

设计赛道 proj317 决赛文档

1 目标描述

本团队针对自动驾驶应用场景下，高效用户态线程调度策略设计与系统实现进行研究，其主要包括以下阶段的目标：

1、 调研 ROS2 调度的基本知识和搭建基本的系统开发环境

熟悉 ROS2 架构并搭建了 ROS2 的开发环境；

理解 ROS2 中现有的调度策略和相关实现；

熟悉和测试 ROS2 的单线程执行器和多线程执行器的功能。

2、 高效调度策略设计

针对真实的自动驾驶系统中，其应用的划分方式进行调研，针对每个任务的执行量级、数据传输的量级进行调研。通过调研，我们确定了高效调度策略设计的基本思想：权衡好系统切换的开销导致的端到端延迟的增加和调整优先级之后，端到端延迟优化之间的关系；

提出了 callback 优先级动态调整机制和按照消息堆积的优先级调整策略；

提出了 callback 到线程的绑定策略和线程到 CPU 核心的绑定策略。

3、 用户态线程调度系统实现

callback 线程绑定的功能：实现 callback 绑定到线程、线程绑定到 CPU 核心；

DDS 消息堆积监控功能：实时监控 callback 的 DDS 消息队列，防止消息堆积

基于链感知的 callback 优先级分配和节点划分功能：根据不同链的优先级，让 callback 继承链的优先级，从而保证关键链的时延。

2 比赛题目分析和相关资料调研

用户态线程是实现 parallel loop 和 SIMD 的一个典型途径，已成为系统的基础组件。他自动将数据分成多块，并分发给多个线程并行处理。但现有用户态线程实现，由于负载均衡、调度分发等问题，不能高效地支持复杂的应用场景。本赛题着眼于自动驾驶应用场景下，ROS2 系统中内核与用户态的协同感知。通过前期调研和系统分析 ROS2 用户态线程的调度策略，我们发现，在当前的 ROS2 设计中，导致用户态与内核态不能协同感知的原因，主要有以下两点：（1）对于同一个链中的多个回调函数，将

这些回调函数划分到多个执行器中，不同的划分方式会导致不同的端到端延迟。同时，获取最佳的划分方式是非常的困难的。(2) 在自动驾驶的场景中，事件的处理是以任务链的形式来进行的，在衡量任务的过程中，端到端延迟是一个重要的指标，端到端延迟越小，这个事件就能得到更快的处理。在实际设计中，自动驾驶公司更希望某些事件发生的时候，能得到更优先级的处理。然而，在 ROS 2 本身的机制中，并不支持设置链的优先级，这和应用过程中存在不兼容。

针对以上的两个关键问题，我们进行了详细的资料调研。目前的 ROS2 中的线程调度策略的研究主要集中于六个方面：可配置内核调度、用户级调度、中间件调度、并行系统调度、用户级系统调度、分层调度。

可配置内核调度是指在内核中提供灵活的机制，使用户能够根据具体需求自定义和扩展调度策略。[1]提出了一个针对软实时的保证系统，该系统能针对分层调度提供了确定性的保障，其中每个层次的分层调度算法支持任意调度算法。[2]通过可插拔策略的 API 来定制内核的调度策略。尽管这些设计体现了调度策略定制的高效性，但是这些调度策略都是在内核空间中实现的，没有考虑到与应用程序进行紧密的协同设计。

Enoki[3]是一个用于快速开发 Linux 内核调度器的框架，支持将新的调度策略实时升级到内核中，支持用户空间调试，并且与应用程序之间具有双向通信功能，实现了跨应用程序和调度器的细粒度核心共享。

Plugsched[4]以模块化的思想，将调度器与 Linux 内核解耦为一个独立的模块，并通过使用数据重建技术将状态从旧调度器迁移到新调度器。该方案可以直接应用于生产环境中的 Linux 内核调度器，而无需修改内核代码。[5]提出了一个 Linux 内核调度扩展的概念验证，该扩展设计为用户监控线程（UMT），允许用户空间进程在所选线程被阻塞或解除阻塞时收到通知，从而使运行时可以在空闲核心上安排额外的工作。[6]提出了一种内存感知的公平份额调度算法，该调度方法精心将运行任务的 CPU 周期与真正的内存相关停滞分离开来，并对任务进行补偿，使其在为时已晚之前获得所需的 CPU 份额。所提出的方法是自适应、有效且高效的，不依赖于任何静态分配或内存硬件资源的分区，并且仅具有可以忽略不计的运行时开销就能提高 QoS 应用程序的性能。此外，该调度设计是一种仅基于软件的解决方案，仅需对内核进行最小的修改，就可轻松地集成到内核调度器中。

用户级调度是指调度和管理线程的机制在用户空间（应用程序级别）中实现，而不是在内核空间中实现。例如，[7]在进程内，使用并发管理来管理保护域的线程。用户级调度支持高效的线程调度，但是需要考虑阻塞系统调用并与阻塞系统调用集成。[8]中定义了一个基于阻塞系统调用使用动态绑定内核到用户级线程的协议，然而，该协议需要在内核级和用户级线程库之间进行复杂的交互，增加了其复杂性。用户级的调度只能在

保护域内起作用，这些方法不能调度其他保护域中的线程。同时，操作系统内核并不直接感知或控制这些用户级线程，而是将它们视为单一的进程来管理。特别是遇到阻塞操作时，无法利用内核的调度机制进行线程切换，可能导致资源浪费。传统的线程调度时，会为应用的每个请求创建一个内核线程，将应用进程在 CPU 上进行调度，这样导致系统资源利用率低下，而且会造成应用程序之间的资源的竞争。[9]实现了一个用户级线程调度方案（Arachne），它能依据负载对应用的线程进行 CPU 核心的调度，让应用程序感知到所需的 CPU 核心数，并独占核心。同时，为每个核心创建的一个内核进程，消除与其他应用的资源竞争。通过对应用程序的线程依据负载在 CPU 核心上进行调度，以优化应用的性能。ghOSt[10]基于 Linux 内核之上构建了高性能的框架，该框架提供了丰富的 API，可以从用户空间接收进程的调度决策，并将其作为事务执行。从而实现了将线程调度策略委派给用户空间进程，并进行了相关的策略优化、无中断升级和故障隔离。LibPreemptible[11]是一个灵活、轻量级且可扩展的抢占式用户级线程库。通过提供一种用于传递定时中断的快速轻量级硬件机制、通用用户级调度接口和为可供用户开发使用的 API，该用户级线程库实现了根据应用程序需求进行自适应调度的调度策略。

中间件调度是一种调度机制，它通过利用内核 API 和优化调度策略，在内核和用户级别进行线程的优先级管理和资源调度，旨在在不同的保护域之间有效分配计算资源，处理调度阻塞，并简化用户级调度器的设计。[12]和[13]通过显式分配优先级，以便只有单个目标线程具有最高优先级。通过巧妙地利用现有的内核策略和抽象，这使得这种方法即使在调度程序的保护域之外也能对线程进行调度。然而，这种方式存在以下缺陷：

（1）不仅需要设计到内核的调度器，还存在双别内核上下文切换（用户态切换到内核态线程，内核态切换新的用户级的线程）和用于优先级管理的 API 的开销。（2）当线程进行系统调用时发生阻塞，唤醒该线程的时间不确定，这给调度器准确跟踪线程的状态带来挑战。[14]提出了一种两级调度机制，其中内核负责为应用程序分配处理器，而应用程序则负责自己的线程调度。这种机制允许应用程序在内核层面和用户空间层面之间进行交互，以更有效地管理线程的执行。

并行系统调度是指在多核系统中有效管理和调度并行任务的过程，以最大化地利用多核处理器的性能。并行系统调度的主要目标是提升系统的并行性和同步能力，同时保证高效利用计算资源，避免任务间的阻塞和死锁。实时系统中，实时系统中主要适用 OpenMP 来提高程序的运行时的并行性能。在这种运行时中，[15]，[16]，[17]为实时处理调度并行任务提出了一系列独特的挑战。这是由于，为了最大限度地利用大量内核，并行运行时将计算分解为细粒度的任务，这些任务非抢先执行，默认情况下在较少数量的系统线程上运行至完成。它们通常在任务执行延迟其进度的

