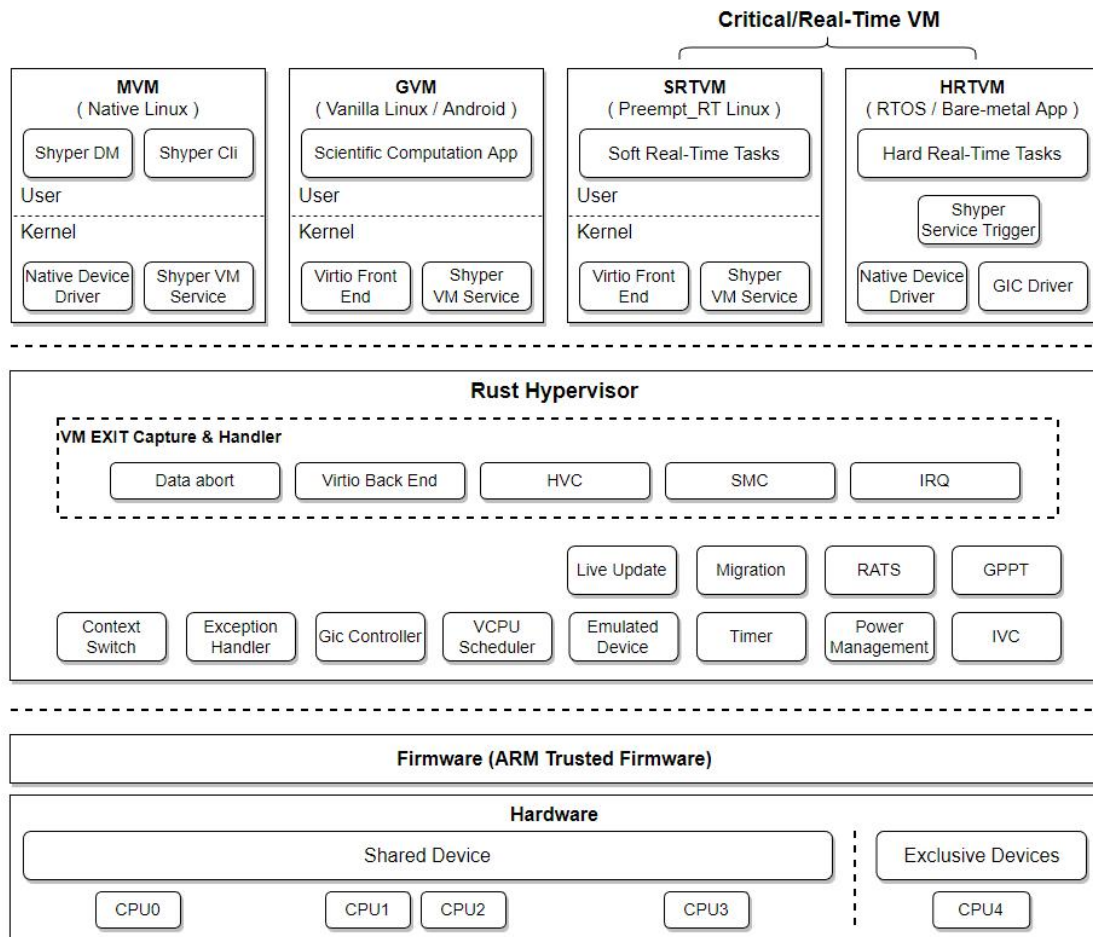


图 22. 混合关键系统架构图

该系统由一个 SRTVM 与一个 HRTVM 组成，二者以 ROS 话题名称为单位建立话题共享内存映射，并通过 IPI 发送核间中断与补充回调处理消息收发，并各自维护自身 VM 上所运行的 ROS 共享内存话题节点的拓扑信息。该系统的具体设计目标如下：

- 保证关键任务的实时性；
- 保证 Linux 原有任务及软件生态仍可以正常运行；
- 保证 Linux 与 RTOS 间在正常情况下相互隔离，并提供 Linux 与 RTOS 间专有的通信方式（共享内存）；
- 为 Linux 提供一个便捷的方式将实时程序转移到 RTOS 中运行；
- 基于 virtio/RPC 等方式为 RTOS 提供兼容 Linux 复杂的硬件接口和驱动支持；

为了满足上述设计需求，本文的混合关键系统实现了在 Shyper 上同时运行一个 MVM (Linux Manager VM) 和 HRTVM (Unikernel Hard Real-Time VM)，其中 MVM 用于管理 Hypervisor 提供的特权接口，并对 HRTVM 进行状态监控，同时可以利用自身的设备驱动，辅助 HRTVM 完成模拟设备的访问；HRTVM 用于提供硬实时支持，有可预测的实时性能，支持实时调度和抢占式调度。

为了保证 Linux 原有的任务及软件生态的正常运行，本项目实现了分析原有任务的资源配置情况，将相应资源均以直通设备、全模拟设备、半模拟设备的方式提供给 MVM，并合理的配置相应的中断转发以使原有软件生态得到良好支持。

为了保证 Linux 与 RTOS 间的正常情况下的相互隔离，且存在共享内存的通信机制，本文将 Linux 与 RTOS 所占用的 CPU、内存、设备等资源在创建时分隔开，并基于 MVM 与 HRTVM 在 Shyper 上的状态，通过 CLI 调用各自内核的 patch，而后调用 Hypercall，并最终在 Shyper 内实现共享内存池的映射。

为了 Linux 可以便捷地将程序转移到 Unikernel 中运行，需要尽可能的保证二者在编译器、链接库和其他工具等在两系统上尽量保持一致；为了保证 RTOS 可以得到 Linux 的丰富硬件接口和驱动支持，本项目利用了 Shyper 提供的 Virtio 设备和中介传递设备的方式，目前分别支持了 virtio-console、virtio-net 和中介磁盘，以使得项目得到正常运行。为了保证系统的实时性，本项目在 Shyper 内做中断分发时将实时中断、非实时中断、与两域内共同中断分隔开，优先处理实时中断。并通过 Shyper 提供的异步机制，降低 RTOS 的非 RT-Thread 陷入 EL2 的时长，通过 MVM 的计算资源，保证关键任务的执行效果。

3.2.2 实时性分析

3.2.2.1 虚拟化实时性所面临的挑战

虚拟化技术大大提高了嵌入式平台的资源利用率，但在资源复用的过程中，确保虚拟机的实时性能面临新的挑战。这些挑战主要包括：

CPU 虚拟化：为了让多个虚拟机共享同一物理核心，CPU 虚拟化通过抽象 CPU 寄存器等上下文信息，实现虚拟机的 vCPU 周期性调度。然而，对于实时性要求较高的

虚拟机来说，与其他虚拟机的 vCPU 一起参与调度会显著影响其计算任务的执行效率。即使采用抢占式的 vCPU 调度算法，也难以将调度延迟控制在实时虚拟机可接受的范围内。

模拟设备：模拟设备通过半虚拟化技术 Virtio，使虚拟机能够共享磁盘、网络、串口等资源。虚拟机在访问这些模拟设备时，需要通过 MMIO 操作陷入 Hypervisor 层级 (EL2)，在 Hypervisor 完成设备请求后，再返回虚拟机层级 (EL1) 继续执行任务。这一过程中，特权级切换的开销以及 Hypervisor 处理函数的计算成本会影响虚拟机任务的执行效率。

模拟中断控制器：模拟中断控制器允许多个虚拟机共享同一组中断控制器硬件。当中断事件发生时，CPU 会切换到 EL2 特权等级执行 Hypervisor 提供的中断处理函数，再将中断分发至相应的虚拟机。此外，虚拟机通过 MMIO 配置中断控制器状态的请求也需要由 Hypervisor 处理。这些虚拟化中断控制器引发的大量陷入事件，会对虚拟机的实时性能产生不利影响。

内存管理：传统面向云服务的虚拟机监视器通常采用动态内存分配策略。当虚拟机访问未映射的内存地址时，会触发 EL2 层级的缺页异常，由 Hypervisor 完成物理页面的分配与映射。当物理内存资源不足时，Hypervisor 需要使用复杂的页面置换机制解决缺页问题。这些内存分配和置换操作会引入大量陷入事件，严重影响虚拟机任务的实时性。此外，由于虚拟化层级的存在，缓存资源竞争更加激烈。虽然理论上可以通过缓存分区策略来提高缓存命中率，但这会影响 Hypervisor 的内存分配逻辑，增加大块连续内存分配的难度。

3.2.2.2 在虚拟化实时性的解决方案

在虚拟化环境中，虚拟化监视器 (hypervisor) 必须有效处理虚拟机发出的各种请求，包括设备请求、HVC、SMC 等。但当 CPU 从虚拟机切换到 Hypervisor 级别时，原本正在执行的任务会被暂停，这可能影响到关键任务的执行。为了确保虚拟机的实时性能，ACRN Hypervisor 设定了 VM-Exit-Less 指标，认为低频率的 VM-Exit 和短暂 VM-Exit 陷入时间是虚拟化实时性强的两个表现。Rust-Shyper 遵循了这一理念，通过实现 Rust 的异步任务模型和中断直通传输策略，以确保关键虚拟机的实时性能。

● 中介传递设备与 Rust 异步任务

传统同步模拟设备与中介传递设备模型的差异在于，中介传递设备模型能够将虚拟机的设备请求委托给非关键虚拟机所在核心代为处理。以模拟磁盘设备为例，传统同步模拟磁盘设备在虚拟机发起请求后，Hypervisor 的后端设备会截获磁盘请求并调用磁盘驱动程序进行数据读写。在请求完成后，CPU 会返回虚拟机层级继续执行原有任务。然而，这一过程中 CPU 长期处于 EL2，无法执行虚拟机相关任务，严重影响虚拟机的实时性能。而在中介传递设备模型下，虚拟机发起请求后，Hypervisor 仅需为当次请求建立相应的任务结构体，并向服务核心发送核间中断，然后即可回到 EL1 继续执行原有计算任务。相比之下，中介传递设备模型有效缩短了设备请求时虚拟机陷入的时长，符合 VM-Exit-Less 的目标。

为了实现中介传递设备模型，Rust-Shyper 利用了 Rust 的 `no_std` 异步语言特性，构建了全新的异步任务管理模块，其处理流程如图 23 所示。该模块能够将关键虚拟机的中介传递设备请求封装为 Rust 的 `async` 异步函数，并将其交由其他核心进行调度执行。为了简化 Rust-Shyper 的代码规模，我们重用了 MVM 的磁盘驱动程序，以协助其他虚拟机完成磁盘请求。这意味着，当 MVM 所在核心接收到来自其他虚拟机的中介传递设备请求时，Rust-Shyper 将通过中断注入的方式通知 MVM 新的设备请求信息，并由 MVM 用户态守护进程来执行相应的磁盘读写操作。这种做法有效减少了 CPU 在 Hypervisor 层级陷入的时间，降低了虚拟化对虚拟机实时任务的影响，从而提高了虚拟机的实时性能。同时，Rust 的异步任务模块优化了异步任务队列的管理逻辑。当任务队列为空时，虚拟机需要向 MVM 发送 IPI 通知新任务的到来。而当任务队列非空时，虚拟机只需将任务添加到队列中，等待 MVM 进行调度执行即可。

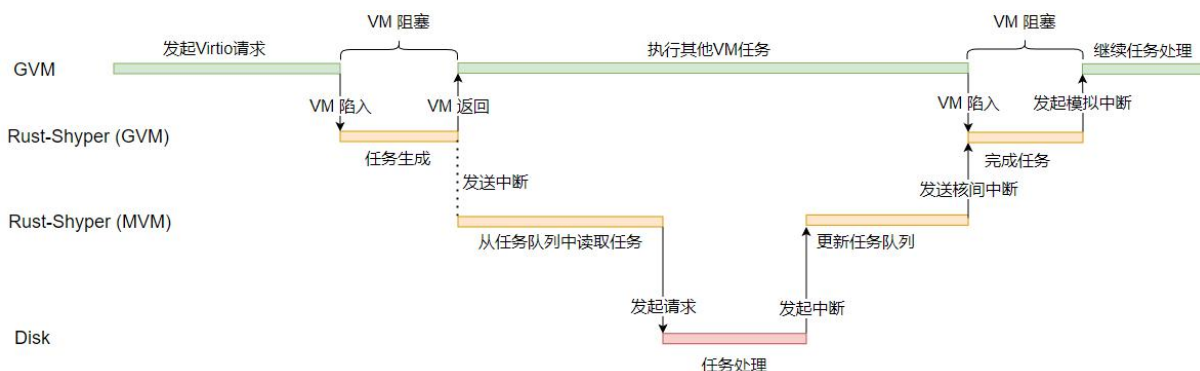


图 23. Shyper 异步任务执行协作图

在 Shyper 中异步任务的逻辑并没有具体实现在 VM 内，而是通过抢占 MVM 核心来完成阻塞任务，在异步任务的服务流程里，GVM 的异步请求是通过 `async` 修饰，存储在调度 `AsyncTask` 中。`AsyncTask` 本身是一个结构体，包含了异步任务的类型、任务来源、任务状态、任务执行函数等数据，而后任务被插入调度队列，等待 MVM 进行调度执行，完成一次异步任务创建。MVM 在调度队列中调度任务时，会根据 `AsyncTask` 中的数据调用 `Future` 结构体的 `poll` 函数，触发异步任务的执行流程。当异步任务完成时，MVM 会通过 IPI 的方式将任务结果发送给任务来源核心，并发送虚拟中断通知虚拟机当前任务已完成。

● 中断虚拟化与中断直通策略

中断虚拟化与中断直通策略是 Rust-Shyper 的一个重要特性。它支持 GICv2 中断控制器的直通和虚拟化策略，并且能够根据虚拟机类型进行灵活配置。

对于非实时虚拟机而言，中断控制器的虚拟化能够有效提升资源利用率，允许多个虚拟机共享同一个物理核心。Rust-Shyper 的虚拟中断控制器需要为虚拟机提供 GICC 和 GICD 两组寄存器的抽象。其中，GICC 的抽象由 ARMv8 体系结构提供的虚拟化扩展支持，而 GICD 的虚拟化则依赖于 Rust-Shyper 的纯软件实现。

尽管模拟中断控制器相对于硬件 GIC 而言具有更高的资源利用率，但硬件 GIC 的性能要优于软件实现的模拟中断控制器。为了提高中断处理的实时性，Rust-Shyper 选择为实时虚拟机提供 GIC 硬件的直通配置，从而避免了其他虚拟机与实时虚拟机共享同一中断控制器资源。

Rust-Shyper 的中断处理流程如图 24 所示。为了支持 EL2 异常等级对中断的拦截处理，Rust-Shyper 在初始化阶段需要为每个 CPU 设置 `HCR_EL2` 寄存器的 `IMO` 位。当中断发生时，由于 `HCR_EL2.IMO` 值为 1，该中断将被硬件引导至 EL2 提供的异常处理入口。

