Jailhouse 到 RISC-V 移植支持 项目申请书

申请人：曹俊杰　项目导师：char　项目编号：25b970275

1. 项目背景与关键概念

1.1 项目背景

随着嵌入式系统在工业控制、车载电子和 IoT 等领域的广泛应用，系统对安全性、实时性和功能隔离的需求日益凸显。openEuler Embedded 致力于在一块硬件平台上同时承载多种操作系统，以满足不同任务对性能和安全隔离的要求。为此，引入了**混合关键系统（Mixed-Criticality Systems, MCS）** 的概念，通过静态分区的方式，将资源划分给不同 OS，使得高可靠性或硬实时任务与通用任务互不干扰。

Jailhouse 是西门子主导开发的轻量级静态分区 Type-1 Hypervisor，它通过预先配置“cell”的方式，把 CPU、内存和外设直接划归给不同的 Guest OS，避免了典型 Type-1 Hypervisor 的复杂调度开销。在 ARM/x86 平台上，openEuler Embedded 已经成功地基于 Jailhouse 实现 MCS 部署，但在新兴的开源指令集 RISC-V 生态中，现有主干尚无完善支持。与此同时，RISC-V 刚刚推出的 Advanced Interrupt Architecture（AIA）为中断直通提供了原生硬件支持，**亟待与 Jailhouse 结合，以满足高实时性系统对中断延迟的严格要求**。

因此，本项目旨在：

a. 完成对 Jailhouse 设计的全面分析，并提出 Jailhouse 在 RISC-V 架构上的移植方案，以及 基于 AIA 的中断直通设计，最终在社区进行技术分享。

b. 基于上述移植与中断直通方案，在支持 AIA 的 RISC-V 硬件环境中部署 openEuler Embedded（含 Jailhouse 与中断直通），并开展性能测试与评估。形成完整的实验数据与优化建议。

1.2 关键概念、

1.2.1 混合关键系统（MCS）

在混合关键系统（MCS）中，同一硬件平台上会同时承载具有不同关键级别的多种任务。高关键任务（如实时控制）对隔离和确定性要求极高，低关键任务（如 Linux）则更关注功能丰富性和易用性。通过静态分区保证两者互不干扰。

1.2.2 静态分区 Hypervisor（本项目中为 Jailhouse）

Jailhouse 属于 Type-1 Hypervisor，这意味着直接运行在裸机之上，无需依赖宿主操作系统调度，从而大幅降低中间层延迟；另外，它通过在启动时会为每个 Guest OS 预先划分“cell”资源单元，将 CPU 核、内存区域和设备中断直接归属到各自隔离域，实现了高效的静态分区隔离；同时，Jailhouse 在 arch 和 platform 层提供了统一的硬件兼容层，封装了不同架构下的特权指令、MMU 配置和中断控制逻辑，极大地简化了跨平台移植工作。

另外，Bao Hypervisor 拥有与 Jailhouse 类似的静态分区在 RISC-V 平台下的静态分区实现与 M-mode 启动方案，需要被重点参考以指导 Jailhouse 的移植与优化。

1.2.3 RISC-V H 拓展（Hypervisor Extension）

RISC-V 指令集新增 H 模式，定义了 Supervisor（S）态与 Hypervisor（HS）态之间的切换、虚拟化页表以及 Trap 向量重定位等机制，为 Hypervisor 提供原生硬件支持。

1.2.4 Advanced Interrupt Architecture（AIA）

AIA 是 RISC-V 提出的下一代中断规范，引入中断控制器的多级优先级排序、虚拟中断门与硬件中断门映射，使得 Hypervisor 可在不经过软件转发的前提下，将中断直通给 Guest OS，大幅提升中断响应速度。