操作（例如，等待另一个任务完成）时定义调度点，此时运行库在同一线程上运行挂起的任务。OpenMP 存在以下的特征，使系统在分析性地满足截止日期的同时保持高利用率的能力变得复杂且困难：（1）调度任务有效阻塞的点，（2）线程上下文被重用以执行挂起的任务的点，（3）对任务内共享数据结构的访问使用临界段原语，以及（4）任务非抢先执行。如果拥有关键部分的任务到达调度点，并且同一个线程执行试图进入相同关键部分的任务，则很容易发生死锁。OpenMP 使用运行时提供的“avoidance”来解决这个死锁。为了避免这种死锁，默认情况下，所有任务都是绑定任务[18]，[19]，这些任务限制运行时执行可能试图占用如此关键部分的任务，然而，这种情形下容易触发任务执行的最坏情况。在[20]中，提出了一种适用于在分层架构上执行具有不规则和大规模嵌套并行性的 OpenMP 程序的线程调度策略。该线程调度策略强制执行线程分配，最大化属于同一并行区域的线程，并在需要负载平衡时使用了一个 NUMA 感知的工作策略。

用户级系统调度允许用户级别对跨多个保护域的系统级别线程进行控制的调度机制。[21] 开发了一种用于 L4 微内核的层次化用户级调度架构，该架构将调度的功能转移到用户级，以提高调度的灵活性，当内核遇到调度不明确的情况时，引入用户决策。[22] 介绍了 Composite 组件系统中的用户级调度层次设计，其动机是为了创建一个既可靠又可预测的系统，并能根据特定应用需求进行配置。其设计目标是允许不受信任的开发者在保护域内安全地开发服务和策略，同时确保用户空间服务的时间控制，及时处理异步事件。其关键特性包括用户定义的调度策略、高效的异步事件处理和上行调用机制，以及对共享数据结构访问的同步管理。[23] 介绍了一种名为 CPU 继承调度的新型处理器调度框架，在该框架中任意线程可以作为其他线程的调度器，支持在一个系统中实现多种不同的调度策略，从而提供更大的调度灵活性。该框架允许模块化和分层控制处理器在不同管理域（如进程、作业、用户和组）中的使用，并准确地分配和记录 CPU 资源。应用程序和操作系统可以实现定制的本地调度策略，并通过框架确保所有不同策略的逻辑性和可预测性。作为附带效果，框架还通过提供一般形式的优先级继承机制来解决优先级反转问题。CPU 继承调度自然地扩展到多处理器，支持处理器管理技术，如处理器亲和性和调度器激活，并且可以在典型环境中以可接受的开销提供这种灵活性。这些调度系统[21]，[22]，[23]是为系统级线程的用户级控制而设计的（即跨多个保护域）。它们支持线程阻塞和唤醒事件向调度器的矢量化[21]，[22]。然而，这种方法需要线程调度的内核的介入，这也会带来额外的开销。

分层调度通过在用户级实现结构化的调度层次，允许在子系统中对中断和传统线程调度进行管理。该机制支持在每个子系统内构建独立的调度策略，同时保证子系统与内核之间的隔离，从而提供低成本、可靠且可预

测（可扩展）的系统。[21]，[23]都提出了实现分层结构的用户级调度器的机制。此外，[13]认为用户级调度对实时系统是有用的，并提供了在中间件设置中适应它的方法。这些工作都没有试图从内核中删除所有阻塞和调度的概念。此外，这些方法没有提供在不求助于昂贵的调度器调用的情况下调度和记帐异步事件（例如，中断）的机制。[24]探讨了如何在调度器的层次结构中提供组合硬实时保证的问题。该论文绍了一种组合方法，该方法通过子调度器的时间需求推导出父调度器的时间需求，确保只有在子调度器的时间需求得到满足时，父调度器的时间需求也能得到满足。[25]提出了一种运行时算法，使得即使在对手完全控制时间的情况下，各分区也无法感知到其他分区的行为变化。这种方法支持使用动态时间分区机制，既提高了响应能力，又保证了算法级别的非干扰性，与静态方法的效果相当。时间可预测性是成功通过隐蔽时间信道进行通信的前提条件。实时系统特别容易受到时间信道的影响，因为实时应用程序由于时间分区的不确定性受到限制，往往会具有时间局部性。在[26]中，展示了即使在时间分区之间严格执行时间隔离的情况下，实时应用程序也可以创建隐藏的信息流。并且引入了一种在线算法，该算法对时间分区的时间表进行随机化，以减少时间局部性，同时保证时间分区的可调度性和时间隔离。

与机器人操作系统（ROS）相关的研究工作主要集中在提高 ROS 的实时性能上。例如，Wei[27]等人首次在多核处理器上提出一种名为 RT-RTOS 的实时 ROS 架构，RT-RTOS 提供一种集成的实时/非实时任务执行环境，使得实时和非实时 ROS 节点分别运行在不同处理器内核的实时操作系统和 Linux 上。Saito[28]等人为 ROS 提出一个名为 ROSCH 的实时调度框架，能够包含 ROS 本不支持的三个功能，分别是同步系统、基于固定优先级的有向无环图（DAG）调度框架以及故障安全功能。Suzuki[29]等人设计并实现一个可加载的内核模块框架，称为透明 CPU/GPU 协调机制上的实时 ROS 扩展，即 ROSCH-G，用于在异构环境中调度 ROS，从而无需修改操作系统内核和设备驱动程序。Saito[30]等人为 ROS 设计一种基于优先级的信息传输机制，以减少高优先级的 ROS 节点的执行时间和时间差异。不过，这些研究仅适用于早期版本的机器人操作系统，而且并未提供保证实时时序约束的分析方法。

自从 2017 年 ROS 2 发布以来，许多研究者针对 ROS 2 做出大量研究，大多数研究工作的目的在于评估和改进 ROS 2 的实时性能。例如，Blaß T[31]等人首次对 ROS 2 执行器进行正式的形式化建模和分析，对 ROS 2 应用程序的端到端延迟进行约束，为后续分析奠定基础。随后的工作，Casini[32]等人提出一种新颖的针对 ROS 2 任务链的响应时间分析，既考虑机器人工作负载中通常遇到的高执行时间方差，又考虑默认 ROS 2 回调调度程序的饥饿自由度。Yang[33]等人使用被称为回调组执行器的实

时执行器替换 ROS 2 中的标准 RCLCPP 执行器，并探讨与比较由此产生的实时性能。

ROS 2 内部采用两种标准执行器：单线程执行器和多线程执行器。Tang[34]等人改进响应时间分析技术，通过形式建模对单线程执行器进行响应时间分析，并提出优化响应时间的优先级分配策略。Jiang[35]等人研究 ROS 2 中多线程执行器的实时调度和分析，并提出多线程执行器调度模型的形式化描述，从而在多线程执行器上开发任务链执行的响应时间分析技术。Choi [36]等人在最近的研究中，使用单线程执行器时，为 ROS 2 框架提出一种新颖的优先级驱动的任务链感知调度程序，并为所提出的调度程序提供端到端的延迟分析。此外，他们接着提出一个针对在 ROS 2 种多线程执行器上运行的任务链的综合响应时间分析框架[37]。近期，Liu[38]等人为 ROS 2 提出一种名为 RTeX 的新型无锁化的多线程执行器，以提高系统在运行时效率和时序可预测性方面的性能。

3 系统框架设计

3.1 ROS 2 架构介绍

3.1.1 整体架构

ROS2 是一个全新的机器人操作系统，在借鉴 ROS1 成功经验的基础上，对系统架构和软件代码全部进行了重新设计和实现。

立足于系统架构图，如图 1 所示，ROS2 可以划分为以下三层：

操作系统层 (OS Layer)：相较 ROS1 只支持 Linux 来说，ROS2 除了支持主流 PC 操作系统 Windows, Mac, Linux 外，甚至还支持 RTOS，ROS2 让开发者有了更多的选择和可能性。

中间层 (Middleware Layer)：ROS2 的核心层，主要由 DDS 与 ROS2 封装的关于机器人开发的中间件组成。DDS 是一种去中心化的数据通讯方式，采用 DDS 通信，使得 ROS2 的实时性、可靠性和连续性上都有了增强。ROS2 还引入了服务质量 (QoS) 管理机制，借助该机制可以保证在某些较差网络环境下也可以具备良好的通讯效果。ROS2 的中间件则主要由客户端库、DDS 抽象层与进程内通讯 API 构成。