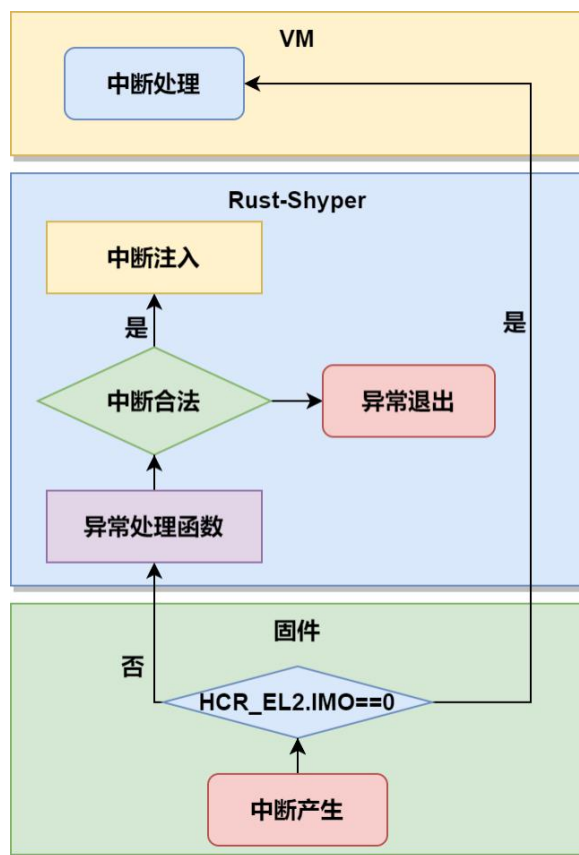


图 24. 中断直通示意图

对于实时虚拟机，Rust-Shyper 会将中断控制器直通给虚拟机。在这种情况下，中断不应被 Rust-Shyper 拦截处理，否则虚拟化层级的中断注入会对虚拟机的实时性能产生影响。因此，对于运行实时虚拟机的 CPU，Rust-Shyper 会将 HCR_EL2.IMO 位清零。这样，当核心运行在虚拟机层级时，中断将直接被虚拟机捕获和处理。

3.3 本章小结

本章主要讲了所用机器人的软硬件设计，并提出了预期的系统设计指标，然后为相应的设计指标简要说明了期望的解决方案。进一步引入了混合关键系统的实时性，列出了当前混合关键系统主要面临的实时性挑战，并以系统的角度对当前基于 Hypervisor + RTLinux + RTOS 的方案实现了对应的解决方案。

第四章 虚拟机间的共享内存与消息传递

虚拟机间通信是指同一物理主机或不同物理主机上的虚拟机之间进行数据交换和信息传递的过程。目前虚拟机通信主要有虚拟网络、共享内存、消息传递和分布式服务等方式，其主要目的是实现虚拟机之间的协作与资源共享，提高系统的灵活性和效率。

在本项目中的嵌入式场景下通常考虑的只有同一物理主机上的多个虚拟机间的通信，在这种通信场景下，基于共享内存的方式实现的通信为当下情境高效通信的不二之选。在 Hypervisor 的共享内存的场景下，两阶段地址翻译实现的共享内存需要不同 VM 各自向下传入自身需要映射的 ipa，再由虚拟化监视器将各个 VM 的 ipa 映射到同一块 pa 上。进一步地，再有各自 VM 的内核将这一部分内存映射到用户态，后续便可以在用户态直接操作这一片内存区域，在此基础上，只需要定义一套 VM 间对这片内存区域的访问规则即可实现用户态下的共享内存通信。在这个模式下的通信实际在 VM 间建立了共享内存后便无需陷入 Hypervisor，相比由 Hypervisor 控制的 VM 间通信极大地减小了 VM-Exit 开销。

4.1 通信模型

Shyper 下基于共享内存的通信模型总体架构如图 25 所示。

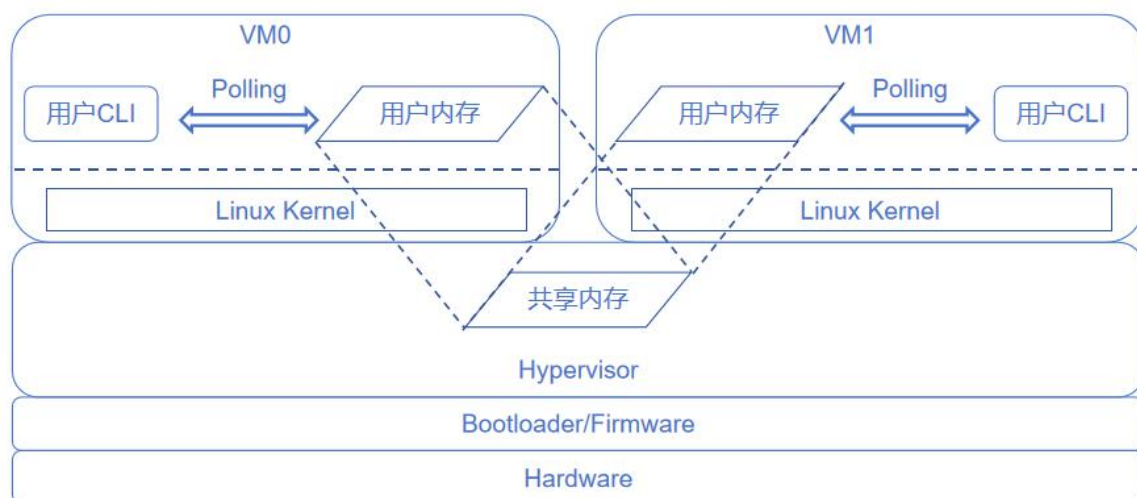


图 25. Shyper 共享内存的通信模型

在实际执行流程中可以分为 Linux 与 RTOS 的用户态与 CLI、Linux 与 RTOS 的内核态与 Hypervisor。Application 通过类 Posix 规范的接口方式直接发送系统调用，或进入 CLI，通过 CLI 的函数触发特定的系统调用。而后在内核处完成该 VM 的共享内存映射的维护，并发送 HVC 进入 Shyper 中完成物理内存的映射。

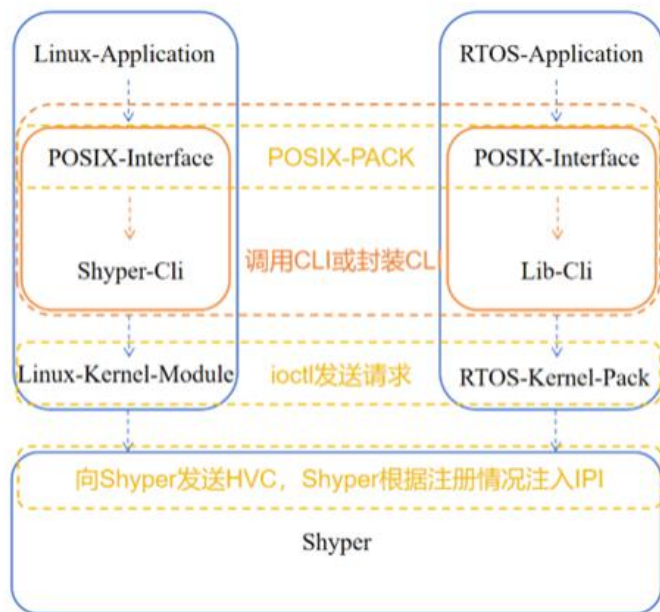


图 26. 本项目中的类 POSIX Shared Memory 在每个系统组件中的存在状态

目前在跨 VM 场景下并没有一个较为公认的虚拟机间通信规范，在本文中采取类 POSIX 的接口方案实现虚拟机间通信。POSIX 可移植操作系统接口（Portable Operating System Interface of UNIX，缩写为 POSIX）实际是 IEEE 为要在各种 UNIX 操作系统上运行的软件而定义的一系列 API 标准的总称。POSIX 对于共享内存的定义及使用方法主要包括

- 创建或打开共享内存对象 `int shm_open(const char *name, int oflag, mode_t mode)`，使用 `shm_open` 函数创建一个新的共享内存对象或打开一个已经存在的对象，其中 `name` 代表共享内存的名称，必须以斜杠/开头，例如 `"/my_shm"`，`oflag` 代表打开标志，常用的有 `O_CREAT`（创建）和 `O_RDWR`（读写），`mode` 代表权限模式（如 `0644`），仅在创建新共享内存对象时使用。
- 设置共享内存对象的大小 `int ftruncate(u64 fd, off_t length)`，`ftruncate` 函数用于调整

共享内存对象的大小，`fd` 代表共享内存对象的文件描述符，`length` 表示要设置的共享内存大小（以字节为单位）。

- 映射共享内存对象到进程的地址空间 `void *mmap(void *addr, size_t length, int prot, int flags, u64 fd, off_t offset)`，`mmap` 函数用于将共享内存对象映射到调用进程的地址空间，其中 `addr` 表示映射的起始地址，一般会选择设为 `NULL`，这样会由系统决定从何处开始映射，`length` 表示映射区域的长度，`prot` 是内存保护标志，如 `PROT_READ`（可读）和 `PROT_WRITE`（可写），`flags` 为映射标志，在共享内存的情境下通常会设置为 `MAP_SHARED`（共享映射），`fd` 是共享内存对象的文件描述符，`offset` 代表映射的偏移量，在直接通过指针或者文件描述符访问时一般设为 0 即可。
- 对共享内存进行读写操作：映射之后，可以像操作普通内存一样对共享内存进行读写。
- 解除映射 `int munmap(void *addr, size_t length)`，`munmap` 函数用于解除内存映射，`addr` 表示映射区域的起始地址，`length` 表示映射区域的长度。
- 删除共享内存对象，`int shm_unlink(const char *name)`，`shm_unlink` 函数用于删除共享内存对象，`name` 代表共享内存的名称，同样必须以斜杠/开头。

4.2 Shyper CLI、Linux Kernel Module

在 Shyper 中对 share memory 的实现基本遵循 Linux 的 `posix` 方式，在用户态下通过 `include` 自定义的 `posix_shmem.h` 来使用 Shyper 提供的跨 VM 共享内存函数。在内核模块与 CLI 中定义了一套与内核模块的交互规范，包括通过 `shyper_dev` 结构体的交互规范、`read/ioctl` 函数的重映射以及与 Shyper 交互的内核服务等。

对于 `shyper_dev` 结构体

- `struct cdev cdev`: 这是一个字符设备结构体，用于表示字符设备。`cdev` 结构体通常包含与设备文件操作相关的信息，如文件操作函数指针等。
- `*char cfg_queue`: 指向一个配置队列的指针，用于存储设备的配置信息或指令。
- `*char send_queue`: 指向一个发送队列的指针，用于存储待发送的数据或指令。

- u64 receive_queue[8]: 一个接收队列数组, 包含 8 个 64 位无符号整数, 用于存储接收到的数据或指令。
- u32 receive_idx[4]: 一个接收索引数组, 包含 4 个 32 位无符号整数, 用于记录接收队列的索引或状态。
- u64 send_idx[4]: 一个发送索引数组, 包含 4 个 64 位无符号整数, 用于记录发送队列的索引或状态。
- u64 usr_pid: 用于存储用户进程的 PID (进程标识符), 标识与该设备相关联的用户进程。
- u64 vmid: 用于存储虚拟机的 ID, 用于虚拟化环境中标识不同的虚拟机。
- *struct task_struct current_task: 指向当前任务 (或进程) 的指针。task_struct 是内核中的一个结构体, 包含了关于进程的所有信息。
- *char cfg_ptr: 指向配置指针的指针, 用于指向具体的配置信息或内存地址。
- *struct ring_queue usr_arg_queue: 指向一个环形队列的指针, 环形队列是一种常用的数据结构, 支持高效的生产者-消费者模型, 用于与守护进程共享数据, 内核模块每次给守护进程发送 signal 会将数据放到环形队列等待守护进程的主动读取。

同时为了实现用户态对内存的映射访问, 需要将 Linux 进程的地址空间经由内核模块进行申请后重映射, 需要定义一个重映射的 file_operations, 并定义一个 miscdevice, 用于完成共享内存部分的 mmap 请求。

```
static struct file_operations shmем_remap_pfn_fops = {  
    .owner = THIS_MODULE,  
    .mmap = remap_pfn_mmap_shmem,  
};
```

file_operations 结构体用于定义与设备文件相关的操作函数。具体字段说明如下:

.owner = THIS_MODULE: 表示该文件操作结构体的所属模块。THIS_MODULE 是一个宏, 指向当前加载的内核模块。这有助于增加模块的引用计数, 防止模块在使用过程中被卸载。

.mmap = remap_pfn_mmap_shmem 指向一个函数 remap_pfn_mmap_shmem, 该函数

实现了 `mmap` 系统调用。`mmap` 用于将设备内存映射到用户空间，使用户进程可以直接访问设备内存。

```
static struct miscdevice shmем_remap_pfn_misc = {  
    .minor = MISC_DYNAMIC_MINOR,  
    .name = "posix_shmem",  
    .fops = &shmем_remap_pfn_fops,  
};
```

`miscdevice` 结构体用于定义一个杂项设备 (misc device)。具体字段说明如下：

`.minor = MISC_DYNAMIC_MINOR` 设备的次设备号。`MISC_DYNAMIC_MINOR` 是一个宏，用于动态分配次设备号，而不是使用固定的次设备号。这样可以避免设备号冲突。

`.name = "posix_shmem"` 设备的名称。这里命名为 "posix_shmem"，这意味着这个设备将在 `/dev` 目录下创建一个名为 `posix_shmem` 的设备文件。

`.fops = &shmем_remap_pfn_fops` 指向 `file_operations` 结构体的指针。该指针告诉内核使用 `shmем_remap_pfn_fops` 结构体中定义的文件操作函数来处理设备文件的操作。

基于文件操作的重映射方式，`shyper` 内核模块为用户态提供了对 `/dev/shyper` 的 `open` 和 `read` 的重映射以及 `/dev/posix_shmem` 设备的 `mmap` 重映射的访问方式。实际 `shyper` 的内核模块再注入后会同时创建 `/dev/shyper` 设备与 `/dev/posix_shmem` 设备，通过 `read` 操作 `/dev/shyper` 设备会触发内核模块中书写的 `shyper_read` 函数，在这种情况下，内核模块的 `shyper_read` 函数会将环形队列 `usr_arg_queue` 的队首节点对应的数据拷贝到用户空间，在用户态可以通过这种方式获得内核空间的数据。同时通过 `ioctl` 函数操作 `/dev/shyper` 设备时会触发内核模块中定义的 `shyper_ioctl` 函数，在 `shyper_ioctl` 中提供了非常多的 `handler` 函数，在用户态可以通过 `ioctl` 向用户态发送一个结构体，内核模块通过预定义操作结构体和 `ioctl` 实现预定义好的内核操作以及发出 `HVC` 等。在共享内存实现下，可以通过 `mmap` 操作 `/dev/posix_shmem` 设备来完成共享内存的映射，内核会在用户态调用 `mmap` 的时候触发 `remap_pfn_mmap_shmem`，并获取用户态陷入时的进程状态以及 `vma` 结构体，可以通过 `shyper_ioctl` 协同的方式，在用户态下协同



调用使得内核模块可以获得用户态的特定映射需求。

在此基础上已经可以使用共享内存使跨 VM 的数据可见，但基于该共享内存通信需要补充同步机制，同时为了考虑 VM-Exit Less，本项目在用户态对于该共享内存实现了一个预定义的访问消息队列结构：

```
struct MessageQueue {  
    int read_ptr;  
    struct ShmMessage {  
        int write_ptr;  
        spinlock_t mut;  
        struct Message {  
            int type;  
            char mesg[256];  
        } msg[64];  
    } *shm_msg;  
} mq;
```

在这一模式下将该内存视为 shm_msg 的起始地址，在用户态调用 shyper_mmap 的会在该消息队列中建立 shm_msg 与该话题名的 share_memory 映射，并将自己的 read_ptr 初始化为当前的 write_ptr 并陷入等待，轮询直到 write_ptr 大于 read_ptr 的时候，将对应位置 shm_msg -> msg[mq.read_ptr + 1] 的值取出，而后取出 mesg 字段并将 read_ptr 自加即完成一次消息收取。在多个写进程的情况下，多个写进程需要先轮询获取 shm_msg 处的自旋锁，而后才可以拿到对应位置的写权限，在完成对应位置的消息写后，更新 write_ptr 后再将自旋锁释放，示意图如图 27。