2. 项目需求分析

2.1 项目目标

2.1.1 完成 jailhouse 设计分析与 RISC-V/AIA 中断直通方案调研

本阶段聚焦产出完整且高质量的文档，确保可用于开源社区进行分享。文档可聚焦以下两个重点。

架构分析相关 — 对比 ARM、x86 与 RISC-V 上 jailhouse 的静态分区模型与硬件兼容层职责，深入剖析关键寄存器（如 mstatus、htinst 等）及 HS ↔ S ↔ U 特权切换流程；配以架构图、时序图与流程图，清晰呈现各模块调用关系与控制流。

RISC-V 移植方案相关 — 基于 H 扩展与 AIA 规范，阐述 CPU 虚拟化（CSR Trap & Emulation）、二级页表与 PMP 保护策略，以及 AIA 中断直通的路由表设计与接口示例；附带伪代码与 UML 图，确保方案可读、可评审、可落地。

2.1.2 基于移植方案的 openEuler embedded 部署与性能测试

在完成设计与方案文档后，将在典型环境（至少是类似与 QEMU-RISC-V 的模拟器，也可以争取尝试真实 RISC-V 板卡）上搭建测试平台，依据前期方案对 jailhouse 进行编译、配置与镜像制作，并部署 openEuler embedded 管理域与轻量级实时域。功能验证阶段将通过示例测试用例（如中断直通的端到端触发流程、特权模式切换的正确性检查等）来检验隔离与直通机制的稳定性；性能评估阶段则可选取典型指标（例如中断延迟、上下文切换开销或基本 I/O 吞吐），利用现有性能分析工具收集数据，以供后续迭代优化。

具体的测试场景、工具与指标也将在实施阶段与导师随时汇报，沟通并及时补充。最终成果包括性能测试报告，其中将至少汇总测试方法、关键数据和对比曲线等内容，并附带所有补丁、脚本与文档，以便项目验收与社区复用。

2.2 重难点分析与思路拆解

2.2.1 RISC-V 虚拟化架构理解与适配

技术挑战： RISC-V H 扩展引入了全新的特权级架构，hypervisor 需要运行在 HS 态，guest OS 运行在 VS 态。相比 ARM/x86 的成熟虚拟化方案，RISC-V 的虚拟化支持涉及大量新增的 CSR 寄存器和指令集扩展，这对移植工作提出了前所未有的挑战。

关键技术要素： 特权级切换机制是首要挑战，需要深入理解 M 态、HS 态、VS 态之间的切换流程和权限管理体系。关键 CSR 寄存器如 hstatus、hedeleg、hideleg、hcounteren 等 hypervisor 相关寄存器的正确配置直接影响虚拟化系统的稳定性和安全性。同时，两阶段地址翻译机制需要基于 hgatp 寄存器实现 guest 物理地址到 host 物理地址的高效翻译，这是内存虚拟化的核心所在。此外，虚拟化异常处理机制需要通过配置异常委托，确保 hypervisor 能够正确拦截和处理 guest 产生的各类异常，实现完整的虚拟化控制。

解决思路： 采用系统性分析 RISC-V Privileged Architecture 规范的方法，重点研究 Volume II 中 Hypervisor Extension 章节，深入理解每个新增特性的设计意图和实现机制。通过对比 ARM 的 Stage-2 页表机制和 x86 的 EPT 技术，借鉴成熟架构的设计思想，结合 RISC-V 的架构特点，设计出既符合 RISC-V 规范又保持高性能的内存虚拟化方案。