应用层 (Application Layer)：指开发者构建的应用程序，在应用层上，ROS2 几乎继承了 ROS 优秀的设计原理和概念，让用户从使用 ROS 到 ROS2 的过渡更顺利，同时也进行了一些使用方法上的改进：从 python2 到 python3 的编程支持、C++标准更新到 C++11、编译系统的改进 (catkin 到 ament)、多机器人协同通信支持等。

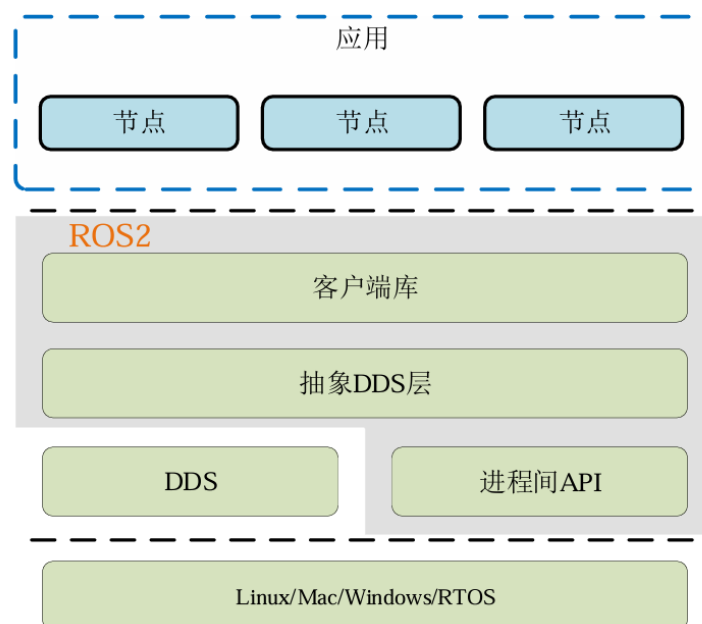


图 1 ROS2 整体架构

3.2.2 rclcpp/rclpy

客户端库是允许用户实现其 ROS 代码的 API。使用客户端库，用户可以访问 ROS2 概念，例如节点、主题、服务等。客户端库采用多种编程语言，因此用户可以使用最适合其应用程序的语言编写 ROS 代码。例如，用 Python 编写可视化工具，以更快地进行原型设计迭代，而对于系统中与效率相关的部分，节点可能更好地用 C++ 实现。

使用不同客户端库编写的节点能够相互共享消息，因为所有客户端库都实现了代码生成器，使用户能够以各自的语言与 ROS 接口文件进行交互。

C++ 客户端库（`rclcpp`）和 Python 客户端库（`rclpy`）都是利用 RCL 中通用功能的客户端库。客户端库不是从头开始实现通用功能，而是使用通用核心 ROS 客户端库（RCL）实现非特定语言的 ROS 概念的逻辑和行为的接口。基于此，客户端库只需要用外部函数接口包装 RCL 中的通用功能。这使得客户端库更轻量化，更易于开发。因为 C 语言通常是调用客户端库最简单的语言，为方便外部函数接口的封装，常见的 RCL 功能通过 C 接口公开。

3.2.3 rcl

RCL 的主要功能是用 C 语言实现了 ROS2 客户端库的抽象接口，RCL 在整个 ROS2 中起到桥接 RMW 和 `rclcpp` 的作用。RCL 是一个比 RMW 更高层次

的 API。

在 RCL 的上层是 `rclcpp`、`rclpy` 它们分别是 RCL 的 C++ 和 python 实现。RCL 接口提供了不限定于语言模式的功能，并且不限定消息类型，避免多次用不同语言实现相同的逻辑和功能，通过重用 RCL 客户端库可以更小，更一致。

RCL 中主要文件包括：`node.c` 定义节点；`publisher.c` 定义发布者；`subscribe.c` 定义订阅者；`client.c` 定义服务客户端；`service.c` 定义服务；`timer.c` 定义计时器；`time.c` 定义 ROS2 时间的概念；`rcl.c` 文件负责 RCL 库的初始化和关闭；`wait.c` 负责等待消息，服务请求，响应，订阅时期准备就绪；`guard_condition.c` 负责境界条件（用来异步唤醒等待节点）；`validate_topic_name.c` 负责验证主题或服务名称的功能；`expan_topic_name.c` 负责 RMW 执行标识符（环境变量）的检查。

3.2.4 rmw

ROS2 在构建中间件解决方案时，采用基于 DDS 的发布订阅系统。由于这些方案有符合 DDS 标准的实现有多种方案，每个方案都有自己的优点和缺点，不同的支持平台、编程语言、性能特征、内存占用、依赖和许可。为了支持多个抽象 API DDS 实现，ROS2 设计了中间件接口 RMW 的细节，实现对不同 DDS 的支持。RMW 中间件接口定义了 ROS 客户端库和任意 DDS 实现之间的 API，从而将通用中间件接口映射到特定中间件的 API 实现。

3.2.5 DDS

DDS 数据分发服务架构是 ROS2 的通信基础，ROS2 的通信模型也是基于 DDS 通信模型，通过对 DDS 实现进行封装，形成 ROS2 的通信网络。DDS 也称为 DDS 中间件，它是一个通信的中间层软件，通过统一的标准规范，不同的企业和厂商可以使用不同的语言，在不同的平台上实现不同的 DDS 软件，而这些软件通过统一的接口又可以互相通信。

DDS 和 ROS 2 在进程间通信方面的交互如图 2 所示。回调通过数据写入器（DataWriter）在主题上发布特定类型的信息，并通过数据读取器（DataReader）接收来自主题的信息。数据写入器和数据读取器由 DDS 创建和注册，为 ROS 2 应用程序提供了与 DDS 交互的途径。DDS 中的一个参与者专门负责管理执行器创建的所有数据写入器和数据读取器的进程间通信。通过各种通信协议（如 TCP/IP、UDP 和共享内存）实现的一组传输通道可用于网络数据传输。默认情况下，UDPv4 通道和共享内存通道分别用于执行程序中的远程和本地进程间通信。

在运行时，默认情况下，一个专用的 DDS 写入器线程负责传输由执行器中的回调发布的消息，该消息在执行器创建的所有数据写入器之间共享，即使这些数据写入器将消息发布到不同的主题。每个 DataWriter 都有一个名为 WriterHistory 的专用 FIFOordered 缓冲区。消息被封装、序列化，然后通过回调写入 WriterHistory，回调进一步选择并通过底层传输协议传输给 writer 线程在主题上订阅的 DataReaders。由于 DDS 支持多对多通信，因此 WriterHistory 中消息的多个副本可以被传递到订阅该主题的不同数据读取器。在 ROS 2 的不同发行版中，从多个 WriterHistories 中选择消息遵循不同的策略，包括 InterestTree 策略（在 ROS 2 Foxy Fitzory 中使用）和 FIFO 策略（在 ROS 2 Humble Hawksbill 中默认使用）。在本文中，我们考虑这两种策略。在订阅端，DDS 监听器线程专用于持续监视信道中的传入消息，这些消息可能是由不同的数据写入器发送的。在 ROS 2 中，默认情况下，两个侦听器线程专用于分别持续监视 UDPv4 通道和共享内存通道，以获取远程和本地传入消息。然后，消息被侦听器线程写入相应 DataReader 的 ReaderHistory 中，等待提取。之后，执行器被告知消息的到达，并激活回调以使用消息执行。通常，DDS 线程(包括写入器线程和侦听器线程)可以在 OS 中的不同调度策略下调度，这取决于关于它们的功能和重要性的特定实现。例如，DDS 线程可以被配置为由 Linux 中的 SCHED_OTHER、SCHED_FIFO 或 SCHED_DEADLINE 调度器来调度。