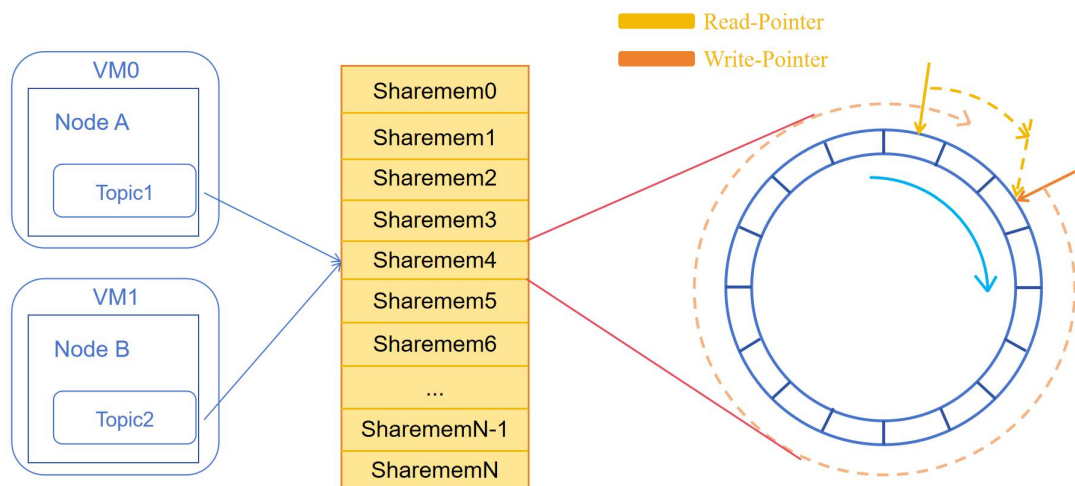


图 27. 基于共享内存的环形消息队列实现

4.3 Shyper 内存映射

Shyper 在虚拟化场景下对地址翻译是一个二阶段的映射，每个客户机的地址会在各自内核视角下为一个 ipa，再由 Shyper 将相应的 ipa 映射到实际的 pa。在硬件虚拟化的支持场景下，VM 可以通过拓展页表以及 VTTBR_EL2 寄存器放置二级页表的基地址指针后硬件可以直接地完成页表的装填。

在虚拟化的二级页表映射下，模拟设备和直通设备的 MMIO 都会直接映射到实际的物理地址中，而其他如 Shyper 的内存、CPU 的私有数据等则会在 VM 可操控的地址后加一层内存屏障后直接保存在相应的物理地址中。对于共享内存部分如图 28，是在 VM 中的视角下会是各自视角下的一片中间物理内存区域，而实际则是由 Shyper 将其映射到同一块物理内存上。

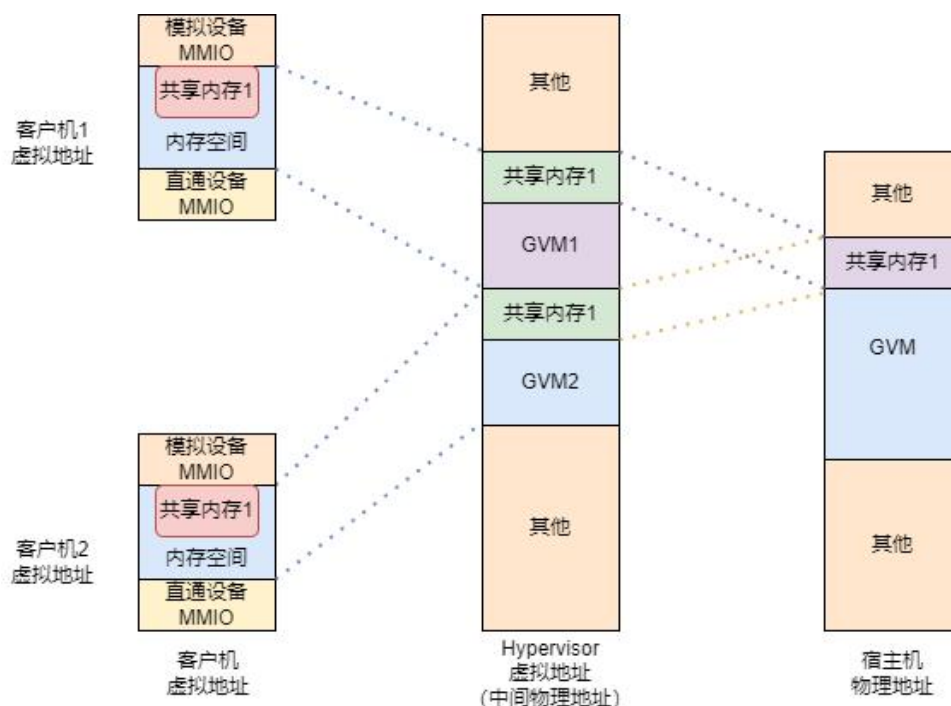


图 28. Shyper 对多 VM 内的共享内存的映射方式

Shyper 会在地址空间内预分配一块内存，在 Shyper 内的内存通过 PageFrame 管理和申请。在 Shyper 内，对于共享内存可以由 Shyper 内通过自建的结构体管理共享内存，并通过 PageFrame 去申请页面大小和控制页面映射。具体可定义：

```
pub static SHARE_MEM_LIST: Mutex<BTreeMap<String, ShareMem>> =
Mutex::new(BTreeMap::new());
```

SHARE_MEM_LIST 用于保存整个 Share Memory 列表，用于管理 Share Memory Name 与 ShareMem 的映射

```
pub struct VmStore {
    pub vm_id: usize, // 请求共享内存的 vm_id
    pub ipa_start: usize, // 请求的共享内存的 ipa 基地址
    pub size: usize, // 请求的共享内存大小
}
```

VmStore 代表共享内存的 VM 结构体，保存上层的单个 VM 用于共享内存建立时的请求信息，包括需要映射到该 VM 的 vm_id，请求共享内存的 VM 对该片共享内存所需要的大小和该 VM 需要映射到的 ipa 地址

```
pub struct ShareMem {  
    pub vm_list: BTreeMap<usize, VmStore>, // 共享内存的 vm 列表  
    pub mem: Arc<PageFrame>, // 共享内存的真实物理页面  
    pub size: usize, // 真正的共享内存的大小  
}
```

ShareMem 代表共享内存存在 Shyper 中的结构，用于实际代表每块 ShareMem Name 下的共享内存映射情况，包括映射到该片共享内存的 vm_list，该共享内存映射到的真实物理页面，以及共享内存的实际所需大小。

4.4 建立共享内存和通信过程

4.4.1 系统启动阶段

在 Shyper MVM 启动后，先在 MVM 上执行 “insmod shyper.ko” 注入内核模块，在这一阶段，主要完成的是内核初始化工作，包括通过在 Shyper 代码内预定义好的 MVM 设备树文件约定的中断号调用 get_irqno_from_node 获取该中断号（0x20），并通过 request_irq 注册该中断；调用 misc_register，注册 shyper 设备和 posix_shmem 设备。

在 CLI 中用户可以通过使用 shyper 可执行文件执行 “shyper system daemon medicate-config.json &” 启动用户态的守护进程，守护进程启动过程中会根据 medicate-config.json 文件中的信息创建中介磁盘，同时会注册 sig_handler_event 函数获取来自内核态的中断信号，并调用 ioctl 函数告知内核模块 daemon 守护进程的 ID 信息。守护进程在初始化完成后会进入 while 循环，完成用户指令的处并接收来自内核模块的中断。

而后通过 CLI 执行 “shyper vm config vm1-config.json” 配置 VM1 的启动参数、内核镜像、内存地址、cpu 数、模拟设备、直通设备以及 dtb 等信息，这时就会在 shyper 内定义这一个 vm，并可以通过 “shyper vm list” 即可查询到 vm 的注册信息。

在这一个状态下，通过另一个连接终端（如 ssh）执行 “minicom -D /dev/hvc0 115200”，打开一个调试 virtio-console 串口监视 VM1 的输出，而后即可在 MVM 执



行 “shyper vm boot 1” 启动 VM1，随后即可在 VM1 上看到启动输出。

4.4.2 共享内存建立过程

这里以 VM0 与 VM1 通过同一个 Share Memory Name 建立起一片共享内存映射为例，简要介绍在整个共享内存建立时的执行过程。

● VM0

1. 通过 `shm_fd = shyper_shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666)`; 打开一个名为宏定义的 `SHM_NAME` 的共享内存文件，并设置为创建，且可读写，并将该共享内存的权限置为 0666，其他映射到该内存的也可以进行读写。如果打开成功，会为 `shm_fd` 返回一个文件描述符，如果不成功返回的文件描述符为 -1。在 User Lib 下调用 `shyper_shm_open` 后会关联打开 `shyper` 设备，通过 `ioctl` 发送 0x1302 系统调用，表示执行 `SHM_OPEN` 的系统调用，在 `SHM_OPEN` 中内核模块会先通过 `copy_from_user` 拷贝用户态传来的 `shm_open` 结构体参数，并 `copy shm_open` 结构体参数中的 `name_buf` 到内核态的共享内存结构体中，然后执行 `kshm_open`，会添加一个 `struct misc_shmem` 项到 `shmem_pool` 中，并将 `shm_fd` 返回，再由内核模块将 `shm_fd` 赋值后使用 `copy_to_user` 拷贝回用户态。
2. 通过 `shyper_ftruncate(shm_fd, SHM_SIZE)` 设定共享内存大小，`shm_fd` 代表通过第一步获得的文件描述符，在内核中的 `shmem_pool` 中会维护记录相应的 `shm_fd`，在建立后的查询均通过 `shm_fd`。在执行 `shyper_ftruncate` 后同样会关联打开 `shyper` 设备发送 0x1303 系统调用，在进入内核后同样 `copy_from_user` 得到调用的结构体参数后，通过 `shm_fd` 检索 `shmem_pool` 并将对应的 `shmem_fd` 处的 `size` 更新为参数输入的 `SHM_SIZE`，并将检索/更新的状态返回，-1 代表失败，0 代表成功。
3. 通过 `shyper_mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0)`; 由系统决定一个位置为 `shm_fd` 文件描述符对应的共享内存映射到一个位置，并将映射到的位置返回，同时设置为这段内存对自

身是可读写，且是 0 偏移的共享映射。在这里的 User Lib 中会关联打开 shyper 与 posix_shmem，先通过向 shyper 发送 0x1304 系统调用传入 shmem_fd 标识，再通过 Linux 内置的 `mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, memfd, ipa_offset)`；映射打开的 `/dev/posix_shmem` 的 fd，在 Linux 下通过 `mmap` 对设备的映射会对应到在内核模块中定义的 `file_operations` 处的 `mmap` 重映射函数，在本项目中使用的是 `remap_pfn_mmap_shmem`，在 `remap_pfn_mmap_shmem` 中会根据 `mmap` 重映射得到的 `file` 与 `vma` 获得映射的 `start` 与 `end` 地址并计算映射长度，然后通过 `hvc_call(kva2ipa(shmem->name), shmem->size, cur_ipa, 0, 0, 0, 0, HVC_MODE(HVC_SHMEM, HVC_SHMEM_INIT))`；陷入到 Shyper 中。

4. 在 Shyper 中会通过 `shmem name` 的 `ipa` 来 `copy` 字符串，并通过 `size` 和 `ipa` 来使用 `pt_map_range` 建立映射关系。在 VM 上传来一个映射请求的时候，会先获取当前 VM 的编号，然后逐个字节获取 VM 通过指针传来的 `shmem name`。进一步查询 `SHARE_MEM_LIST`，看是否已经有同名的共享内存映射，如果有则直接建立映射关系，创建当前 VM 的 `VMStore` 结构体，并将该结构体放入到 `ShareMem` 的 `vm_list` 内，表示当前的 `ShareMem` 将该 VM 的以该 `ipa` 为起始大小为 `size` 的内存添加到了该 `ShareMem` 的映射上，进一步的如果新添加的 `size` 大于已经建立映射时设置的 `size`，会先申请一块新的足够大的连续内存，然后将原有 `ShareMem` 中的内容 `memcpy` 到新的内存中，然后把原内存释放；如果没找到同名的映射，则新建一个 `ShareMem`，进行页面分配，并将该 `ShareMem` 加入到 `SHARE_MEM_LIST` 中，然后将该 `ShareMem` 与 VM 的 `ipa` 通过 `pt_map_range` 建立映射关系。
5. 在 Shyper 建立好映射返回后，再由 `remap_pfn_mmap_shmem` 继续执行获取当前映射的物理页框号，然后设置新的页偏移，并通过 Linux 内置的 `remap_pfn_range(vma, start, pfn, size, vma->vm_page_prot)` 将该片内存与用户进程的内存建立映射关系，而后简单计算下一段共享内存的内核地址偏移，并恢复内核变量的标识位后将映射得到的指针返回。



6. 这时在用户态即可以输出返回的指针了，并可以简单地通过 `sprintf((char*)ptr, "Hello, shared memory!");`来向该共享内存地址写入字符串。
7. 在使用完成后可以通过用户态调用 `shyper_munmap(ptr, SHM_SIZE)`，将对应指针位置相应大小的内存取消映射，在 Linux 下 `shyper_munmap` 会关联 `shyper` 设备的打开操作，而后 `ioctl` 发送 `0x1305` 的系统调用，在进入内核后会 `copy` 用户态下传来的内核操作结构体，而后调用 `kshm_munmap` 减少一次在内核结构体中保存的该共享内存的引用，当该共享内存的引用减少到 0 的时候，该共享内存所对应的内存便会在内核中释放掉。

● VM1

1. 在 VM1 上仍然需要通过 `shm_fd = shyper_shm_open(SHM_NAME, O_RDWR, 0);`打开一个名为宏定义的 `SHM_NAME` 的共享内存文件，但此时已经无需设置创建与权限位（这种写法不会导致错误，但不符合 POSIX 规范），设置权限为可读写。同样如果打开成功，会为 `shm_fd` 返回一个文件描述符，如果不成功返回的文件描述符为 -1。在 User Lib 下调用 `shyper_shm_open` 后会关联 `shyper` 设备，通过 `struct device` 绑定“`shyper`”设备，而后通过驱动中定义的 `shyper` 设备访问规范，直接通过 `kshm_open(SHM_NAME, O_RDWR, 0, &shm_fd);`进行系统调用，在 `kshm_open` 执行时会添加一个 `struct misc_shmem` 项到 `shmem_pool` 中，并将 `shm_fd` 赋值给传入的 `shm_fd` 指针。
2. 通过 `shyper_ftruncate(shm_fd, SHM_SIZE)` 设定共享内存大小，`shm_fd` 代表通过第一步获得的文件描述符，同样在内核中的 `shmem_pool` 中会维护记录相应的 `shm_fd`，在建立后的查询均通过 `shm_fd`。在执行 `shyper_ftruncate` 后同样会关联绑定到 `shyper` 设备，并通过 `kshm_ftruncate` 触发系统调用，在进入内核后通过 `shm_fd` 检索 `shmem_pool` 并将对应的 `shmem_fd` 处的 `size` 更新为参数输入的 `SHM_SIZE`，并将检索/更新的状态返回，-1 代表失败，0 代表成功。
3. 在 Unikernel 仍可通过 `shyper_mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);`由系统决定一个位置为 `shm_fd` 文件描述符对应的共享内存映射到一个位置，并将映射到的位置返回，同时设置