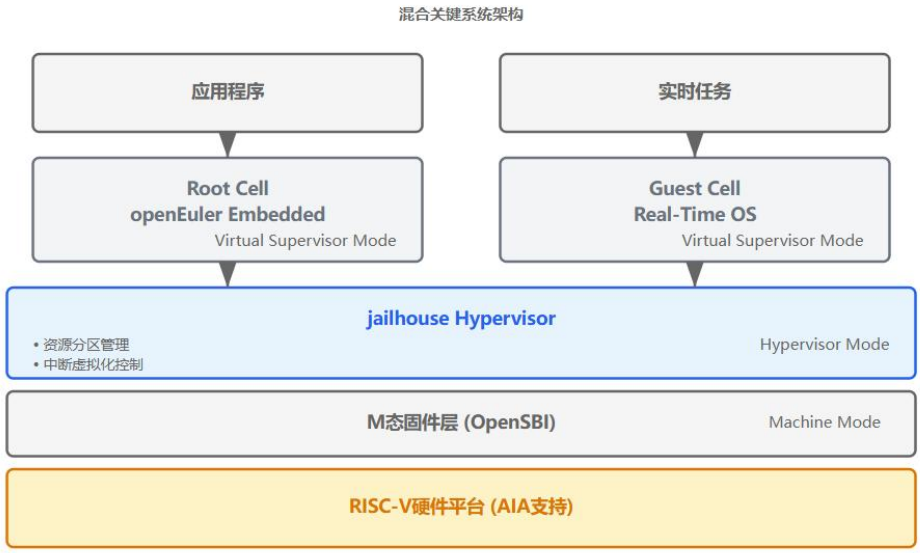

图 1：RISC-V 虚拟化特权级架构图

2.2.2 jailhouse 架构移植的双层抽象

技术挑战：jailhouse 采用 arch 和 platform 两层硬件抽象的优雅设计，在保持代码结构清晰的同时实现了良好的可移植性。RISC-V 架构的独特性要求我们在保持其轻量级设计理念的前提下，完成深度的架构适配工作。

Arch 层移植重点：处理器初始化是 arch 层的核心工作，需要实现 RISC-V 特定的 hypervisor 模式启动序列和关键配置流程。内存管理抽象层面需要适配 RISC-V 的 Sv39/Sv48 页表格式，并实现完整的两阶段地址翻译机制。异常处理框架的建立则需要深入理解 RISC-V 的 trap handling 机制，建立与 jailhouse 现有异常处理模型的有效映射关系，确保系统的稳定性和可调试性。

Platform 层移植重点：RISC-V SBI（Supervisor Binary Interface）接口的处理会是 platform 层的关键挑战，需要妥善处理与 M 态 firmware 的交互，特别是时钟管理、处理器间中断（IPI）等关键系统调用。

解决思路：通过深入分析 jailhouse 在 ARM64 平台的成熟实现，识别其中可复用的抽象层设计和代码结构。同时参考 Bao hypervisor 在 RISC-V 平台的实现经验，特别是其处理 M 态引导和 HS 态切换的机制，设计出既兼容 jailhouse 现有框架又充分发挥 RISC-V 架构优势的移植方案。

2.2.3 基于 AIA 的中断虚拟化方案

技术挑战： RISC-V AIA 代表了中断架构设计的最新发展方向，引入了 IMSIC 和 APLIC 等先进组件。相比传统 PLIC，AIA 在可扩展性、性能和功能丰富性方面都有显著提升，但这也带来了更高的实现复杂性。

核心技术方案： MSI 中断直通是 AIA 虚拟化的核心优势，需要充分利用 IMSIC 的硬件虚拟化能力，实现 guest OS 直接接收 MSI 中断，从而大幅减少中断处理的延迟。中断隔离机制的设计必须确保不同 guest 之间的中断资源完全隔离，防止恶意或错误的 guest 影响其他虚拟机的正常运行。性能优化方面，目标是最小化 hypervisor 在关键中断处理路径上的介入，通过硬件直通技术实现接近原生性能的中断响应。

实现策略： 首先深入分析 AIA 规范中针对虚拟化场景的硬件支持特性，设计基于静态分区的中断资源管理方案，这与 jailhouse 的设计理念高度契合。其次实现从硬件中断控制器到 guest OS 的零拷贝直通路径，减少不必要的数据复制和上下文切换开销。最后，保持与传统 PLIC 的向后兼容性，确保移植方案能够在不同代际的 RISC-V 平台上稳定运行，扩大应用范围。