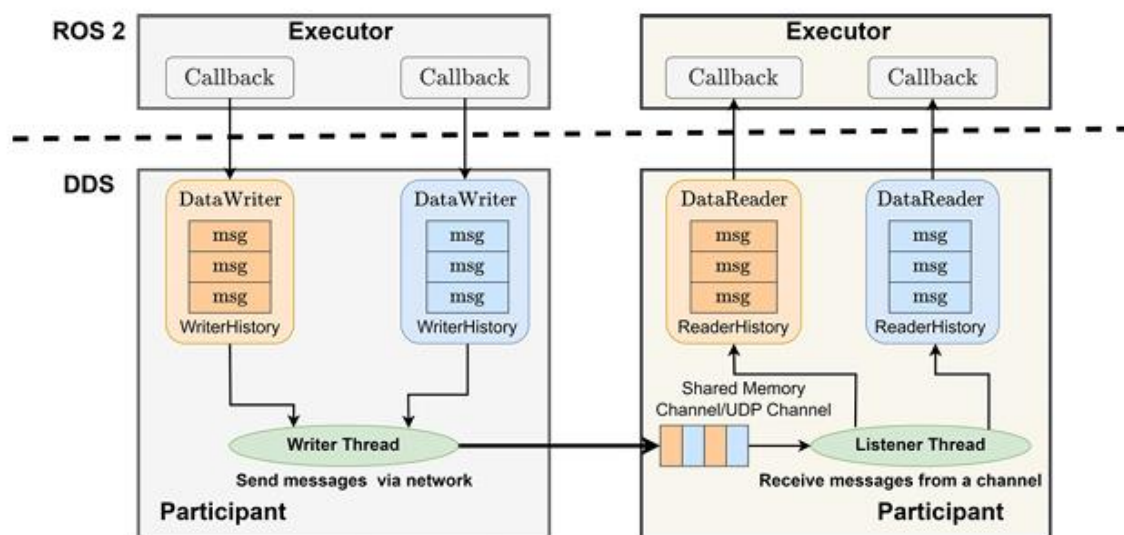


图 2 DDS 进程间通信图[39]

3.2.6 多线程执行器

一个 ROS2 应用是由多个节点（Node）组成。一个节点中有多个回调

函数 (Callback)，一个或多个节点映射到操作系统中的一个进程。ROS2 内置的执行器 (Executor) 用于协调和调度进程中回调。ROS2 提供了 2 种执行器。一种是单线程执行器，在单线程中执行回调；另一种是多线程执行器跨多个线程执行回调。它们通过一个 `spin()` 循环来获取所有在 `waitable` 中准备好的回调。使用 `getNextReadyExecutable` 来获取下一个准备好的回调，并执行准备好的回调。每个线程采用了 RR 调度 (时间片轮转) 策略。定时器是一个特殊的回调，在一个时间窗口内，可以随时添加到可等待集合中。

ROS2 的通信机制有三种：Publish-Subscribe、Service-Client、Actions。执行器 (Single Thread Executor) 根据回调函数的优先级来决定它们的执行顺序。回调函数的优先级取决于：回调的类型和注册顺序 (Registration Order)。回调顺序的优先级为：定时器回调、订阅型回调、服务型回调和客户型回调。

如果 `Readyset` 为空，那么执行器将所有就绪 (ready) 的常规回调放入 `Readyset`。常规回调是除了定时器回调之外的回调。`Readyset` 更新的时间点称为轮询点。两个相邻轮询点之间的时间间隔称为处理窗口 (Processing Window)。定时器回调不需要等待轮询点。首先执行定时器回调，然后是订阅型回调，然后是服务型，最后是客户型。如果某些回调具有相同的类型，我们将根据它们的注册顺序执行它们。更早在执行器上注册的回调则具有更高的优先级。如果回调有多个实例，则只有第一个实例可以执行。如果 `Readyset` 中所有回调的第一个实例都执行了，也就是说 `ReadySet` 是空集，那我们就到达下一个轮询点，新的处理窗口将开始。

如图 3 所示，ROS2 的多线程执行器通过调用执行器实例的 `spin()`，当前线程开始查询 `rcl` 和中间件层以获取传入消息和其他事件，并调用相应的回调函数，直到节点关闭。为了不抵消中间件的 QoS 设置，传入消息不会存储在客户端库层的队列中，而是保留在中间件中，直到回调函数对其进行处理。等待集用于通知执行程序中间件层上的可用消息，每个队列有一个二进制标志。等待集还用于检测计时器何时过期。

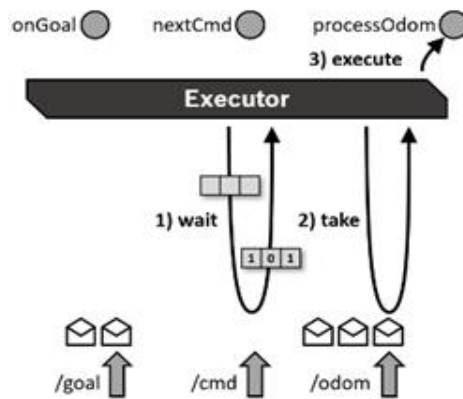


图 3 多线程执行器处理逻辑图

如图 4 所示，多线程执行器包括如下的处理步骤：

step1: `get_next_executable()` 调用 `get_next_subscription()` 来检查当前是否收到消息；

`get_next_subscription` 遍历当前节点的所有 `rcl_subscription_t` 句柄，对每个 `rcl_subscription_t` 句柄检查其所在回调组是否为 `MutuallyExclusive`，如果是 `MutuallyExclusive` 互斥组那么还得确保当前回调没有其他线程正在调用。

Step2: 在 step1 得到一个包含 `subscription` 的 `AnyExecutable` 对象之后，我们执行 `execute_any_executable` 中的 `execute_subscription` 函数，开辟 `message` 空间，操作 `rcl` 和 `rmw` 句柄获取消息，如果 `rmw_take_with_info` 返回 `RMW_RET_OK` 的话表示 `rmw` 有消息，则进行 `handle_message` 步骤（step3）。

Step3: `handle_message` 首先根据模板参数 `CallbackMessageT` 将返回数据转换成对应类型；然后将带类型的消息数据分发给回调函数进行处理。