为这段内存对自身是可读写，且是 0 偏移的共享映射。在这里的 User Lib 中会关联打开 shyper 与 posix_shmem，先通过 shyper 进入 shm_open 系统调用传入 shmem_fd 标识，然后通过 posix_shmem 进入 kshm_mmap 系统调用，进一步即可直接通过 hvc_call(kva2ipa(shmem->name), shmem->size, cur_ipa, 0, 0, 0, 0, HVC_MODE(HVC_SHMEM, HVC_SHMEM_INIT));陷入到 Shyper 中。

4. 在 Shyper 内两 VM 的执行逻辑相同，映射完成后会将返回值在系统调用过程中通过以指针赋值的方式传递到用户态。
5. 这时便可以在用户态下通过返回的指针访问该共享内存的内容了 printf("mmap success: %s\n", (char*)ptr)。
6. 在使用完成后可以通过用户态调用 shyper_munmap(ptr, SHM_SIZE)，将对应指针位置相应大小的内存取消映射，在 Unikernel 中 shyper_munmap 也会关联 shyper 设备的绑定操作，而后通过 kshm_munmap 减少一次在内核结构体中保存的该共享内存的引用，当该共享内存的引用减少到 0 的时候，该共享内存所对应的内存便会在内核中释放掉。

对于删除部分，由于跨 VM 间的引用并没有一个较为成熟的规范，Linux 上是通过进程的 PID 来记录引用的方式来控制共享内存的映射及取消映射的，但在 Hypervisor 的视角下的 PID 并无实际意义，而且共享内存的直接写入本身面向的需求场景也更关注高性能和实时性，对安全性应交给用户来保证，因此在本项目中约定 shmem_unlink 由用户保证在何时调用，比如由创建者决定何时删除等，在调用一次 shmem_unlink 之后相应 SHM_NAME 标识的 share memory 便会在内核和实际的内存中并释放掉。

● VM0

1. 在调用 shyper_unlink(SHM_NAME)后，会关联进入到 shyper 设备中，在 unlink 时会直接释放内核中对该 SHM_NAME 对应的共享内存结构，并调用 hvc_call(kva2ipa(arg->name), 0, 0, 0, 0, 0, 0, HVC_MODE(HVC_SHMEM, HVC_SHMEM_DELETE))向 Shyper 发送 HVC。
2. 进入 Shyper 后，会先获取对应 VM，然后从对应 VM 处获取 SHM_NAME，并查询是否存在同名的共享内存，如果存在则接触映射，释放这一段内存，并



删除在 SHMEM_MEM_LIST 中保存的结构，然后 FLASH TLB；如果没有同名的共享内存则返回错误。

4.6 本章小结

本章较为详细地介绍了本项目基于 POSIX 规范创建的一套创建跨 VM 的共享内存的方式，并结合 Linux 与 Unikernel 的 CLI 与 Kernel Module 中的创建和建立映射关系详细叙述了在内核和用户态维护的整个共享内存数据结构，并据此讲了 Shyper 最终对于两 VM 共享内存的映射和基于该共享内存下的消息队列，以及该消息队列在跨 VM 时的访问方式，最后以整个通信流程为例详细说明了跨 VM 在执行通信时的调用和执行方案。

第五章 ROS 通信拓展

5.1 ROS 通信拓展

5.1.1 ROS 原生通信机制

ROS 中为了满足所有通信场景的需求，提供了几种节点间的通讯方式，包括话题（Topic）、服务（Service）和动作（Action），如图 29 所示。其中话题和服务是两种最基础的通信方式，分别对应消息收发过程的异步和同步方式，Action 可以理解为是对 Service 方式的进一步封装。在本项目中，为了验证降低通信延迟的效果，选用话题和服务这两种基础通信方式作为实现的主要目标和通信测试的 baseline。

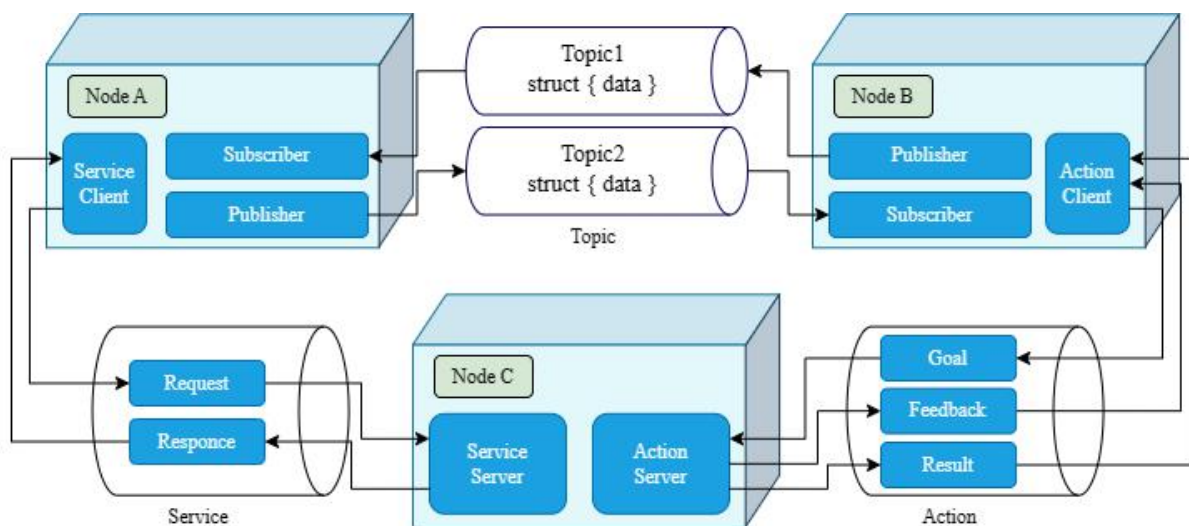


图 29. ROS 内存在的通信方式

话题通信是一种多对多的异步的进程间通信方式。在话题通信中，主要有发布者（Publisher）、接收者（Subscriber）和话题（Topic）三个概念。发布者向指定话题发布消息，订阅者订阅话题并接收发布到话题上的所有消息。一个话题同时可以有 0 到多个发布者和 0 到多个接收者。话题通信的运作方式如图 30 所示。ROS 利用 Master 节点和网络通信维护话题通信。在某一话题上新增发布者或订阅者时，ROS 首先利用 XML/RPC 更新该话题上的节点信息，再利用 TCP 建立发布者与订阅者之间的连接，之后发布者就可以通过 TCP 向订阅者发布消息了，此过程如图 31 所示。

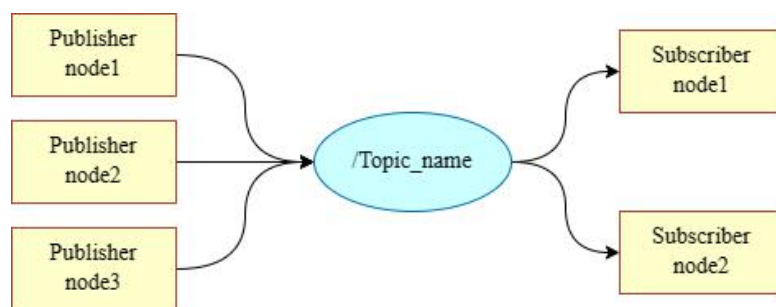


图 30. 话题通信的广播方式

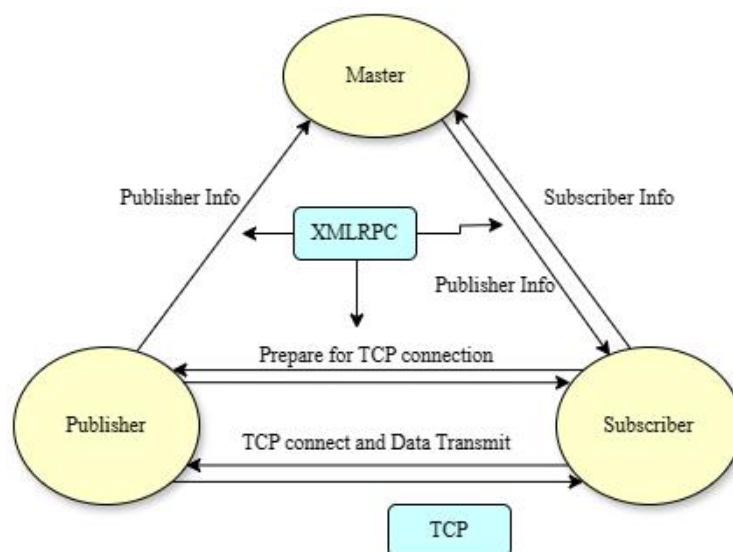


图 31. ROS 话题更新方式在 TCP 上的实现

利用网络实现的话题通信机制存在两点性能上的弊端。（1）重复的数据拷贝。在原有的实现方式下，发布者在发布一条消息时，需通过 TCP 向每个该话题的订阅者发送一次该消息。若该话题上有较多订阅者，就需要将同样的数据拷贝并发布多次，增大通信延迟。（2）利用网络通信造成的时间延迟。很多情况下，多个 ROS 节点运行在同一主机上，这些 ROS 节点若通过网络通信，操作系统网络协议栈会先将数据处理成网络包，再将网络包发送给一个虚拟的网络接口即环回接口，到达环回接口的网络包会被网络协议栈传递给目标 ROS 节点。不难看出，在同一主机上的 ROS 节点利用网络进行通信是不够直接的，存在多余操作，增大通信延迟。

服务通信是一种多对一的同步的进程间通信方式。在服务通信中，主要有服务节点 (Server)、客户节点 (Client) 和服务 (Service) 三个概念。一个服务节点提供一种服务，可以有多个客户节点向这个服务发出请求，服务节点接收请求并做出回复，客户节点在发出请求后阻塞等待回复。服务通信的运作方式如图 32 所示。类似于话题通

信的实现方式，原生的 ROS 服务通信也是利用网络通信进行信息协商和数据传递，如图 33 所示。

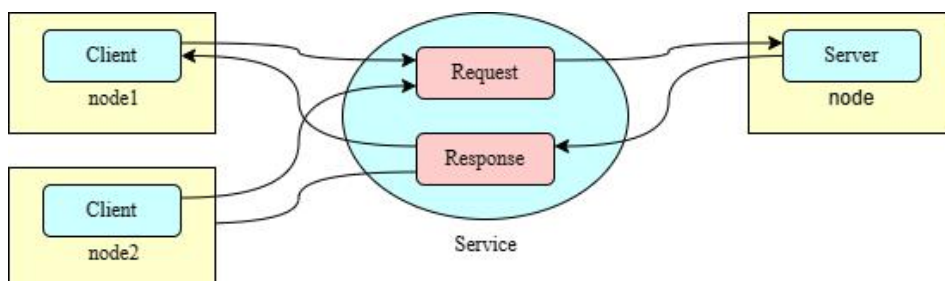


图 32. ROS 服务的通信机制

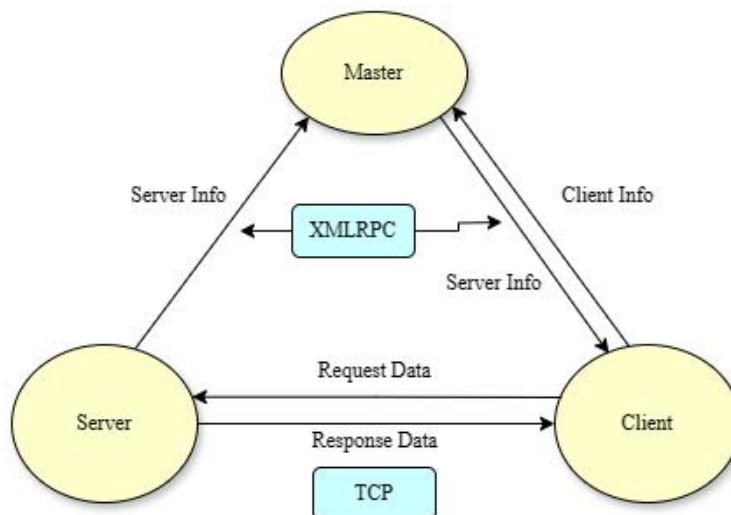


图 33. ROS 服务的更新方式在 TCP 上的实现

虽然服务通信是多对一通信，不存在类似话题通信中的重复拷贝相同数据的问题，但是由于原生的服务通信仍然基于网络通信实现，在多个节点运行在同一主机的情况下，经过网络协议栈和虚拟网络接口的数据传递方式仍不够简洁，存在多余操作，增大了通信延迟。

通过以上分析可知，基于网络通信的原生 ROS 通信机制造成了不必要的通信延迟，本项目将基于上文给出的系统架构，将 ROS 通信机制改为基于内存实现，从而避免上述问题。

5.1.2 共享内存的通信拓展

基于内存的 ROS 通信相比于基于网络的 ROS 通信具有实时性上的优势。ROS 应用中，很多情况下会有多个 ROS 节点运行在同一个主机上，这时 ROS 节点间若仍使

用网络通信，数据会经过网络协议栈和虚拟网卡传递到目的节点，这期间会经历多余的数据拷贝和数据转化过程，造成额外的时间开销（如图 34 左图）。另外，在同一个数据有多个接收者时，数据发布者需分别向各个接收者发送同样的数据，同样会造成额外的时间开销。本论文观察到在多个 ROS 节点运行在同一主机上时，可以使用内存作为节点间通信的桥梁。具体来说，数据发送方和接收方可以共同访问同一片内存，传递数据时，发送方将数据写入内存，接收方将数据从内存中读出。这种实现下，数据的传递方式是简洁的，与基于网络通信的方式相比，大幅减少了数据拷贝和转化次数（如图 34 右图）。另外，在向多个接收者发布同一数据时，只需将数据一次性写入共享内存中，各接收者分别从共享内存中读出数据即可，避免了多次发送同一数据的时间浪费。

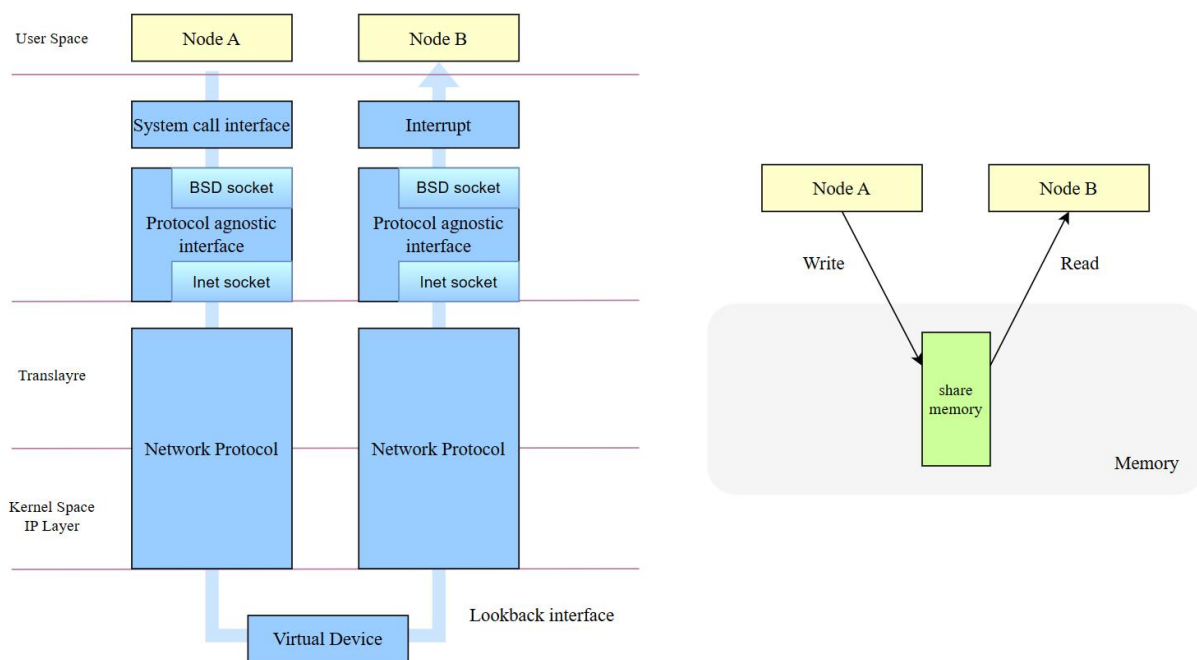


图 34. 基于本地网络设备的 ROS 通信与基于共享内存的 ROS 通信

在将 ROS 的通信机制由基于网络改写到基于内存的过程中，会遇到一些实现上的困难。由于本项目希望改写后的基于内存的 ROS 通信接口能简单的替换已有 ROS 应用中的通信代码，所以应保证在 ROS 通信接口的调用者看来，改写前后的通信接口的运行时行为应该是一致的。为此，需要（1）实现正确的同步机制，保证各节点读写共享内存时没有冲突。（2）在话题机制中，每个订阅者能接收到订阅话题后，该话题上