2.3 可能的切入方式

2.3.1 自底向上的渐进式移植策略

项目采用分阶段的渐进式开发模式，确保每个阶段都有明确的里程碑和可验证的成果。第一阶段重点构建最小化的 RISC-V hypervisor 框架，实现核心的特权级切换、异常处理和基础的 cell 管理功能。第二阶段攻克内存虚拟化核心技术，实现基于 Sv39 的两阶段地址翻译和完整的内存隔离机制。第三阶段完成中断系统集成，从传统 PLIC 兼容开始，逐步引入 AIA 的先进特性，最终实现高性能的 MSI 中断直通和完整的中断隔离机制。

图 2：jailhouse 架构抽象层次图

2.3.2 对上适配：管理系统集成

jailhouse 依赖于运行在 root cell 中的 Linux 内核模块实现管理功能，移植工作需要将驱动模块完整适配到 RISC-V 平台，确保控制接口和监控功能正常工作。同时需要适配 openEuler embedded 的构建系统，实现用户空间工具与 hypervisor 的可靠通信。为保持易用性，移植方案将保持与现有 jailhouse 管理工具的高度兼容性，并针对 RISC-V 平台特性实现专门的配置、监控和调试功能。

2.3.3 对下适配：硬件抽象层构建

通过深入分析 jailhouse 在 ARM64 和 x86 平台的实现，提取可复用的设计模式，建立 RISC-V 特定的硬件抽象接口。架构层面需要实现 CPU、内存、中断三大虚拟化子系统的完整支持。平台层面需要设计灵活的设备驱动框架，支持不同 RISC-V SoC 平台，实现基于设备树的硬件发现机制，并建立与 OpenSBI 等标准固件的兼容接口。

2.2.4 RISC-V 平台“运行时启用”模式的挑战与探索

核心挑战：Jailhouse 在 x86/ARM 平台采用独特的“运行时启用”模型，即由已启动的 Linux 系统加载内核模块，进而启用 Hypervisor 并将 Linux 自身“降级”并迁移至虚拟机中。

然而，这一模式与当前 RISC-V 虚拟化生态的普遍实践形成了鲜明对比。目前，几乎所有公开的 RISC-V Hypervisor（如 Bao）均遵循“固件优先”（Firmware-First）的启动模式，而 Jailhouse 这种“运行时启用”（Runtime-Enablement）的方案在 RISC-V 上尚无经过验证的先例。 这背后的根本原因，是 RISC-V 严格的 M -> HS -> S 特权级架构，它天然禁止了运行在 S-mode 的 Linux 内核主动获取更高权限的 HS-mode。

因此，直接复制 x86/ARM 的实现路径是行不通的。我们经过深入分析，确定解决方案的关键在于利用拥有最高权限的 M-mode 固件（OpenSBI）作为安全的“仲裁者”。

解决思路：模仿 ARM64 架构通过 HVC (Hypervisor Call) 指令从内核态（EL1）陷入 Hypervisor 态（EL2）的成熟模式，在 RISC-V 上设计一套等效机制。即，运行在 S-mode 的 Jailhouse 内核模块将通过一次特定的 ecall（环境调用）请求 M-mode 的帮助，由 M-mode 安全地完成硬件状态切换，并将执行权限“交棒”给位于 HS-mode 的 Jailhouse Hypervisor。这套基于 ecall 的安全跳板机制，是本次移植工作中至关重要的架构创新。

A. S-mode 侧：jailhouse.ko 内核模块的改造

源码参考：将参考 arch/arm64/ 目录下的实现，特别是 arch/arm64/setup.c 中的 jailhouse_enable 等函数。

实现逻辑 (drivers/jailhouse/main-riscv.c)：我们将创建新的 main-riscv.c 文件，其核心函数 jailhouse_enable_riscv() 的执行步骤如下：

1.停转其他 CPU 核心:

思考：在启用 Hypervisor 之前，必须确保只有一个 CPU 核心（引导核）在活动。

实现：可以调用 Linux 内核的 cpu_down() 接口，安全地将所有非引导核置于离线状态。这部分逻辑将直接借鉴 drivers/jailhouse/main.c 中的 jailhouse_enable() 对 cpu_hotplug_disable() 和 smp_call_function_single() 的使用。