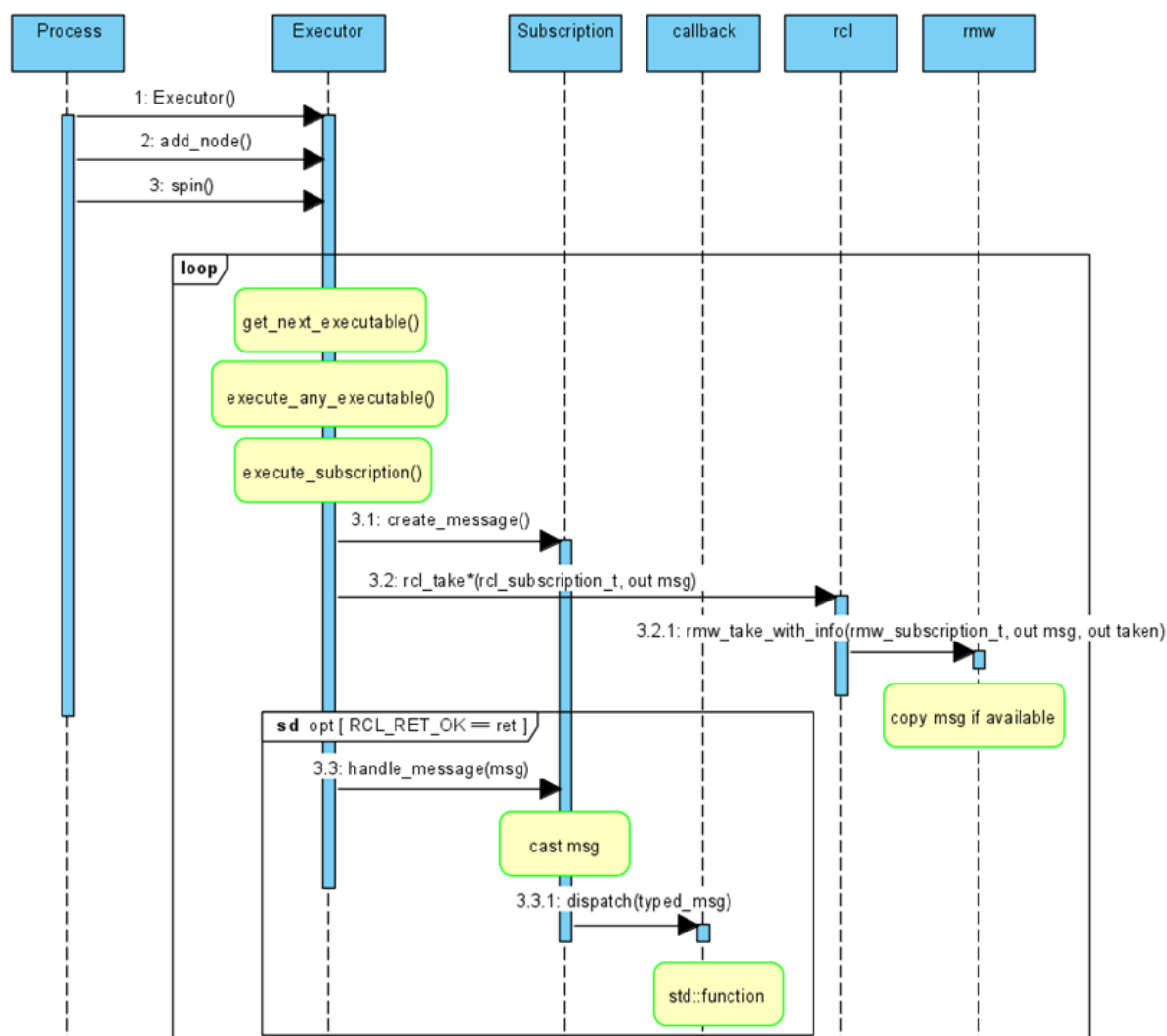


图 4 多线程执行器时序图

我们将应用程序视为一组回调处理链，这些回调必须以特定的顺序交互才能完成它们的任务。例如，在自动驾驶车辆的软件堆栈中，基于激光雷达的感知流水线应用程序可以具有若干数据相关回调链，这些回调链过滤接地点、将剩余点聚类成可检测对象、按感兴趣区域过滤对象、对剩余对象进行特征检测，并最终随时间跟踪对象。图 5 所示为 Apex.AI 开发的 Autoware 参考系统的链配置[40]。来自不同节点的多个链被划分为四个执行器，由图中不同的节点框颜色表示）。

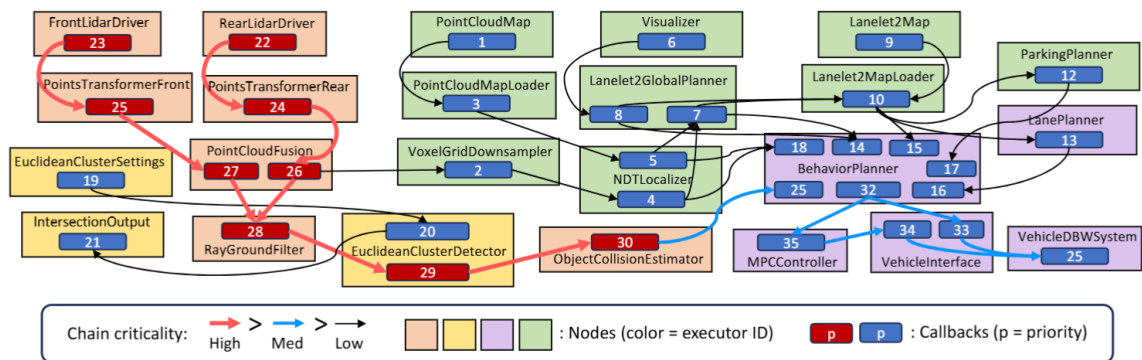


图 5 Autoware 参考系统链配置示意图

3.2 ROS 2 优化设计

3.2.1 callback 线程绑定设计

callback 的线程绑定设计主要分为两个方面，一方面是 callback 的静态绑定，即在 executor 启动之前，设置好静态的绑定。这样在运行过程中，callback 启动之后，该 callback 只能由指定的线程进行处理。另一方面是实现 callback 的动态绑定，在 executor 运行的过程中，动态设置 callback 的归属属性。

● callback 的静态绑定

在 ros2 程序启动前，通过配置文件，设定线程的个数，同时设置 callback 绑定到对应的线程。设置完成之后，待 ros2 启动之后，其 callback 会在对应的线程执行。

callback 的静态绑定执行流程如下：

(1) 初始化阶段：

每个 Callback 实体内部 (Timer、Subscribers 等) 增加一个数据项 `thread_affinity` (默认值为-1)，用来记录该回调实体对应的执行器线程亲和性；

多线程执行器创建完毕后，调用执行器中的重写函数 `set_thread_affinity()`，为每个 callback 实体指定对应的执行器线程亲和性。例如，当执行器创建的线程个数为 4 时，则执行器线程 ID 为 0-3，利用 `set_thread_affinity()` 函数将每个 Callback 实体内部的 `thread_affinity` 数据项修改为 0-3 中的一个。

(2) 运行阶段：

执行器线程按照其 ID 绑定对应 CPU 核心；

一个执行器线程拿到互斥锁，遍历 `wait_set`，`wait_set` 中的就绪回调按照回调函数的隐性优先级排序，即不同类型的回调 Timer 的优先级高于

Subscribers，同一类型的回调先注册进执行器的优先级越高。

当 `wait_set` 不为空时，执行器线程按照隐性优先级遍历就绪回调函数组，找到一个就绪回调后，判断该就绪回调内部的 `thread_affinity` 数据项是否与执行器线程 ID 相等。

如果不相等，执行器线程继续遍历就绪回调函数组，直到遍历结束或者找到一个 `thread_affinity` 数据项与执行器线程 ID 相等的就绪回调，

如果执行器线程从 `wait_set` 中找到一个 `thread_affinity` 数据项与执行器线程 ID 相等的就绪回调，执行器线程将该就绪回调从 `wait_set` 中取出并释放互斥锁，执行器线程执行就绪回调。

如果执行器线程未从 `wait_set` 中未找到一个 `thread_affinity` 数据项与执行器线程 ID 相等的就绪回调，执行器线程重新更新 `wait_set`，即将 `wait_set` 清空后加入目前的就绪回调函数组。

● callback 的动态绑定

修改 `ros2` 的源码，创建一个动态绑定线程的 API。在程序运行的过程中，能通过调用这个 API 将 callback 绑定到对应的线程。绑定的操作直接用静态绑定的 API。

3.2.2 DDS 消息堆积监控机制设计

DDS 和 ROS2 作为独立的框架，两个框架在联合使用的过程中，都没有针对对方进行特定的优化。例如，当 `ros2` 的节点在借助 DDS 传输大量的数据的时候，Callback 的 `DataReader` 缓冲区会填满数据，Callback 会不断的在缓冲区中获取旧时间戳的数据，造成这种现象主要有两个原因，一是 `pub` 节点的数据写入过于频繁，导致 `DataReader` 中缓冲区的数据挤压，二是 `DataReader` 所对应的 callback 被分配到的处理器的时间过小，导致 callback 不能有足够的执行时间。基于此，我们设计了一个在线的 DDS 消息堆积监控机制，该机制能实时监控 `DataReader` 中缓冲区的情况，如果其数据量超过给定的设定的阈值，则向特定的监控进程发送信号，由该监控进程进行处理。

该机制的具体流程如下：

- 数据获取：使用 DDS 提供的接口获取 `DataReader` 的缓冲区状态，包括缓冲区的大小和当前存储的数据量。

- 阈值检测：设定一个阈值，当缓冲区的数据量超过该阈值时，触发

警报。

- 信号发送：当检测到缓冲区超出阈值时，向特定的监控进程发送信号。

- 监控进程：监控进程接收信号并采取相应的处理措施，例如记录日志、调整参数、通知用户等。

3.2.3 基于链感知的 callback 优先级分配和节点划分机制

由于真实的自动驾驶场景需求是以数据处理为核心，我们提出了链感知节点分配方案。我们将节点分配给执行器上，然后将执行器映射到可用的 cpu 核。其核心的思想上尽可能将同一条链分配给同一个 cpu 来最小化链之间的干扰。分配方案是离线完成的，不会引入运行时开销。

输入：执行器数量 (M)、可用 CPU 核数 (P) 和 chain 集 (c)

//将 chain 中的 callback 分配到执行器中

如果有空闲的多线程执行器 e

- 计算该执行器中存在的链的个数 k，存在的线程数 m

- 将未分配最高优先级的 chain 分配到单个多线程执行器 e 中，直到该节点的利用率超过 1.

- 给这条 chain 中的节点 $\max(1, m-k)$ 个线程

- 如果有剩下的 callback 没有分配，则将其分配到剩下的执行器中，同时保证利用率不超过 1

否则，将节点分配到不为空的执行器中

- 计算该执行器中存在的链的个数 k，存在的线程数 m

- 将未分配最高优先级的 chain 分配到单个多线程执行器 e 中，直到该节点的利用率超过 1.

- 给这条 chain 中的节点 $\max(1, m-k)$ 个线程

- 如果有剩下的 callback 没有分配，则将其分配到剩下的执行器中，同时保证利用率不超过 1

//将执行器中的节点分配到 CPU 中

如果有空闲的 cpu P

- 将含有高优先级的链的节点分配到空闲的 cpu，直到该 cpu 的利用率超过 1.