发布的所有消息。(3) 在服务机制中, 每个客户节点能够正确接收到与自己发出的请求对应的响应, 并在收到响应前阻塞。

具体的需要实现一个用户态 ROS 通信拓展库 `ros_shmem.h`, 主要为一个 C++ 类, `ros_shmem`, 用于存放所有的 ROS 直接操作函数和内建变量, 主要包括 `init`、操作句柄、轮询频率、进程状态、`log` 等级、`spin` 逻辑、回调执行和发布订阅的类以及服务端客户端的类, 及其相应的异步/同步通信方式。具体可分为以下三方面工作:

- ROS 基本操作库:

- `init` 与 `command line`: 通过命令行参数输入的“节点名称”和 `init` 创建函数, 这一部分主要实现的是 ROS 话题拓展的节点创建。具体则会在每个如同这样的节点建立的时候会伴随调用 `ros_shmem.h` 中实现的 `init` 函数, 在实际的 `init` 执行是会关联执行一个监控进程陷入 `while` 循环, 并内建了一个 `node_list`, 用于维护本进程创建的节点单元。
- `NodeHandle` 操作句柄: 在这里是需要实现一个可以操作话题交互函数的变量, 同时提供一个与未来直接接入 ROS Master 的话题注册的预留接口类 (实际在创建句柄的时候并不会直接执行注册操作)。
- `Rate` 轮询频率: 这里是直接在函数里进一步调用了 ROS 原生的 `loop_rate`, 这里的主要目的是统一类调用接口。
- `Process Status`: 实际 ROS 在执行过程中需要判断节点状态在就绪状态下即可正常通信, 这里的实际在执行中会调用到 `ros_shmem` 实现的关联监控进程是否正常进行, 这一点是通过内建变量在 `init` 的时候即将该进程关联的监控进程的 PID 放置在内建变量中, 但在接入 ROS Master 的情况下会直接调用原生的 `ros::ok()`。
- `Log Level`: 在这里实际会调用到 ROS 提供的消息 `log`, 并同步 `log message` 到 ROS `log` 中。比如调用过程中使用 `ROS_SHMEM_INFO("message", msg.data.cstr());` 实际会调用到 `ROS_INFO("message", msg.data.cstr());`。
- `spin` and `spinOnce`: 在补充的通信拓展中的 `spin` 实际会使原进程在此处于 `hang` 的状态, 一直进入监控进程循环判断消息标记状态, 然后在消息写入时触发调

用回调函数。在补充通信拓展中的 `spinOnce` 则是会在此执行一次回调情况判断，无论成功或是失败后均直接返回，在源程序中继续进行。在 `spin` 与 `spinOnce` 的实现上，从实现的作用角度上保证了与 ROS 的原生功能作用上保持同步。

● 话题机制

- **Publisher:** 在话题机制下的 Publisher 需要实现一个 `advertise` 以完成一个 Publisher 的建立，实际需要传入发布话题与消息队列的大小，然后需要实现一个 `publish` 内建函数用于消息的发布。在实际的执行过程中，在执行 `advertise` 后会通过第四章提到的 POSIX Share Memory 创建一个相应长度的共享内存，并对此片共享内存通过第四章提到的 `MessageQueue` 进行访问，在每次通过 Publisher 建立的对象通过内建的 `publish` 函数发布消息时，便会在 CLI 中实际操作 `Message` 中轮询获取当前 `MessageQueue` 定义的自旋锁位置的锁变量，并进一步通过当前 `write_ptr` 中的位置来写入消息。
- **Subscriber:** 在话题机制下的 Subscriber 需要实现一个 `subscribe` 用于建立对应话题的查询与订阅队列的大小，还有需要触发的回调函数。在实际的执行过程中，会同样根据话题名称基于第四章的 POSIX Share Memory 的 User Lib 创建一个对应大小的共享内存，然后通过第四章提到的 `MessageQueue` 对该片内存的锁变量位置进行轮询，在监测到已经收到消息时候便会通过传入的函数指针执行一次对应的回调函数，将 `read_ptr` 对应位置的 `msg` 收取，而后将自身视角下的 `read_ptr` 自增。

● 服务机制

- **Service 预定义库:** 在 Service 模式下的每个服务在原 ROS 的模式下也均需要预定义，ROS 软件本身在 `service_example` 中实现了部分 `service` 定义，在本项目也需要定义类似的 `service` 库，这里仅测试一个简单的加法操作，并将结果返回，与 ROS 原生库无差异。
- **Client:** Client 需要实现一个 `ServiceClient` 和 `serviceClient` 的函数调用，在 `serviceClient` 的调用下会通过预定义时直接建立一个对应服务名的客户服务，

会关联执行 POSIX Share Memory 分别创建对应服务名的 `.node_list`、`{ vmid << 16 | pid }_client` 和 `{vmid << 16 | pid}_server` 共享内存，在服务创建时会首先在 `node_list` 中加入当前节点的 `vmid << 16 | pid` 信息，然后在向 Server 发送请求时使用 MessageQueue 对 `{ vmid << 16 | pid }_server` 获取对应的写内存锁，在发送消息后对 `{ vmid << 16 | pid }_client` 轮询读位等待 server 的反馈消息，具体的 MessageQueue 的操作方式与话题中的方式相同。

- Server: Server 的实现需要实现一个 ServiceServer 类和 advertiseService 的操作函数，在 Server 创建后需要同时创建对应服务的 `.node_list`，然后通过 `struct node_list{ spinlock_t node_list_lock; int node_count; int rt_level; int nodes[64];};` 访问该结构体当前的 `node_count` 去遍历 `nodes` 去获得目前已经映射的 `{ vmid << 16 | pid }_client` 和 `{ vmid << 16 | pid }_server`，并创建对应的共享内存映射，然后根据实时等级轮询所有消息队列的 `_server` 读位，在有消息的时候即调用 Server 定义时确定的回调函数，然后将消息放到相应的 `_client` 消息队列的位置（此时因为 Service 下是一对一的，因此不需要使用锁），而后更新对应的 `_client` 的 `write_ptr` 即可实现一次服务。在 client 创建和退出时会在 `node_list` 中先获取自旋锁，然后添加或删除对应的共享内存映射，server 在每次处理的时候也会先请求自旋锁，而后查看 `node_list` 中的情况，根据对应的 `rt_level`，建立响应序列后逐一响应。

Python 版的实现需求和实现逻辑与以上类似。

5.2 实时进程在 Unikernel 的适配

5.2.1 Unikernel MicroROS

由于 ROS1 本身对于操作系统的绑定非常高，向 Unikernel 移植一个 ROS1 是一件非常困难的事，而实际基于本文提供的共享内存的 ROS 通信拓展是不需要绑定 ROS1 原框架的，并且在本文提供的共享内存的 ROS 通信拓展是可以将话题内容注入到本地通信框架的话题列表中的，因此可以在不同 VM 上使用不同的 ROS 框架。由于 Unikernel 本身并不适配完整的 ROS1 或 ROS2，但可以移植一个简化版的 ROS2 框

架——Micro-ROS。

Micro-ROS 是一个开源的机器人操作系统（如图 35），它可以将简化版的 ROS2 适配到资源非常有限的平台，并且可以完成绝大多数的本身基于 ROS 的通信功能。

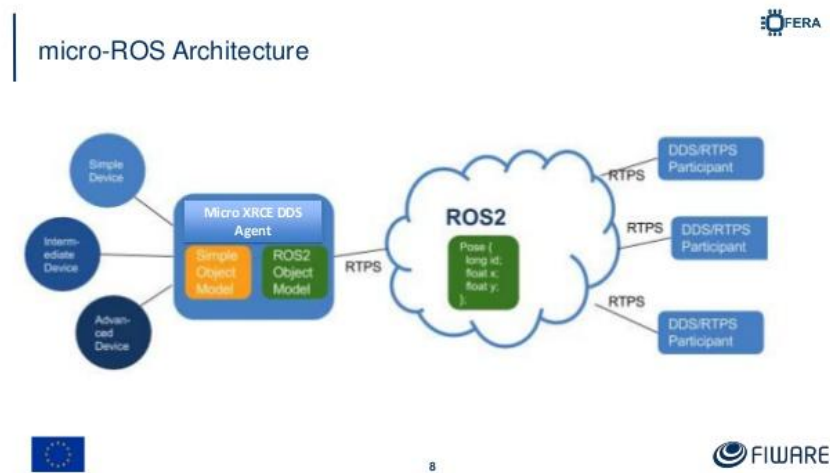


图 35. Micro-ROS 架构

为 Unikernel 移植 Micro-ROS 首先需要已经构建好 ROS2 的 Ubuntu/Debian 操作系统，而后可以通过以下命令构建 Micro-ROS 开发环境。

```
1 # Source the ROS 2 installation
2 source /opt/ros/$ROS_DISTRO/setup.bash
3
4 # Create a workspace and download the micro-ROS tools
5 mkdir microros_ws
6 cd microros_ws
7 git clone -b $ROS_DISTRO https://github.com/micro-ROS/micro-ros-build.git src/micro-ros-build
8
9 # Update dependencies using rosdep
10 sudo apt update && rosdep update
11 rosdep install --from-path src --ignore-src -y
12
13 # Build micro-ROS tools and source them
14 colcon build
15 source install/local_setup.bash
```

图 36. Micro-ROS 环境构建命令

5.2.2 ROS 实时进程移植

由 5.1 中对 ROS 通信机制的拓展，可以实现绝大多数的实时节点移植，这些基本



的 API 已经可以满足日常开发项目的需求，但目前 ROS1 的很多项目已经产生了很强的耦合度，会出现某个较为复杂的功能可能仅有某个子模块对实时性的需求比较高，但该子模块已经与该复杂功能的其他模块高度耦合，在类似的场景下是不易于移植到 Unikernel 的，但实际从应用场景思考，这类高度耦合的模块本身的实时性差也是因为耦合度较高的情境下，该应用会依赖很多实时性较差的节点传来的消息，或调用一些非实时的函数接口而导致实时性较差，这类不适宜移植到 Unikernel 的节点本身也很难直接通过系统支持硬实时性而获得实时。

在实际应用场景中，对 ROS 节点的移植可以主要分为以下几类：

独立模块：所有函数均可以在模块内独立实现，模块内的操作无需通过外界信号响应即可正常执行。这类 ROS 进程大概占 ROS 应用进程的 85%，且实时类进程通常属于这个范畴。这类进程主要包括机器人的各种传感器件的采集进程、传感器件的直接驱动/处理程序、不依赖图形界面的决策算法等，对于这类进程向 Unikernel 移植的主要评估点在于 Unikernel 是否支持相应传感器的驱动控制，以及 Unikernel 的软件生态是否可以支持这类进程的库调用，如果在 Unikernel 现有生态可以支持其运行依赖的话，那这类进程在不考虑移植工作量的情况下会非常适合于移植到 Unikernel 来达到实时性效果。

依赖原 Linux 系统中的话题数据用于执行特定功能的节点：进程需通过某些 ROS 话题的参数获得进程的执行依据，而后在运行中便可以独立执行一次相应功能的进程。这类 ROS 进程大概占 ROS 应用进程的 10%，这类进程中也可能会包含一些实时应用。这类进程主要有串口通信、机械臂总线式电机驱动等，这些进程通常会机器人的某些功能的执行器件，对于这类进程，通常推荐移植在 Unikernel 上，因为这类进程通常的控制来源也比较少，仅需对其发布节点通过共享内存的机制替换对应的函数即可把节点更换到 Unikernel 上，比如串口通信，在这种情况下如果控制节点在 Linux 上也至少可以达到绕过网卡的性能提升，如果控制节点也在 Unikernel 则是直接实现了硬实时性。但如总线机械臂，可能在使用的过程中需要配合 MoveIt 使用，在这种有上层依赖的情况下移到 Unikernel 仍需要在 Linux 上创建一个影子进程完成 MoveIt 的支持和共享内存的通信，在这种情况下是不推荐移植到 Unikernel 的。

在运行中需要频繁依赖 Linux 系统中传来的消息才能执行相应动作的节点：这种通常为机器人核心逻辑处的决策算法以及上层控制软件或可视化软件等。这类 ROS 进程大概占 ROS 应用进程的 5%，这类进程中大多不会产生实时需求，且绝大多数的实时限制也不在硬实时的支持上，通常这类进程的实时性需要同时也会由其上层和下层决定。但也有可能有些极端型需求，如将机器人的协同控制算法分配更多的时间片资源进行运算，但在这种情况下由于将其移植到 Unikernel 后仍然会存在原 Linux 下的 ROS 进程需要依赖其通信为上层运动逻辑提供支持的情况，这种情况则需要对该进程创建影子进程，用于同时满足上层 Linux ROS 对该进程的话题通信需求和共享内存通信需求，而这种情况下引入的两段通信需求和分到的额外时间片之间需要做一个平衡取舍。

以 rrt_exploration 算法举例，这种情况下的移植评估可以通过对 Python 的 cProfile 来实现，在实际运行对应协同控制算法时可以在命令行加入 “-m cProfile -o output.prof” 参数来实现输出运行过程的 timeline（如图 37），而后可以使用 snakeviz 等工具查看其 timeline 的执行情况，如果移植后的通信时延 + 算法计算的执行时长小于原算法的执行时长，则移植是有效的，反之则不应当移植。（对于 C++ 程序可以通过 ARM Development Studio 中的 Streamline 进行 Profile，实际的对比指标与 Python 相同）