2.准备 Hypervisor 内存空间:

思考：Jailhouse 核心代码需要被加载到一个特定的、对 Linux 透明的物理内存区域。

实现：使用 memblock_alloc() 或类似的内核函数，分配一块物理连续的内存，用于存放 jailhouse.bin。同时，解析设备树中的 jailhouse 节点，获取为 Hypervisor 预留的内存区域信息，确保分配的内存在此区域内。

3.定义并执行 SBI 厂商自定义扩展调用 (Vendor-Specific SBI Extension):

思考: 这是与 M-mode 通信的唯一标准途径。为了在不影响现有标准 SBI 规范的前提下实现我们的功能，最稳妥和标准的方法是利用为特定厂商或实现保留的自定义扩展空间。

实现:根据 RISC-V SBI 规范，0x0A000000 至 0x0AFFFFFF 的范围被指定为“厂商自定义扩展”使用。我们无需向社区申请，而是可以直接在此范围内为本项目的实现选取一个 ID（例如，选取一个与 openEuler 或 Jailhouse 相关的特定值，如 0x0A00004A，J for Jailhouse）。这完全符合 SBI 规范，确保方案的合规与独立性。

后期标准化可能: 在项目成功完成并验证有效性后，我们可以整理成果，向 RISC-V 社区或 OpenEuler SIG 提议，探讨将其推广为通用标准接口的可能性。但在本项目范围内，我们将严格遵循厂商自定义的路径。

伪代码：最终的“跃迁”调用可以实现如下。

```
// 在 jailhouse_enable_riscv() 的末尾
#define SBI_EXT_ID_JAILHOUSE_OE 0x0A00004A // Vendor ID: 'J' for Jailhouse
phys_addr_t entry_pc = jailhouse_phys_addr; // jailhouse.bin 的物理入口

// 这个调用是“单程票”，成功后 M-mode 将直接切换到 HS-mode，不再返回 S-mode
sbi_ecall(SBI_EXT_ID_JAILHOUSE_OE, 0, // 使用厂商自定义的ID
          entry_pc,         // a1: hypervisor 入口地址
          config_phys_addr, // a2: cell 配置文件物理地址
          cpuid_to_hartid(boot_cpu_id), // a3: 当前的 hart id
          0, 0, 0);

// 若返回，则表示失败，必须回滚所有操作（如重新上线 CPU）
panic("Jailhouse: SBI call to enable hypervisor failed!");
```