- 将剩下的节点分配给利用率最低的 cpu

如果没有空闲的 cpu，则直接将剩下的节点分配给利用率最低的 cpu。

输出： 分配结果

4 开发计划

时间	开发内容
3 月 23 日-5 月 31 日	对 ROS2 的单线程执行器和多线程执行器的功能进行熟悉和测试
6 月 1 日-7 月 1 日	实现了基于链的 callback 线程绑定设计
7 月 1 日- 7 月 15 日	实现了 dds 消息堆积监控机制
7 月 15 日-7 月 31 日	实现了基于链的 callback 动态优先级调整机制
7 月 31 日-后续	对项目进行进一步对完善，多做一些实验数据

5 系统测试情况

5.1 ros2 执行器的评估测试

ROS 2 支持单线程执行器与多线程执行器，当 ROS 2 系统中注册对应多条任务链的多个回调函数时，回调函数的优先级分配与回调函数组的线程绑定会对任务链的端到端延迟造成干扰，而任务链的端到端延迟是评估系统实时性能的重要衡量因素。本实验的目标在于评估与测试使用单线程执行器、多线程执行器以及多个单线程执行器时任务链端到端延迟的不同，从而研究与分析不同类型执行器对于系统实时性能的影响。

本实验在 ROS 2 系统中注册三条任务链，每条任务链中由三个回调函数组成，如图所示，其中，三条任务链中的九个回调函数分别编号 1 至 9，并且每条任务链的第一个回调函数是由时间触发的定时器，其余两个回调函数是由事件触发的订阅者。

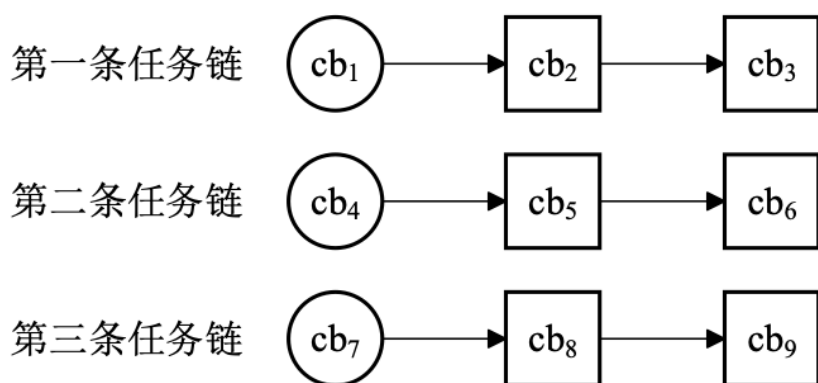


图 6 链示意图

在下列六组配置的情况下，本实验测试三条任务链的端到端延迟的变化情况：

1. 九个回调函数全部加入一个单线程执行器中（SingleThread）；回调顺序是{1, 2, 3, 4, 5, 6, 7, 8, 9}。

2. 九个回调函数全部加入一个多线程执行器中，并且多线程执行器的线程数目为三（MultiThread）；回调顺序是{1, 2, 3, 4, 5, 6, 7, 8, 9}。

3. 将九个回调函数分成三组，回调顺序分别是{1, 2, 3}、{4, 5, 6}与{7, 8, 9}，每个回调函数组加入一个单线程执行器中（MultiExecutor1）；

4. 将九个回调函数分成三组，回调顺序分别是{1, 5, 9}、{2, 6, 7}与{3, 4, 8}，每个回调函数组加入一个单线程执行器中（MultiExecutor2）；

5. 将九个回调函数分成三组，回调顺序分别是{1, 4, 7}、{2, 5, 8}与{3, 6, 9}，每个回调函数组加入一个单线程执行器中（MultiExecutor3）；

6. 将九个回调函数分成三组，分别是{1, 4, 7}、{8, 5, 2}与{9, 6, 3}，每个回调函数组加入一个单线程执行器中（MultiExecutor4）。

本实验是在一台服务器电脑上进行，该桌面 PC 配备 16 核 CPU，运行频率固定为 1.20GHz。每个 CPU 有 16.5MB 缓存，系统配备 62.5GB DRAM。本实验使用 ROS 2 Humble 版本，并采用了默认服务质量（QoS）设置的进程内 API。每个定时器回调函数的执行周期设置成 300ms，每个回调函数的执行时间设置成 100ms，消息大小设置为 4 字节，具体为 Int32，以确保将消息传输的开销降至最低。每个实验的运行时间超过五分钟，为考虑初始化影响，前 500 个样本被丢弃。在不同的配置下，每条任务链的端到端延迟的变化情况如图所示：

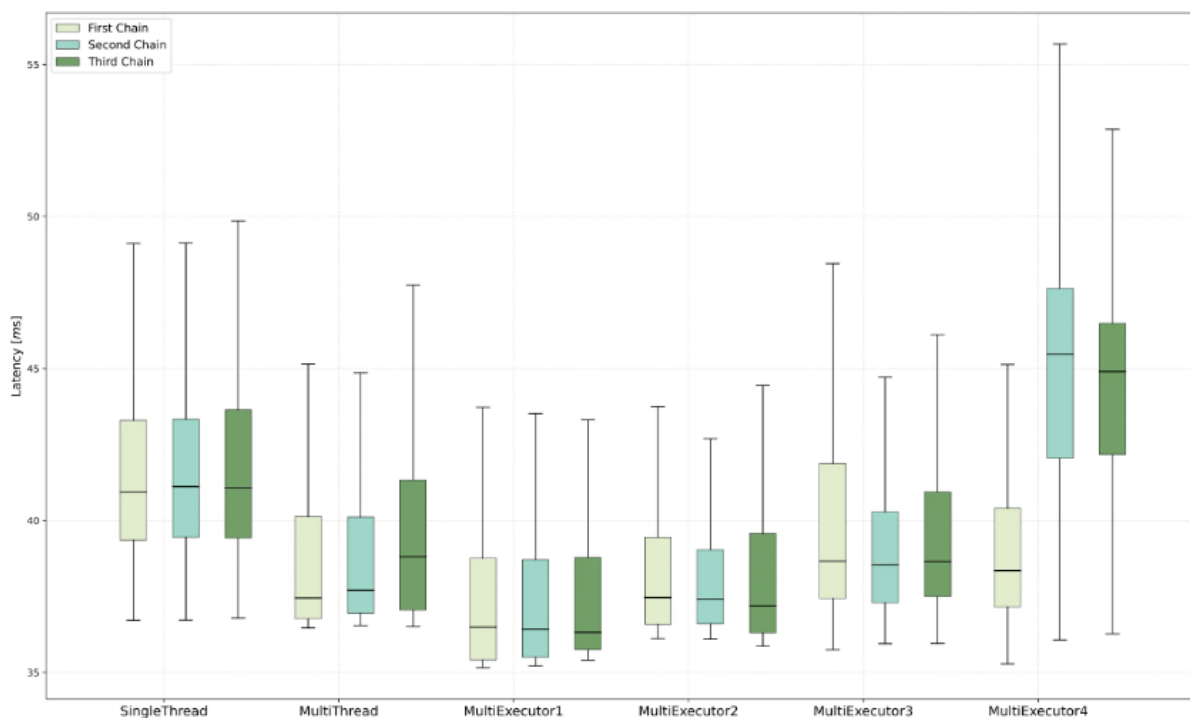


图 7 实验图

在不同的实验配置下，每条任务链的端到端延迟的更详细细节如下表所示，其中，CPU 利用率表示在 ROS 2 系统运行过程中桌面 PC 的每秒 CPU 平均利用率，Switch 表示在 ROS 2 系统运行过程中工作线程在用户态与内核态之间的每秒平均切换次数。

```
pidstat -u process_name -u -w 1
```

每 1 秒钟获取一次 process_name 线程的切换次数和 CPU 利用率。

	Single - Thread ed	Multi- Thread ed	Multi- Execut or1	Multi- Execut or2	Multi- Execut or3	Multi- Execut or4
CPU 利用率	46%	45%	38%	41%	44%	45.50%
Switch	64.85	45.8	0	0	0	0
第一条链平 均值（ms）	41.76	40.27	38.65	40.05	41.1	40.08
第一条链标 准差（ms）	3.87	6.64	6.34	8.35	6.57	5.62
第二条链平 均值（ms）	41.91	40.33	38.41	38.7	39.88	44.79
第二条链标 准差（ms）	3.99	6.71	5.95	3.76	4.55	4.19