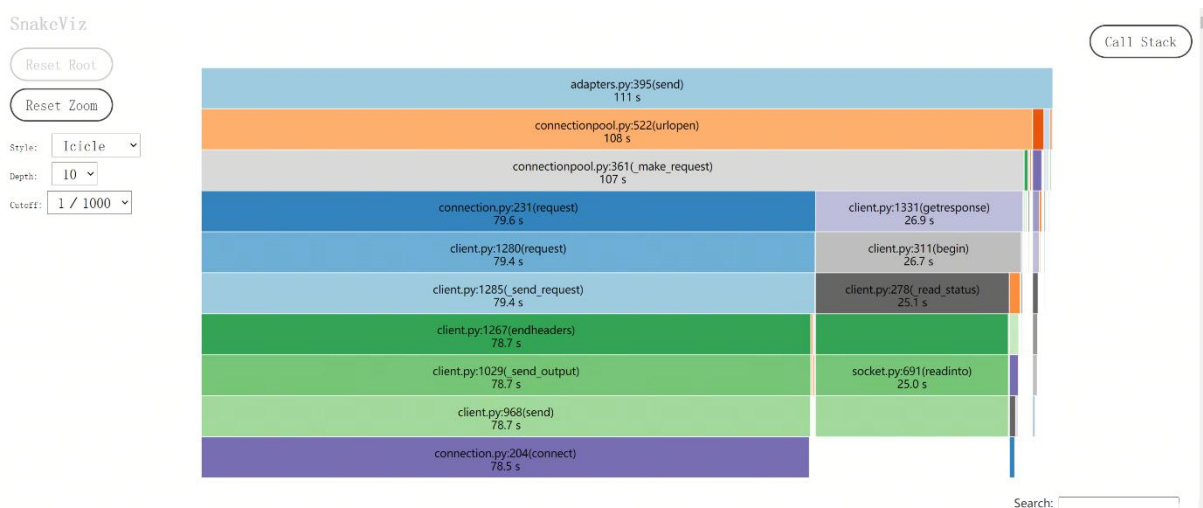


图 37. Python 的 cProfile timeline

在实际移植过程中，可能对于某些既需要通过向 Linux 的发送原生的 ROS1 话题信息，由期望可以通过共享内存优化实时性的场景，比如自动建图的过程，实际自动

建图、运动逻辑、地图搜索算法均可以通过在 Unikernel 实现达到一个很高的实时性效果，但由于 Linux 的可视化软件需要通过原生的 ROS 话题去读取机器人当前的空间位置，在类似这种场景下就可以通过在 Linux 上建一个 Shadow Process 的方式，它实际会与 Unikernel 建立一个共享内存通信，来获取到相应的话题内容，然后在这个进程将该话题的内容信息转发到 ROS 原生话题中，这样就可以同样在可视化软件上得到相应的话题信息了，而且在这种情况下，其他进程均运行在 Unikernel，具有一个很高的计算实时性，在这种情况下同样是一个有效的移植。

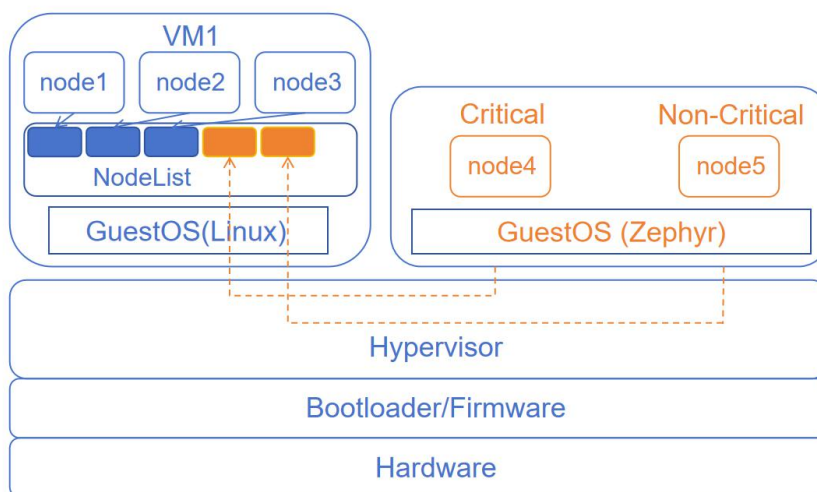


图 38. Shadow Process 下节点在该混合关键系统下的存在形式

5.3 Linux + Unikernel 下对 ROS 节点的实时调度

当在 Unikernel 上分配较多的实时线程，且 Linux 上本身也存在着很多的软实时进程下，实际在程序运行的不同阶段，对 Unikernel 上的线程的实时需求可能会产生变化，而且在整个混合系统的运行下，排除部分系统进程，当系统在复杂的应用场景下运行较多的进程导致 CPU 数不足以为每个任务直接绑定一个时，通过一端的调度决策往往能在这些复杂场景下取得更好的效果。

在实时场景下，目前应用较多的实时调度算法有固定优先级抢占式调度、最早截止时间优先调度、最小松弛时间优先调度、反馈控制实时调度等等，但在跨 VM 的管理下，这些调度方式在跨 VM 时的管理较为繁琐，且调整的灵活性较差，并不实用。

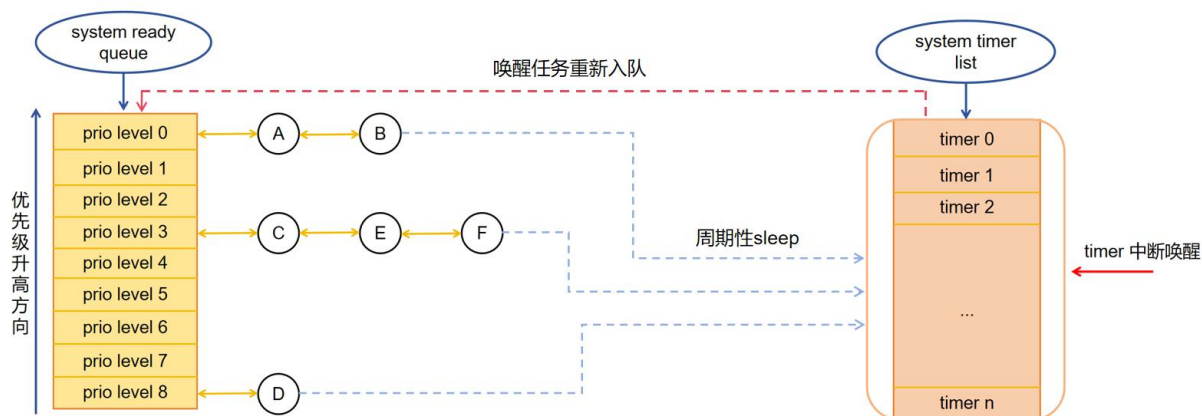


图 39. 任务调度流程示意图

具体 AMLFQ 的调度策略如下：

1. 首先需要为每个任务 Linux/Unikernel 分配其计划完成时间与超时代价，和 Linux 与 Unikernel 运行线程间的亲和关系（不设置则默认没关系）
2. 当 Unikernel 任务进入系统调度时，直接放在最高优先级参与调度（最上层队列）
3. 当一个任务用完整个时间片前主动放弃 CPU 时间片的线程在调度队列中的优先级不变
4. 当一个任务用完整个时间片后，如果是实时任务，则保持不变，如果是非实时任务，则降低其优先级，移至下一级队列
5. 在每个实时任务用完其时间片后会根据亲和关系判断一次当前任务调度代价，若通过亲和关系判断出了饥饿现象则调高相应亲和实时线程的优先级
6. 实时任务在任务数大于 CPU 数时需要定期休眠

这样对于一个非实时的大任务在 Unikernel 中的调用就会如图 40 所示，可以保证所有任务都可以及时地被响应，并且对实时要求较高的线程可以分到足够多的时间片，且计算较复杂且不关键的大任务会逐渐减少 CPU 的时间片占用。当任务量较多时，AMLFQ 可以及时的通过实时性和亲和关系判断可以有效的预防线程饥饿现象。



图 40. 非实时的大任务在该混合关键系统下的调度情况

5.4 本章小结

本章主要介绍了本项目在基于跨 VM 的共享内存通信基础上对 ROS 原有话题机制的拓展，以及对 ROS 实时节点的移植，并根据在移植 ROS 实时节点中碰到的一些问题，提出了相应的移植参考和移植建议。并基于在移植过程中的性能经验提出了 AMLFQ 的基于跨 VM 的实时任务与任务间的亲和性的多机反馈队列调度方式，并分析了该调度方式在特定场景下的执行效果。

第六章 性能测试

6.1 测试平台说明

本基于 Type-1 型虚拟机 Rust-Shyper，在 Rust-Shyper 上启动多个虚拟机 VM(Virtual Machine)，将实时性要求高的 ROS 节点单独部署在特定 VM 中，并利用 Rust-Shyper 对上层 VM 调度的灵活性，保证实时性要求高的 ROS 节点所在 VM 拥有更高的时间片占用率，从而提高 ROS 应用系统的实时性。另外，本项目在 Shyper 层和 VM 层向上提供了共享内存和自旋锁接口，并基于将原本基于网络的 ROS 通信机制改写到基于内存，进一步降低了单系统内 ROS 节点间通信延迟，从而提高系统整体实时性。

本章将全面测试系统各优化环节的实时性提升，包括将 ROS 通信机制改写为基于内存后对 ROS 节点间通信延迟的影响和基于实时性的对 ROS 节点向多个 VM 的合理分配及多进程/线程的混合关键系统下任务的合理调度对系统实时性的影响。最后，给出了本项目在实际应用场景下，对实时性的优化效果。

6.2 混合关键系统实时性能分析

6.2.1 虚拟化后原 VM 的实时性能

对于任何一个 Hypervisor 来说，中断处理延迟以及虚拟机退出 (VM EXIT) 的开销，都会极大程度上决定了上层虚拟机的计算性能以及实时性能。本文采用 Rust-Shyper 通过引入中断控制器部分直通 (GPPT) 以及任务远程核心分配 (TRCA) 这两种机制在最大程度上减少上述两类开销。图 41 是文件操作延迟、本地通信带宽、进程创建延迟、系统调用延迟，将 Linux Native 作为基准，在 Shyper 虚拟化后的性能差异前后对比图。

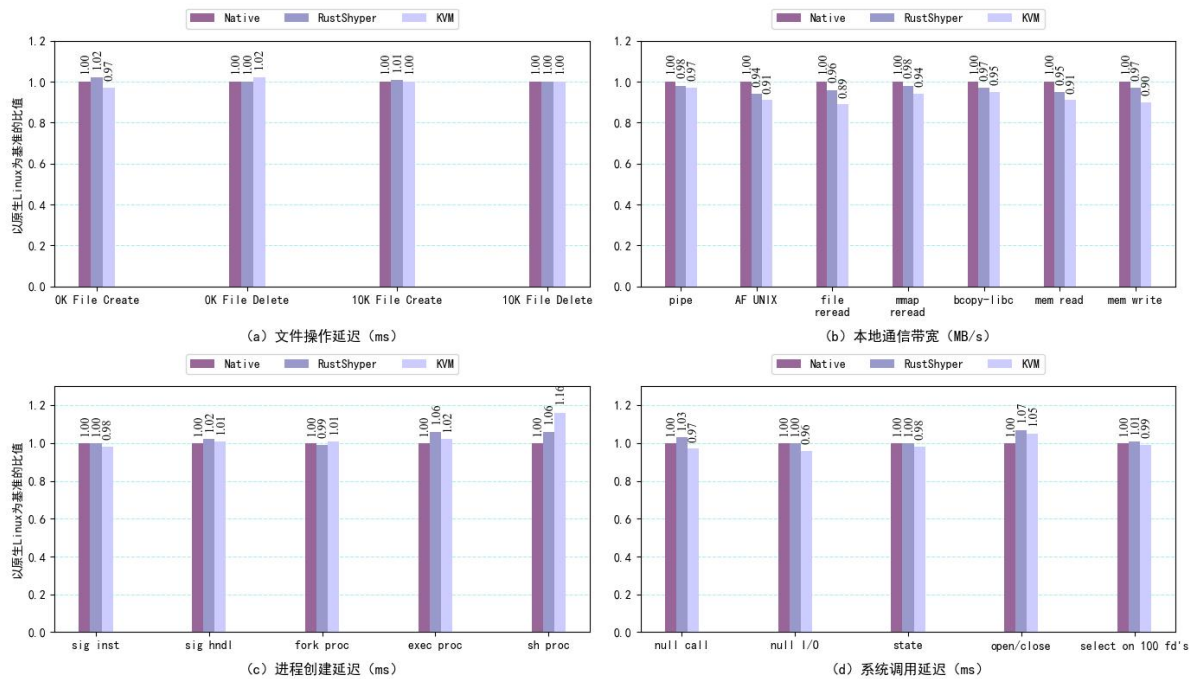


图 41. 虚拟化后的相关系统基础性能与 Native Linux 对比图

6.2.2 LMBench

LMBench 是一种用于测量计算机系统性能的基准测试工具。它通过一系列微基准测试 (micro-benchmarks) 来评估系统的不同方面, 包括处理器、内存、文件系统和网络性能。LMBench 的设计目标是提供一种快速、简单且可移植的方法来衡量系统性能, 并能够比较不同系统的性能差异。LMBench 的主要功能和特性包括:

- 处理器性能测试:
 - 上下文切换时间: 衡量不同进程和线程之间切换的时间。
 - 系统调用开销: 测量简单系统调用 (如 `getpid()`) 的执行时间。
 - 进程创建时间: 评估创建新进程 (如 `fork()`) 的时间。
 - 管道带宽和延迟: 测试进程间通信的效率。
- 内存性能测试:
 - 内存读写带宽: 测量内存的读写速度。
 - 缓存和内存层次结构延迟: 评估从处理器缓存到主内存的访问延迟。
- 文件系统性能测试:
 - 文件创建、删除和状态查询时间: 测量文件系统基本操作的效率。

- 磁盘 I/O 带宽：评估磁盘读写速度。
- 网络性能测试：
 - 网络带宽和延迟：通过不同的网络协议（如 TCP、UDP）测试网络传输的效率和响应时间。

在本项目中在本地启用了 Preempt-RT Linux、Shyper with Preempt-RT Linux、Preempt-RT Linux with KVM，并以 Native Linux 作为基准归一化得到以下结果。

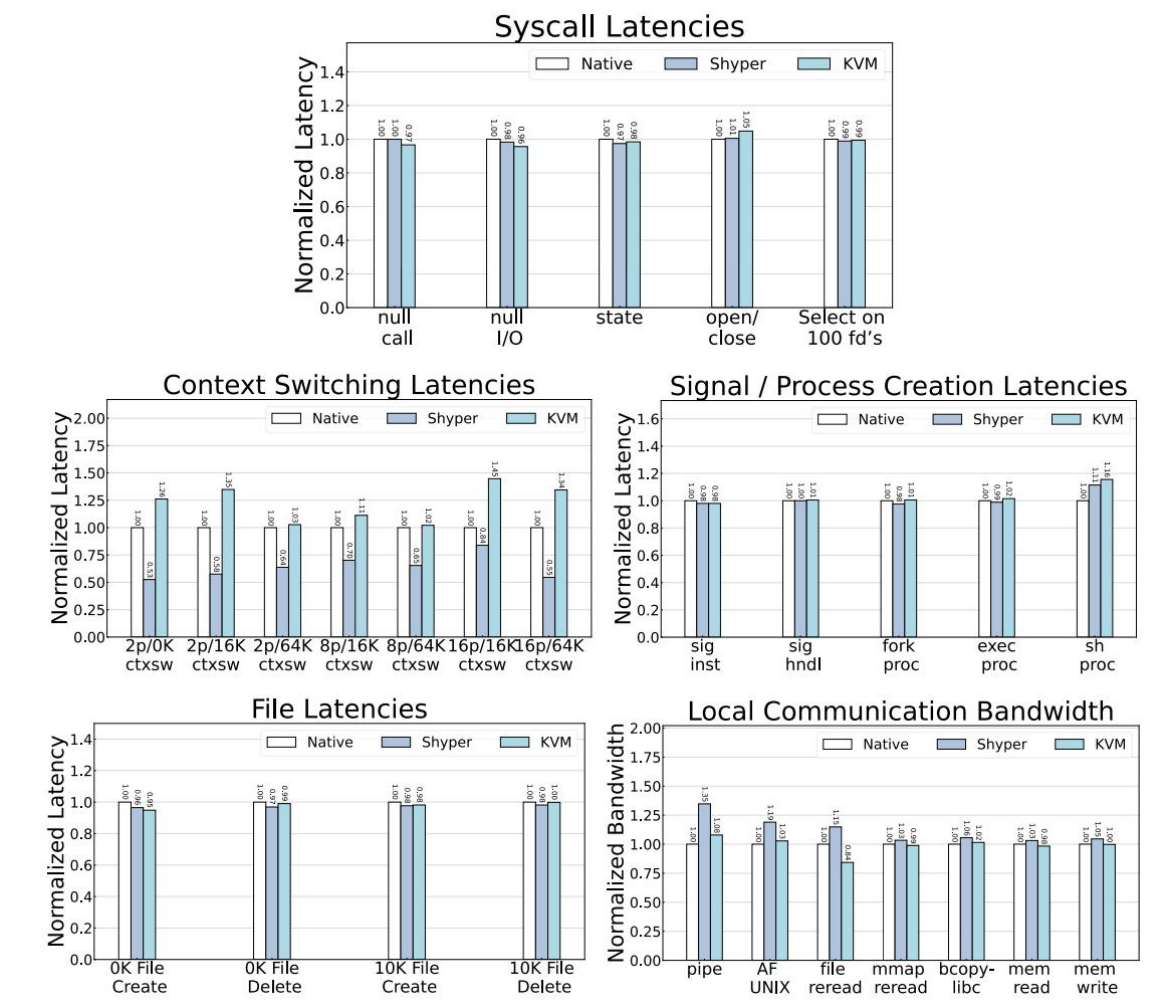


图 42. LmBench 测试结果

6.2.3 不同数据量下的跨 VM 传输时延

传输时延是衡量 ROS 节点通信过程延迟的重要指标。在本节是使用 ROS 话题机制，并以单发布者单接收者情况作为基准测试程序。这部分我们在本项目的 Rust-shyper 多