B. M-mode 侧：OpenSBI 固件的功能扩展

源码参考：我们将参考 OpenSBI 项目中 lib/sbi/sbi_ecall_vendor.c 的实现方式，添加我们的自定义扩展。

实现逻辑 (lib/sbi/sbi_jailhouse_ext.c - 新建)：

1.**注册 SBI 扩展：**

在 OpenSBI 的 sbi_platform 接口中注册我们的 sbi_jailhouse_ext 句柄。

2.**实现 jailhouse_go_to_hs_mode 核心函数：**

思考：M-mode 的职责是完成最底层的硬件配置，为 HS-mode 的执行铺平道路。任何遗漏都将导致系统崩溃。

伪代码：基于对 CSR 的直接操作。实现将使用 OpenSBI 提供的 CSR 读写宏。

```
__attribute__((noreturn))
void jailhouse_go_to_hs_mode(unsigned long entry_point,
                             unsigned long config_addr,
                             unsigned long hart_id) {

    // 1. 设置二级页表 (Stage-2 MMU)
    // 在 HS-mode 启动初期，通常先使用一个简单的身份映射（物理地址==虚拟机地址）
    // 来确保 Hypervisor 自身代码可以正常执行。
    // satp 在 HS-mode 下变为 hgatp
    unsigned long hgatp_val = (HATP_MODE_BARE << HGATP_MODE_SHIFT); // BARE mode: no translation
    write_csr(CSR_HGATP, hgatp_val);

    // 2. 配置异常委托 (Crucial Step!)
    // 将所有源于 VS/VU-mode 的、应由 Hypervisor 处理的异常，从 M-mode 委托给 HS-mode。
    // 这包括缺页、虚拟指令等。
    unsigned long hedeleg_val = (1 << CAUSE_ILLEGAL_INSTRUCTION) |
                                (1 << CAUSE_LOAD_PAGE_FAULT) |
                                (1 << CAUSE_STORE_PAGE_FAULT) |
                                (1 << CAUSE_VIRTUAL_INSTRUCTION); // and others...
    write_csr(CSR_HEDELEG, hedeleg_val);
    // 中断委托也同样重要
    write_csr(CSR_HIDELEG, ALL_SUPERVISOR_INTERRUPTS_MASK);

    // 3. 设定 HS-mode 的入口点
    write_csr(CSR_MEPC, entry_point);

    // 4. 修改 mstatus，准备权限切换
    unsigned long mstatus_val = read_csr(CSR_MSTATUS);
    // 清除 mstatus.MPP 位
    mstatus_val &= ~MSTATUS_MPP;
    // 将 mstatus.MPP 设置为 HS-mode (0b10)
    mstatus_val |= (PRV_HS << MSTATUS_MPP_SHIFT);
    // 将 mstatus.MPV (Machine Previous Virtualization) 置 1，表明返回后进入虚拟化模式
    mstatus_val |= MSTATUS_MPV;
    write_csr(CSR_MSTATUS, mstatus_val);

    // 5. 通过寄存器向 Hypervisor 传递参数
    // 根据 RISC-V 调用约定，hart_id 放入 a0，config_addr 放入 a1
    struct sbi_trap_regs *regs = sbi_trap_get_regs();
    regs->a0 = hart_id;
    regs->a1 = config_addr;

    // 6. 执行 mret. CPU 将从 mepc 指定的地址（即 Jailhouse 入口）开始，
    // 在 HS-mode 和虚拟化开启的状态下执行。
    // 这是一个原子操作，此函数将永不返回。也是 Jailhouse on RISC-V 生命周期的起点。
    // * ---- Code below is unreachable ---- */
    // execute_mret_and_jump();

}
```

C. 本方案的优势

“基于 M-mode 协作的运行时启用”方案，不仅解决了 RISC-V 特权级与 Jailhouse 模型的核心矛盾，更在技术选型上展现了深度考量和工程上的成熟度。其优势体现在以下几个层面：

1.架构保真度高，尊重项目哲学：

Jailhouse 的核心设计哲学之一就是其独特的“运行时启用”能力。用户无需专门的引导程序或启动链，即可在标准 Linux 环境下动态地开启和关闭虚拟化。本方案通过 M-mode 作为安全跳板，完美复现了这一核心体验。这保证了 Jailhouse 在 RISC-V 上的行为与其在 x86/ARM 上的行为高度一致，极大地降低了用户的学习理解成本，并保持了工具链的兼容性。

对比替代方案（Hypervisor-First 启动）：另一种看似简单的思路是彻底放弃运行时启用，转而采用类似 Bao Hypervisor 的“固件优先”模式，即由 OpenSBI 直接启动 Jailhouse，再由 Jailhouse 启动 Linux。我在项目推进中预计不会优先实施此方案，因为它将从根本上改变 Jailhouse 的架构，可能使其成为一个“有名无实”的 Jailhouse fork。这将导致与上游社区的实现产生巨大分歧，后续合并补丁、社区协作的成本将很高。

2.实现侵入性最低，兼顾社区协作：

本方案对现有生态系统的修改是精准可控的。在 Linux 侧，它只是一个标准的内核模块；在固件侧，它利用了 SBI 规范中预留的、专为厂商设计的扩展接口。整个方案清晰地界定了 Linux Kernel、OpenSBI 和 Jailhouse 各自的职责边界，没有对任何一方的核心代码进行侵略性的修改。