第三条链平均值 (ms)	41.96	40.68	38.34	39.58	40.85	44.61
第三条链标准差 (ms)	4.19	5.77	5.8	5.9	6.37	4.41

实验结果显示，使用不同类型的执行器，或者让具有不同优先级的回调函数注册在不同的执行器上运行，对任务链的端到端延迟存在显著影响。

5.2 系统性能评估测试

为了模拟真实世界的系统，我们参考 github 上的开源项目 reference-system 创建一个节点图，Timer 的周期和节点的执行时间都符合真实世界的特征。“reference-system” 是一个示范性的项目，旨在提供一个标准化的系统用于评估和展示 ROS 2 的各种功能和特性。

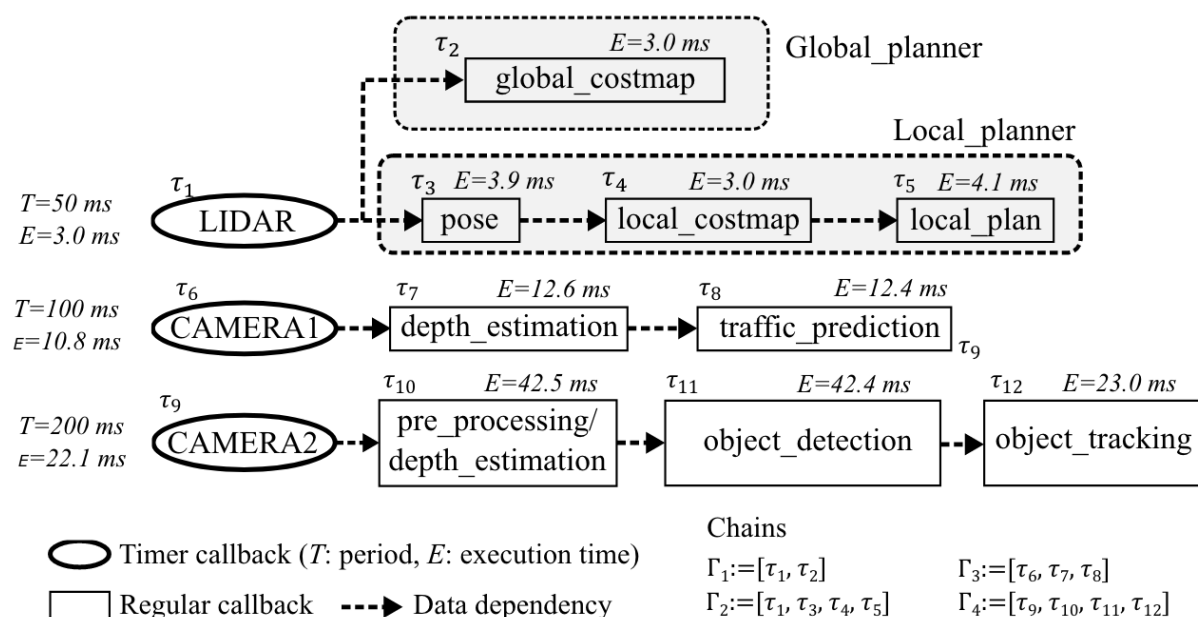


图 8 链示意图

为了评估在 ROS 2 多线程执行器中实现分区和半分区调度的潜在好处并验证实际效用，我们基于图 9 的案例研究进行了一项实验。我们的设置包括在配置有四个线程的多线程执行器上运行一组处理链。对于完全分区调度场景，我们为每个线程分配了三个回调。对于半分区场景，我们为每个计时器回调指定了线程关联，同时允许剩余的回调按照默认的全局调度策略在任何线程上执行。图 9 显示了我们案例研究的处理链配置，其中较低的链索引表示较高的优先级。我们测量每个链的响应时间作为我们的主

要指标，以评估多线程执行器中分区和半分区调度策略的有效性。

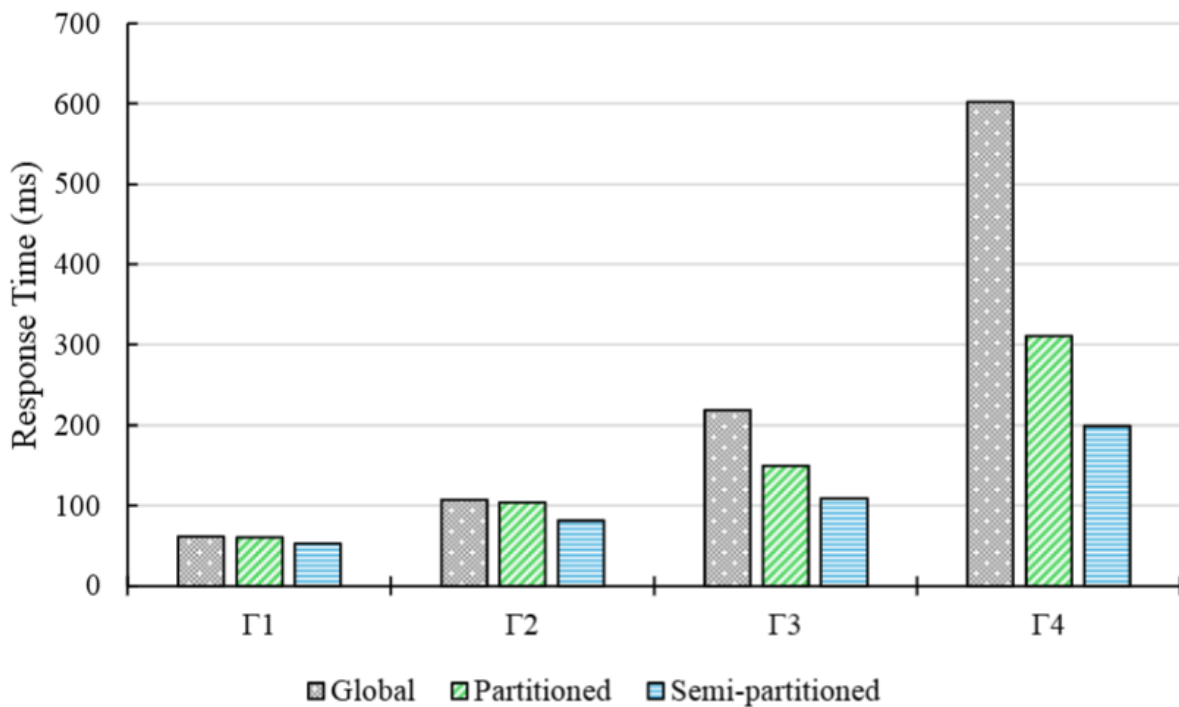


图 9 实验结果图

6 遇到的主要问题和解决方法

问题一： 如何创建一个较好的隔离环境，将系统中的干扰给排除出来。

我们主要从两个方面实现理想的隔离环境，一个是把单独的核心给隔离出来，第二个是对单独的核心固定 cpu 频率。具体的方法如下：

- 步骤一： 设置恒定的 CPU 频率

所有 CPU 的频率都设置为 1.50 GHz

```
# run it as root
```

```
sudo su
```

```
echo -n "setup constant CPU frequency to 1.50 GHz ... "
```

```
# disable ondemand governor
```

```
systemctl disable ondemand
```

```
# set performance governor for all cpus
```

```
echo performance | tee
```

```
/sys/devices/system/cpu/cpu*/cpufreq/scaling_governor >/dev/null
```

```

# set constant frequency
echo 1500000 | tee
/sys/devices/system/cpu/cpu*/cpufreq/scaling_min_freq >/dev/null
echo 1500000 | tee
/sys/devices/system/cpu/cpu*/cpufreq/scaling_max_freq >/dev/null

# reset frequency counters
echo 1 | tee /sys/devices/system/cpu/cpu*/cpufreq/stats/reset >/dev/null

echo done

sleep 1
# get freq info
echo `cpufreq-info | grep stats | cut -d ' ' -f 23-25`
● 步骤二：隔离 CPU
将 CPU 2,3 隔离以运行测试
# modify kernel cmdline
cd ~
dd if=/boot/firmware/boot.scr of=boot.script bs=72 skip=1

# edit boot.script and modify bootargs to
ubuntu@ubuntu:~$ cat boot.script | grep "setenv bootargs" | head -1
setenv bootargs " ${bootargs} rcu_nocbs=2,3 nohz_full=2,3 isolcpus=2,3
irqaffinity=0,1 audit=0 watchdog=0 skew_tick=1 quiet splash"

# generate boot.scr
mkimage -A arm64 -O linux -T script -C none -d boot.script boot.scr

# replace boot.scr
sudo cp boot.scr /boot/firmware/boot.scr

sudo reboot

# check cmdline
ubuntu@ubuntu:~$ cat /proc/cmdline
coherent_pool=1M 8250.nr_uarts=1 snd_bcm2835.enable_compat_alsa=0