虚拟机架构上以传输数据量为变量，对跨虚拟机通过网络读写，虚拟机内通过网络读写，跨虚拟机通过内存读写，虚拟机内通过内存读写共八种情况测试了传输延迟，所得数据如图 44、图 45 所示。

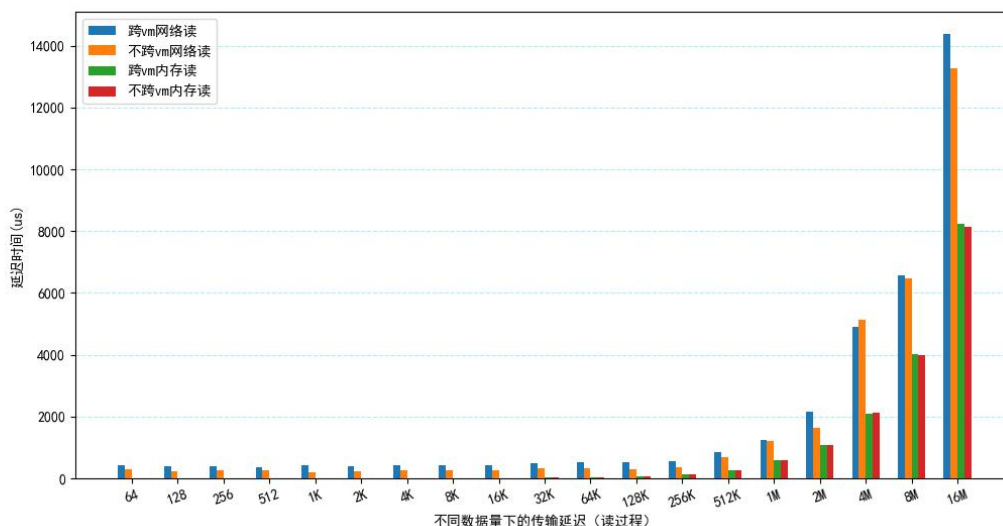


图 43. 通过内存的读操作的传输时延对比图

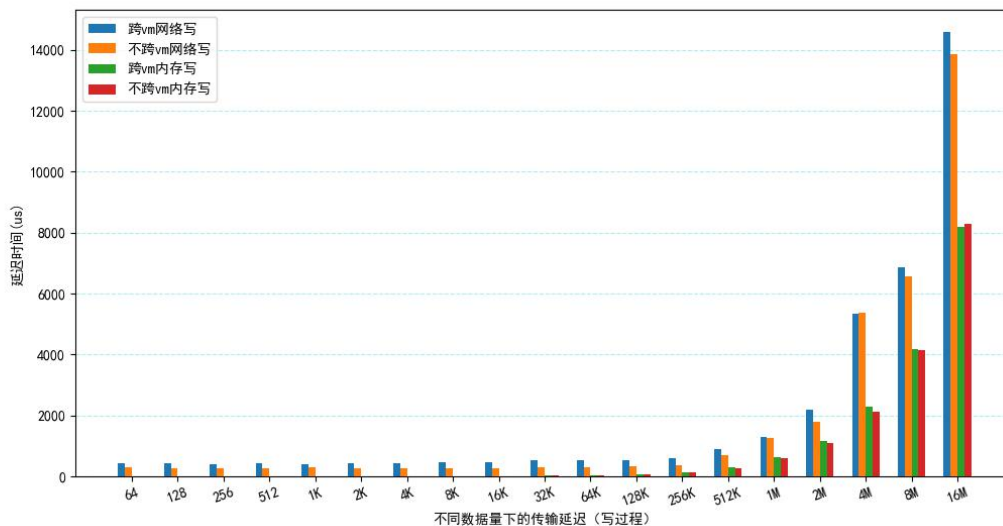


图 44. 通过内存的写操作的传输时延对比图

从测试结果可以看出，总体而言，基于内存的读过程和写过程的传输延迟要大幅低于基于网络的读写过程。对于数据量较大的情况，基于内存的读写过程的传输延迟大致为基于网络的对同样数据量的读写传输延迟的一半，基于在内核测试的总结，应当是单个节点内的网络通信通过 loop device 最后仍然是在内存中存在。

6.2.4 不同数据量下的响应时延表现

响应时延是 ROS 两节点间通信过程中，从一方产生数据，经数据发送与传输过程，到另一方做出数据接收响应这一过程的总延时。不难发现，响应时延直接反映出 ROS 应用实时响应的速度，直接影响了节点间的通信频率。这一部分，我们仍使用单发布者单接收者的话题机制通信作为基准测试程序，以传输的数据量作为变量，对虚拟机内部和虚拟机之间基于网络和内存通信进行了全面的延时测试，测试结果如图 46、图 47。可以看出，在传输数据量在 64B 到 16MB 范围内，基于内存的通信延迟全面低于基于网络的通信。尤其对于数据量较小（64B 到 32K）的通信情况，基于共享内存的通信延迟保持在 10us 以内，而基于网络的通信延迟在 500us 以上，Shyper – RTLinux – RTOS 相比 Native Linux 在小数据的实时通信传输速度上获得了 68.6x 的性能提升，这种延迟优势来源于基于内存通信相比于基于网络对函数调用栈的简洁和对数据处理的便捷。

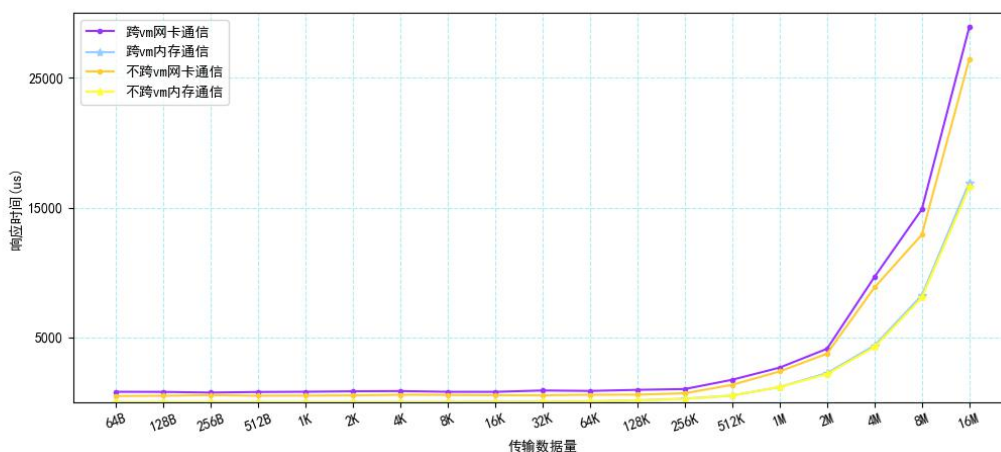


图 45. 通过共享内存的传输时延在不同数据量下的表现

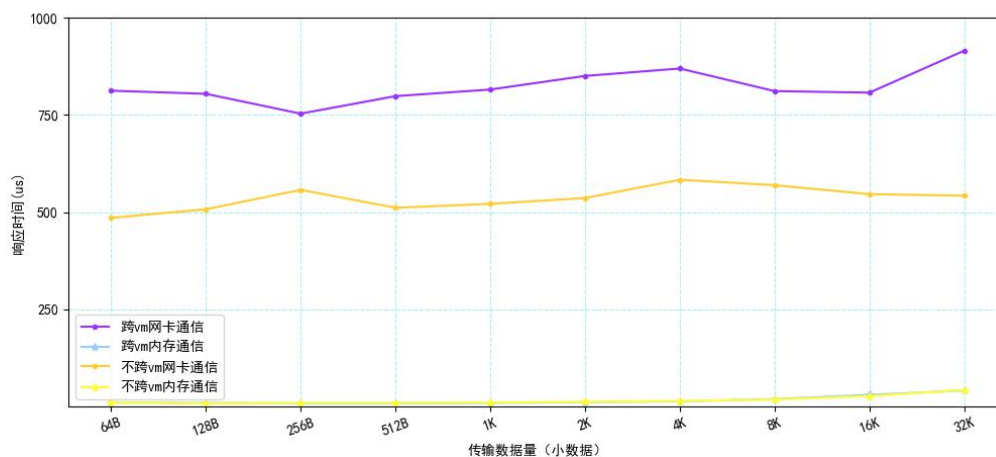


图 46. 通过共享内存的传输时延在 64B-32K 下的表现

6.2.6

6.3 ROS 应用实时性能

ROS 应用程序的实时性能评估是指机器人系统在运行过程中能够及时响应各种传感器输入、执行各种任务和控制动作的可确保性效果。

6.3.1 ROS_Bench

ROS_Bench 是基于 ros2_benchmarking 项目适配到 ROS1 中来测试 RobotAlarm message、RobotControl message、RobotSensor message 的接收率和接收时延随这几种消息丢包率的变化，它代表了系统在高负载下随丢包率的上升时系统对相应种类任务的响应效果是否还可以满足要求。

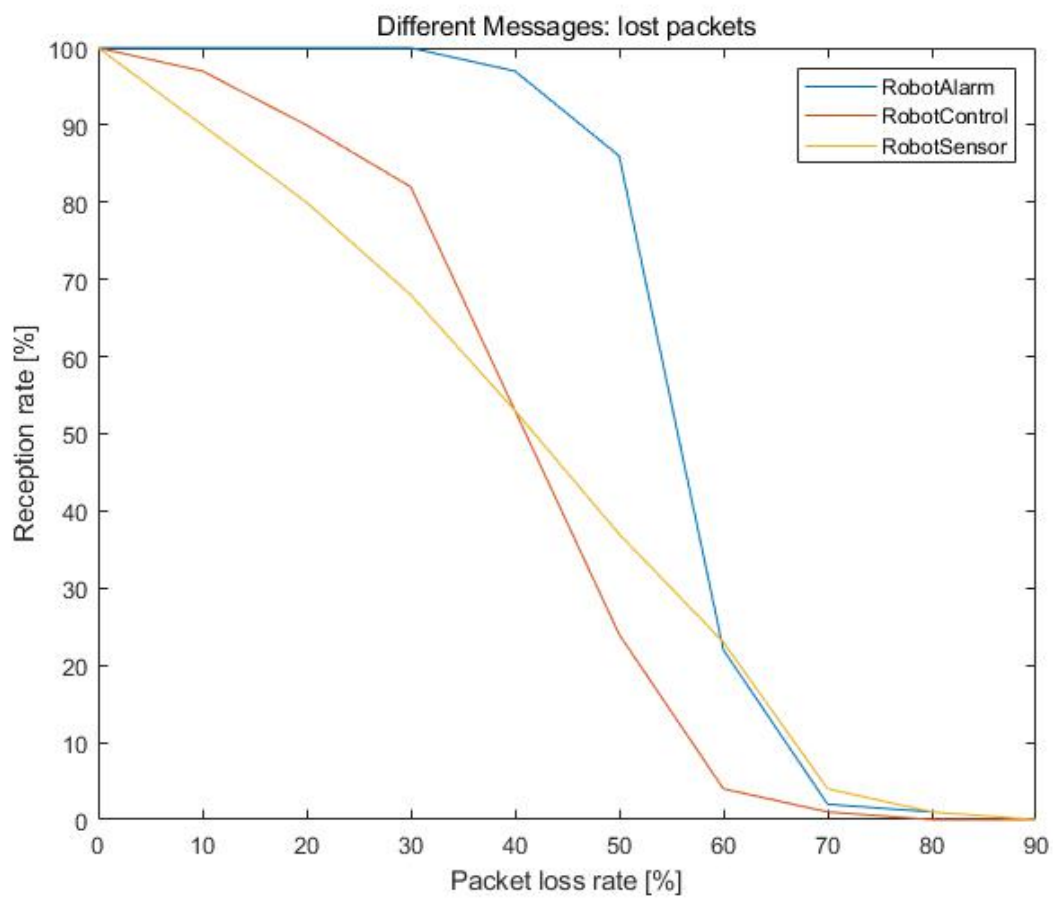


图 47. 不同类型数据在不同丢包率下的响应百分比

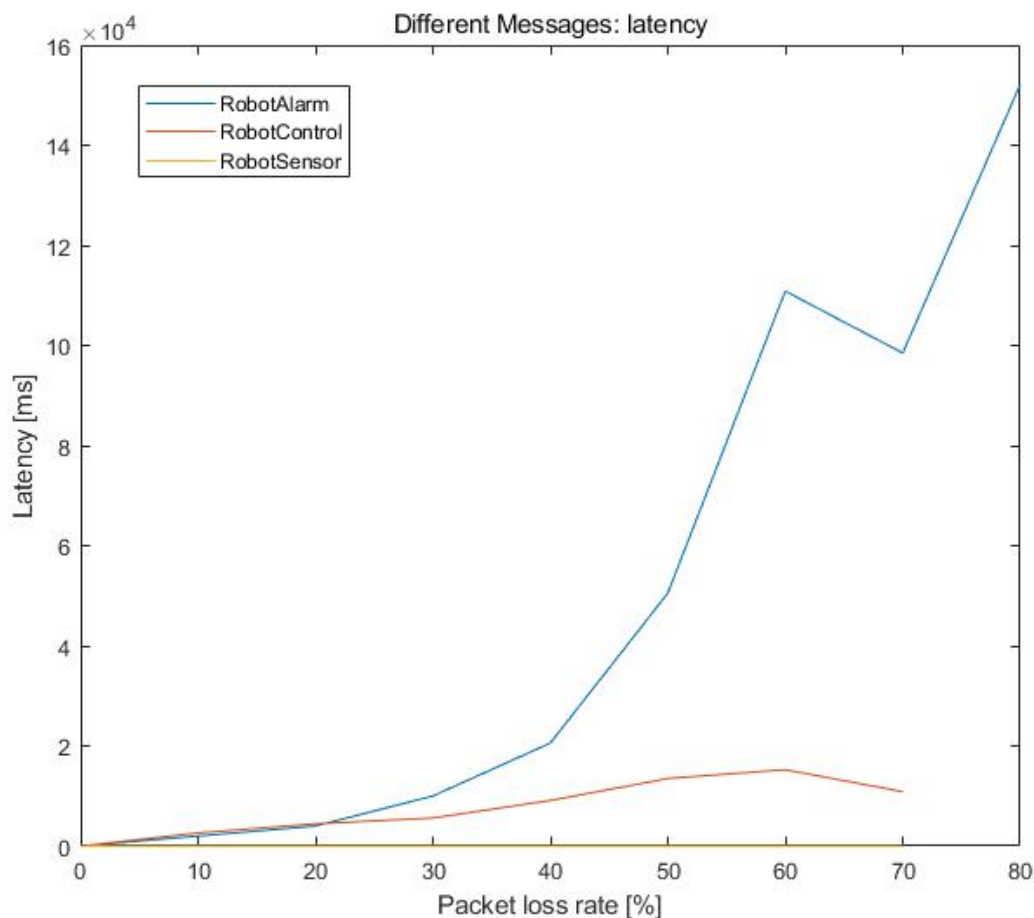


图 48. 不同类型数据在不同丢包率下的响应时延

6.3.2 实时性性能在多接收节点下的表现

ROS 应用中，经常会有一个话题上有一个发布者和多个订阅者的情况。由于采用网络通信时，发布者需逐个将消息发送到订阅者，这导致最后接收消息的订阅者的响应时延很长，限制了系统的实时性。而当采用本项目实现的基于共享内存的通信时，由于各订阅者主动访问共享内存，只要系统能分配资源执行订阅者，订阅者就能保证在一定时间内做出响应，而不必等待发布者向订阅者逐个发布消息。这一部分，我们运行单发布者，多订阅者的话题应用，在传递 1KB 数据的条件下，测量了订阅者中的最长响应时间，统计数据如图 53。从图中可以看出，基于共享内存情况下，多订阅者的最长响应时间与单订阅者情况差距较小，响应时间几乎不受订阅者数量影响。另外，在订阅者数量增加到 7 之后，最长响应时间的增加是由于 CPU 核数资源被实时进程占