对比替代方案（改 Linux 内核）：另一个可能的思路是在 Linux 的早期启动流程中（head.S）强行插入跳转到 HS-mode。我不会优先尝试此方案，因为它需要对 Linux 内核的 RISC-V 架构代码进行侵入式修改。这不仅会使工作极度依赖特定的内核版本，难以维护，更不太可能被上游 Linux 社区所接受。

3.风险前置与闭环，具备工程成熟度：

本申请书没有回避项目中最大的技术难点，而是将其作为核心进行剖析，并给出了一个从 S-mode 到 M-mode 再到 HS-mode 的逻辑上完全闭环的解决方案。方案深入到了 CSR 寄存器配置和 ecall 参数传递等具体细节，这表明本人已经完成了充分的预研，将理论风险转化为可执行的工程任务，显著提升项目成功的确定性。

综上，我们选择的方案是一个能在忠于 Jailhouse 核心架构、高度考虑开源社区协作模式、确保技术风险可控这三个维度上取得完美平衡的路径，且具备很高的工程可实现性。

图 3： RISC-V 运行时启用流程图

3. 个人背景与项目契合度论证

3.1 个人背景

3.1.1 教育背景

本人曹俊杰，目前就读于新加坡国立大学电子与计算机工程系攻读硕士学位。在学术研究中，我专注于系统软件及计算机体系结构领域，对底层系统设计与优化具有深厚兴趣。

3.1.2 工业界实习或自主实践经验

NXP Semiconductor（恩智浦半导体）- BSP Software Intern

在恩智浦半导体公司实习期间，我曾负责 Smart-NIC 控制平面的相关开发工作，参与了虚拟化全栈技术的实现与性能优化。这一经历让我对 Type-2 hypervisor 的架构设计、性能瓶颈以及硬件加速机制有了全面的工程实践认知。

Horizon Robotics(地平线) - 工具链开发实习生

在地平线实习期间，我在编译器部门曾负责 Journey 6E/M 系列自动驾驶 SoC 的预研工作，是部门首个向新一代 NPU 芯片内新增的 SPU（一颗玄铁 C906 RISC-V）部署计算任务的员工。算子移植过程中涉及 RISC-V 上的固件开发（使用 RUST 开发），benchmark 过程中涉及基本的 RISC-V 指令内嵌，这让我对 RISC-V 架构以及多核调度有了基本了解。

Intel - OS Software Engineer

目前在英特尔的实习工作聚焦于 Linux 内核实时性分析，基于 PREEMPT_RT 补丁进行系统 debug 与调优。这一经历强化了我对系统底层机制的理解，特别是中断处理流程的深度认知，这给本项目中基于 AIA 的中断虚拟化实现建立了良好基础。

X86 内核复现项目

我曾参考早期 Linux 0.12 版本，在 QEMU 环境中面向 X86 架构上自主实现了一个 32 位的操作系统内核，该项目涵盖了早期 Linux 的全部组件，确保我对 kernel space 编程以及 QEMU 的使用有了一定掌握。

3.2 与项目契合度论证

基于上述个人背景，我认为自己在多个关键技术领域与本项目需求高度契合：

3.2.1 虚拟化技术基础与架构理解

在 NXP 的 Smart-NIC 项目中，我深度参与了基于 KVM/QEMU 的 Type-2 虚拟化方案，对 CPU 虚拟化及 I/O 虚拟化的核心机制有了全面认知。虽然 jailhouse 作为 Type-1 静态分区 hypervisor 与 KVM 在架构理念上存在差异，但底层的硬件虚拟化原理（如特权级切换、MMU 配置、中断路由）具有相通性。这一经验为我理解 jailhouse 的轻量级设计理念和静态资源分区机制奠定了坚实基础。

3.2.2 RISC-V 架构基本理解及移植类工作的实践经验

在地平线的实习经历中，我是部门首个向 RISC-V SPU（玄铁 C906）部署计算任务的工程师，这一经验让我对 RISC-V 的指令集架构、多核调度机制以及跨架构移植的复杂性有了实际认知。更重要的是，这段经历让我更加熟悉“面向特定架构适配/移植”类形工作的工作模式。

3.2.3 系统底层与中断处理专业能力

目前在 Intel 的工作聚焦于基于 PREEMPT_RT 的 Linux 内核实时性分析，这一经历让我对中断处理流程、内核调度机制以及底层硬件抽象层有了深度理解。考虑到本项目的核心技术挑战之一是基于 RISC-V AIA 实现中断直通，我在实时系统中断延迟分析和优化方面的经验将直接支撑 AIA 中断虚拟化方案的设计与实现。

4. 项目时间安排

项目持续时间为2025 年 7 月 1 日 - 9 月 30 日 ，共 13 周。
时间规划将包含至少约 2 周弹性，确保任务顺利交付。

阶段 I（第 1 - 3 周 | 7.1 - 7.21）：需求分析与方案设计
在前三周内将完成对 jailhouse 在 ARM/x86 与 RISC-V 平台的深度对比分析，并细化 CPU、内存 与 RISC-V AIA 中断直通方案，产出详尽的关键设计文档与架构图。

周 1（7.1 - 7.7）
第一周深入研读 RISC-V H 拓展规范，梳理特权级切换与 MMU 机制，对比 ARM/x86 实现，以明确移植基础。

周 2（7.8 - 7.14）
第二周聚焦 jailhouse 内存虚拟化与硬件兼容层分析，确立 arch 与 platform 两层适配思路，并搭建设计文档框架。

周 3（7.15 - 7.21）
第三周完成 RISC-V AIA 中断架构调研，撰写并评审《中断直通设计方案》初稿，最终定稿以指导后续开发。

阶段 II（第 4 - 7 周 | 7.22 - 8.18）：基础功能移植与环境搭建
在四周内搭建 RISC-V 开发与测试环境，完成 jailhouse CPU 与 内存虚拟化核心功能的移植与基本验证。

周 4（7.22 - 7.28）
搭建交叉编译链、QEMU 仿真及硬件调试平台，并将 jailhouse 源码集成至 RISC-V 工具链，为移植做准备。

周 5（7.29 - 8.4）
实现 jailhouse 在 RISC-V 上的 CPU 虚拟化入口与上下文切换逻辑，并通过单元测试验证 guest 切换功能。

周 6（8.5 - 8.11）
完成内存虚拟化功能移植，设计并实现页表初始化与访存隔离机制，验证多 guest OS 的内存隔离效果。

周 7（8.12 - 8.18）
落地 arch/platform 适配接口，进行端到端基础功能验证，确保「CPU+内存」虚拟化在仿真环境中稳定运行。

阶段 III（第 8 - 11 周 | 8.19 - 9.15）：中断直通与系统集成

在四周内重点实现 AIA 中断捕捉与转发逻辑，并与 openEuler embedded（Buildroot + BusyBox）深度集成，完成端到端功能测试与性能基准。

周 8（8.19 - 8.25）

编码实现 AIA 中断捕捉与转发模块，搭建中断上下文管理，并编写测试用例检验方案正确性。

周 9（8.26 - 9.1）

在 QEMU 仿真和真实硬件上验证中断直通，通过多种外设中断触发测试转发效率，并记录延迟指标。

周 10（9.2 - 9.8）

根据测试结果优化中断调度与转发逻辑，减少响应延迟，更新测试脚本并形成性能基准报告。

周 11（9.9 - 9.15）

将移植后的 jailhouse 与 openEuler embedded 集成，部署管理 VM 与 RTOS guest，完成混合 OS 环境下的端到端测试。

阶段 IV（第 12 - 13 周 | 9.16 - 9.30）：收尾、优化与社区分享

最后两周集中于 bug 修复、性能优化、文档完善与社区分享，其中包含约两周弹性时间用于调试与应急调整。

周 12（9.16 - 9.22）

集中修复各阶段测试暴露的残余 bug，对性能瓶颈进行最后优化，并撰写完整技术文档与最佳实践指南。

周 13（9.23 - 9.30）

整理源码仓库与测试数据，进一步完善社区分享材料。