满，存在订阅者需要等待调度。

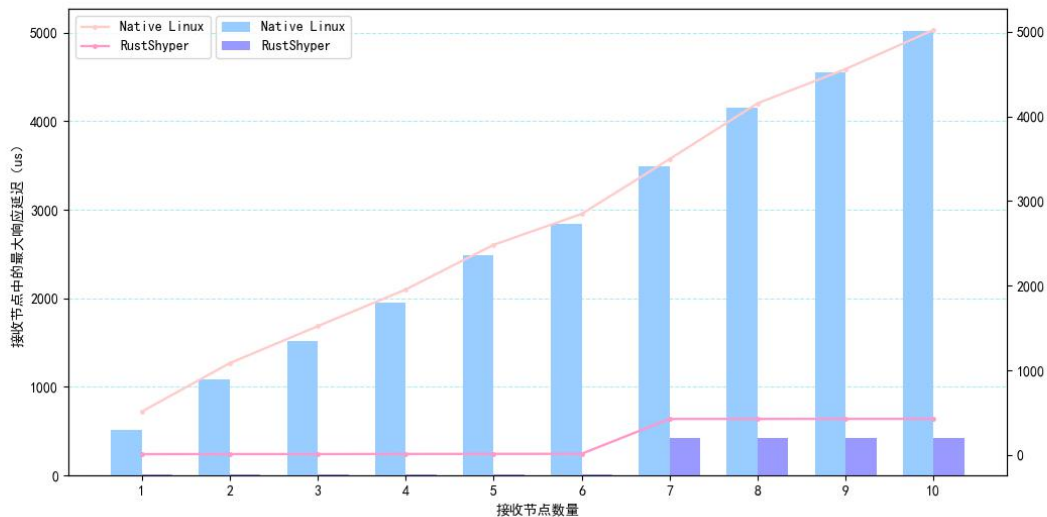


图 49. 不同接收节点数量下接收 1KB 数据的传输时延变化图

6.4 本章小结

本项目的实验设计完备地自底向上测试了整个系统对实时性的提升。从实验结果看来，本项目从底层系统的任务处理速度和对系统调用的响应速度，到实际应用中不同负载下 ROS 通信的响应速度都具有明显提升，从整体上提升了实时性，为 ROS 应用部署提供了一个实用的，高实时性的架构。



总结与展望

ROS1 的实时性一直是其在关键领域应用时较受争议的点，本项目在混合关键系统下对 ROS1 原生态得以兼容并使得实时应用可以在几乎无感的情况下移植到 Unikernel 而获得硬实时能力，这对于 ROS1 原有生态的推广和 ROS1 在关键领域的应用是一个非常重要的突破，基于本项目搭建的 ROS1 混合关键系统环境可以在 MVM 上达到与 Native 仅 3% 内的性能差异下完全兼容原生态，同时可以在 HRTVM 上达到实际意义上的硬实时性。

具体的，本文完成的工作如下：

软硬件协同构建了一套对 ROS 机器人的混合关键系统架构。基于 JetRover 替换了电路的降压、稳压逻辑，并将主控板更替为国产的 RK3588，同时完成了原 Jetson Nano 中的 ROS 应用到 RK3588 的适配，并刷新了 RK3588 的 Bootloader，使其支持 NVMe 硬盘驱动，在此基础上进一步完成了 Shyper 与 Unikernel HRTVM 的启动，同时完成了 mali gpu 和 npu 在 Shyper MVM 的适配。

仿照 POSIX Share Memory 接口实现了一个用户库，并添加了用户态对 Linux 内核的系统调用方式；增设了一个 shyper.ko 的内核模块，会创建 /dev/shyper 和 /dev/posix_shmem 设备，可以通过这两个设备实现 ioctl、mmap 等函数的重映射以实现自定义的系统调用；在内核模块中添加了本机内的 Share Memory 维护，使 Share Memory 的维护更有层次化，不需要每次均陷入 Hypervisor，尽可能地保证了 VM-Exit Less 的设计理念；添加了 Shyper 中对共享内存的映射逻辑，可以通过上层 HVC 调用完成共享内存的映射；基于共享内存存在用户态实现了一个 MessageQueue，通过规定的数据结构轮询处理自旋锁，以此达到跨 VM 共享内存区域的读写同步。

为 ROS1 提供了一个共享内存话题通信与服务通信的拓展，实现了基本的 init、command line、NodeHandler、Rate、Process Status、LogLevel、spin 等基本逻辑与 Publisher、Subscriber、Client、Server 以及预定义操作库，支持了绝大多数情境下的 ROS1 开发需求；提出了一套在当前混合关键系统场景下的 ROS1 进程的实时性移植简



易，并提供了串口等传感设备与 `rrt_exploration` 等协同计算控制算法的移植样例；实现了一个基于 AMLFQ 的覆盖跨 VM 的进程调度方案，满足了跨 VM 场景下的协同抢占调用的基本需求，针对硬实时场景下可以保证实时任务获得足够的时间片，同时可以使小任务得到迅速的响应，并可以检测非实时的大任务执行，较大程度地避免了非实时的大任务对于系统资源的争抢。

在以上实现的基础上对系统进行了虚拟化损耗与整个 ROS 混合关键系统的实时性测试，经验证，本系统可以在 64B 到 32K 的小数据的实时通信传输速度上获得相比 Native Linux 的 68.6x 的性能提升；在较复杂的 32 个任务场景下，高实时传输 1KB 数据的传输时延上获得了 118.6x 的性能提升；证实了该混合关键系统对实时性有非常高的改善效益，以及在兼容 ROS1 的实时场景下有较大的应用意义。

从实现的视角来看，虽然项目已经基本实现了对于 Linux 原 ROS1 生态的完整兼容和在运行起该混合关键系统下几乎无痛的实时性程序移植，但本项目还是存在一定的优化点的，主要集中在跨 VM 部分的同步方式上，当前项目主要关心的点是极致的实时性能与 VM-Exit less，对于特定内存的同步操作需要较大的 CPU 时间片用于轮询自旋锁，未来项目期望可以实现跨 VM 的互斥锁、读写锁以及信号量等同步方式，或对自旋锁的轮询单独实现可被抢占式，这样可以在通用场景下获得更好的使用体验。

参考文献

- [1] Lee M, Krishnakumar A S, Krishnan P, 等. Supporting soft real-time tasks in the xen hypervisor[J].
- [2] Sundar Rajan A K, Feucht A, Gamer L, 等. Hypervisor for consolidating real-time automotive control units: Its procedure, implications and hidden pitfalls[J]. Journal of Systems Architecture, 2018, 82: 37 – 48.
- [3] Casini D, Biondi A, Cicero G, 等. Latency Analysis of I/O Virtualization Techniques in Hypervisor-Based Real-Time Systems[A]. 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)[C]. Nashville, TN, USA: IEEE, 2021: 306 – 319.
- [4] Jiang Z, Audsley N C, Dong P. BlueVisor: A Scalable Real-Time Hardware Hypervisor for Many-Core Embedded Systems[A]. 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)[C]. Porto: IEEE, 2018: 75–84.
- [5] Xi S, Wilson J, Lu C, 等. RT-Xen: towards real-time hypervisor scheduling in xen[A]. Proceedings of the ninth ACM international conference on Embedded software[C]. Taipei Taiwan: ACM, 2011: 39 – 48.
- [6] Gain G, Soldani C, Huici F, 等. Want more unikernels?: inflate them![A]. Proceedings of the 13th Symposium on Cloud Computing[C]. San Francisco California: ACM, 2022: 510 – 525.
- [7] Eiling N, Kröning M, Klimt J, 等. GPU Acceleration in Unikernels Using Cricket GPU Virtualization[A]. Proceedings of the SC ' 23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis[C]. Denver CO USA: ACM, 2023: 1588–1595.
- [8] Kuo H-C, Williams D, Koller R, 等. A Linux in unikernel clothing[A]. Proceedings of the Fifteenth European Conference on Computer Systems[C]. Heraklion Greece: ACM, 2020: 1 – 15.
- [9] Raza A, Unger T, Boyd M, 等. Unikernel Linux (UKL)[A]. Proceedings of the Eighteenth European Conference on Computer Systems[C]. Rome Italy: ACM, 2023:



590 – 605.

- [10] Raza A, Sohal P, Cadden J, 等. Unikernels: The Next Stage of Linux' s Dominance[A]. Proceedings of the Workshop on Hot Topics in Operating Systems[C]. Bertinoro Italy: ACM, 2019: 7 – 13.
- [11] Williams D, Koller R, Lucina M, 等. Unikernels as Processes[A]. Proceedings of the ACM Symposium on Cloud Computing[C]. Carlsbad CA USA: ACM, 2018: 199 – 211.
- [12] Reghenzani F, Massari G, Fornaciari W. The Real-Time Linux Kernel: A Survey on PREEMPT_RT[J]. ACM Computing Surveys, 2020, 52(1): 1–36.
- [13] Lin C-H, Wu C-K. Performance Evaluation of Xenomai 3[J].
- [14] Yang C-F, Shinjo Y. Obtaining hard real-time performance and rich Linux features in a compounded real-time operating system by a partitioning hypervisor[A]. Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments[C]. Lausanne Switzerland: ACM, 2020: 59–72.
- [15] Lupu C, Albişoru A, Nichita R, 等. Nephele: Extending Virtualization Environments for Cloning Unikernel-based VMs[A]. Proceedings of the Eighteenth European Conference on Computer Systems[C]. Rome Italy: ACM, 2023: 574–589.



附录 A 脚本及测试源码

1. HVC 测试

```
#include <unistd.h>
#include <fcntl.h>
#include <linux/ioctl.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/file.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <wait.h>

int test_shmem_init(char *name) {
    int shyper_fd = open("/dev/shyper", O_RDWR);
    struct {
        char *name;
        int name_len;
    } ioctl_release;
    ioctl_release.name = name;
    ioctl_release.name_len = strlen(name);
    ioctl(shyper_fd, 0x1302, &ioctl_release);
    close(shyper_fd);
    return 0;
}

int main() {
    char *name = "my_shared_memory";
    test_shmem_init(name);
    return 0;
}
```

2. Linux POSIX Share Memory 测试

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
```



```
#include <unistd.h>

#define SHM_NAME "/my_shared_memory"
#define SHM_SIZE 1024 // 1KB

int main() {
    int shm_fd;
    void *ptr;

    // 创建共享内存
    shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    if (shm_fd == -1) {
        perror("shm_open");
        exit(EXIT_FAILURE);
    }

    // 设置共享内存大小
    if (ftruncate(shm_fd, SHM_SIZE) == -1) {
        perror("ftruncate");
        exit(EXIT_FAILURE);
    }

    // 映射共享内存
    ptr = mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }

    // 使用共享内存
    sprintf((char*)ptr, "Hello, shared memory!");

    // 取消映射共享内存
    if (munmap(ptr, SHM_SIZE) == -1) {
        perror("munmap");
        exit(EXIT_FAILURE);
    }

    // 关闭共享内存文件描述符
```




```
if (close(shm_fd) == -1) {  
    perror("close");  
    exit(EXIT_FAILURE);  
}  
  
printf("success");  
return 0;  
}
```

3. Linux MISC Device 测试

```
#include <stdio.h>  
#include <stdlib.h>  
#include <fcntl.h>  
#include <unistd.h>  
#include <string.h>  
#include <errno.h>  
  
#define DEVICE_PATH "/dev/misc_device"  
#define BUF_SIZE 256  
  
int main() {  
    int fd;  
    char write_buf[BUF_SIZE] = "Hello, Kernel!";  
    char read_buf[BUF_SIZE];  
  
    // Open the device  
    fd = open(DEVICE_PATH, O_RDWR);  
    if (fd < 0) {  
        perror("Failed to open the device");  
        return errno;  
    }  
    printf("Open device, fd = %d\n", fd);  
  
    // Write to the device  
    if (write(fd, write_buf, strlen(write_buf)) < 0) {  
        perror("Failed to write to the device");  
        return errno;  
    }  
  
    printf("fd = %d\n", fd);  
    fd = open(DEVICE_PATH, O_RDWR);
```



```
printf("fd = %d\n", fd);

// Read from the device
if (read(fd, read_buf, BUF_SIZE) < 0) {
    perror("Failed to read from the device");
    return errno;
}

printf("Read from the device: %s\n", read_buf);
// Close the device
close(fd);

return 0;
}
```

4. Linux multi vm Share Memory 测试

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include "../src/include/posix_shmem.h"

#define SHM_NAME "/my_shared_memory"
#define SHM_SIZE 1024 // 1KB

int main() {
    int shm_fd;
    void *ptr;

    // 创建共享内存
    shm_fd = shyper_shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    if (shm_fd == -1) {
        perror("shm_open");
        exit(EXIT_FAILURE);
    }

    // 设置共享内存大小
    if (shyper_ftruncate(shm_fd, SHM_SIZE) == -1) {
        perror("ftruncate");
    }
}
```



```
    exit(EXIT_FAILURE);
}

// 映射共享内存
ptr = shyper_mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
shm_fd, 0);
if (ptr == MAP_FAILED) {
    perror("mmap");
    exit(EXIT_FAILURE);
}

printf("start: %p\n", ptr);

// 使用共享内存
sprintf((char*)ptr, "Hello, shared memory!");

printf("mmap success: %s\n", (char*)ptr);

// 取消映射共享内存
if (shyper_munmap(ptr, SHM_SIZE) == -1) {
    perror("munmap");
    exit(EXIT_FAILURE);
}

// 删除共享内存
if (shyper_unlink(SHM_NAME) == -1) {
    perror("unlink");
    exit(EXIT_FAILURE);
}

printf("all success");
return 0;
}
```

附录 B 项目成果展示视频

百度网盘链接: <https://pan.baidu.com/s/19rIWCG1ubRxW4XzrXgRDIQ?pwd=si2f>