```

```
snd_bcm2835.enable_hdmi=1 bcm2708_fb.fbwidth=0 bcm2708_fb.fbheight=0  
bcm2708_fb.fbswap=1 smsc95xx.macaddr=DC:A6:32:2E:5
```

```
4:97 vc_mem.mem_base=0x3ec00000 vc_mem.mem_size=0x40000000  
net.ifnames=0 dwc_otg.lpm_enable=0 console=ttyS0,115200 console=tty1  
root=LABEL=writable rootfstype=ext4 elevator=deadline roo
```

```
twait fixrtc rcu_nocbs=2,3 nohz_full=2,3 isolcpus=2,3 irqaffinity=0,1  
audit=0 watchdog=0 skew_tick=1 quiet splash
```

```
# check interrupts
```

```
# Only the number of interrupts handled by CPU 0,1 increases.
```

```
watch -n1 cat /proc/interrupts
```

```
# check soft interrupts
```

```
watch -n1 cat /proc/softirqs
```

```
# check isolated CPUs
```

```
cat /sys/devices/system/cpu/isolated
```

```
2-3
```

```
cat /sys/devices/system/cpu/present
```

```
0-3
```

```
# run reference system on CPU2
```

```
taskset -c 2
```

```
install/autoware_reference_system/lib/autoware_reference_system/autoware_def  
ault_singlethreaded > /dev/null
```

```
# get pid
```

```
RF_PID=`pidof autoware_default_singlethreaded` && cat  
/proc/$RF_PID/status | grep ^Cpu
```

```
# check how many threads are running
```

```
ps -aL | grep $RF_PID
```

```
3835      3835 ttyS0      00:03:46 autoware_defaul  
3835      3836 ttyS0      00:00:00 autoware_defaul  
3835      3837 ttyS0      00:00:00 autoware_defaul  
3835      3838 ttyS0      00:00:00 autoware_defaul  
3835      3839 ttyS0      00:00:00 gc  
3835      3840 ttyS0      00:00:00 dq.builtins
```

3835	3841 ttyS0	00:00:00 dq.user
3835	3842 ttyS0	00:00:00 tev
3835	3843 ttyS0	00:00:00 recv
3835	3844 ttyS0	00:00:00 recvMC
3835	3845 ttyS0	00:00:00 recvUC
38353846	ttyS0	00:00:00 autoware_defaul

7 分工和协作

梁浩纯：

- 总体架构设计

负责比赛赛题解决方案总体设计框架的规划和确定。

确定系统模块间的接口和交互方式。

协调团队成员间的技术方案，确保系统设计的一致性和完整性。

- 高效调度策略设计

研究并设计高效的任务调度算法，优化系统性能。

负责实现和测试调度算法，确保其在各种使用场景下的稳定性和高效性。

处理调度策略的性能评估和优化，确保系统能在多任务环境下高效运行。

穆元震：

- ROS2 系统环境搭建

负责 ROS2 (Robot Operating System 2) 的环境配置和搭建。

安装和配置必要的开发工具和依赖库，确保开发环境的稳定和高效。

提供 ROS2 开发环境的使用文档和培训，帮助团队其他成员快速上手。

- CPU 核隔离功能实现

研究和实现 CPU 核隔离技术，以提高系统的实时性和性能。

配置操作系统内核，确保特定任务能够独占某些 CPU 核。

进行性能测试和调优，确保 CPU 核隔离功能的有效性。

- 系统总体功能验证与测试

制定详细的测试计划，确保所有系统功能得到充分验证。

设计和实现自动化测试工具，提高测试效率和覆盖率。

负责最终系统的整体测试，确保系统稳定性和性能达到预期目标。

刘峻一：

- 技术文档撰写

负责编写和维护项目的技术文档，包括设计文档、用户手册和 API 文档等。

确保文档内容准确、详细，并易于理解和使用。

定期更新文档，反映项目进展和技术变更。

- 代码组织与管理

负责项目代码的结构化管理，制定代码规范。

使用版本控制系统（Git）进行代码管理。

进行代码审查和合并，确保代码质量和一致性。

8 提交仓库目录和文件描述

OS 决赛代码包括 `intra_process_demo` 和 `rclcpp` 两部分：

`intra_process_demo` 为我们团队所搭建的实验 demo，主要是 5.2 小节所提出的参考系统的实现，以评估本团队提出的设计方案在多个处理链上多线程执行器的性能优化。包括 `include` 文件夹（相关头文件）、`src` 文件夹（源码实现）和 `test`（相关测试用例）。

`rclcpp` 为用户态线程高效调度策略的实现，主要是 `callback` 线程绑定设计、DDS 消息堆积监控机制设计、基于链感知的 `callback` 优先级分配和节点划分机制，包括 `include` 文件夹（相关头文件）、`src` 文件夹（源码实现）。

9 比赛收获

通过学校老师和工业界（华为）老师的指导，我们团队成员之间相互协作，共同完成了本次的赛题。在比赛初期，我们通过搭建 ROS2 系统和阅读相关论文，对比赛的赛题进行进一步的深入理解，提炼出在 ROS2 系统中存在的真实问题，使比赛的赛题与系统设计实现有了更好的结合。在这一过程中，我们的工程实现能力和信息检索和提取能力得到了良好的训练。在之后的高效调度策略设计和系统实现过程中，我们收获了如何从理论上分析问题产生的根本原因和验证所提出的策略的有效性的思考方法，同时也让我们学会了定位问题，复现问题，调试代码的一些工程实践技巧。此外，通过参与这个比赛，也增加了我们团队成员的凝聚力和攻坚克难的信念。

10 参考文献

- [1] J. Regehr and J. A. Stankovic, “HLS: a framework for composing soft real-time schedulers,” in Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420), London, UK: IEEE Comput. Soc, 2001, pp. 3–14. doi: 10.1109/REAL.2001.990591.
- [2] B. B. Brandenburg, “Scheduling and locking in multiprocessor real-time operating systems”.
- [3] Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, and Thomas Anderson. 2024. Enoki: High Velocity Linux Kernel Scheduler Development. In Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24). Association for Computing Machinery, New York, NY, USA, 962–980. <https://doi.org/10.1145/3627703.3629569>
- [4] Teng Ma, Shanpei Chen, Yihao Wu, Erwei Deng, Zhuo Song, Quan Chen, and Minyi Guo. 2023. Efficient Scheduler Live Update for Linux Kernel with Modularization. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 194–207. <https://doi.org/10.1145/3582016.3582054>
- [5] A. Roca, S. Rodríguez, A. Segura, K. Marquet and V. Beltran, "A Linux Kernel Scheduler Extension for Multi-core Systems," 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC), Hyderabad, India, 2019, pp. 353-362, doi: 10.1109/HiPC.2019.00050.
- [6] Kim J, Shin P, Kim M, et al. Memory-aware fair-share scheduling for improved performance isolation in the Linux kernel[J]. IEEE Access, 2020, 8: 98874-98886.
- [7] M. S. Mollison and J. H. Anderson, “Bringing theory into practice: A userspace library for multicore real-time scheduling,” in 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), Philadelphia, PA: IEEE, Apr. 2013, pp. 283–292. doi: 10.1109/RTAS.2013.6531100.
- [8] Anderson T E, Bershad B N, Lazowska E D, et al. Scheduler activations: Effective kernel support for the user-level management of parallelism[J]. ACM Transactions on Computer Systems (TOCS), 1992, 10(1): 53-79.
- [9] Qin H, Li Q, Speiser J, et al. Arachne: {Core-Aware} thread management[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018: 145-160.
- [10] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. GhOST: Fast & Flexible User-Space Delegation of Linux Scheduling. In Proceedings of the ACM SIGOPS 28th Symposium on Operating

Systems Principles (SOSP '21). Association for Computing Machinery, New York, NY, USA, 588–604. <https://doi.org/10.1145/3477132.3483542>

[11] Li Y, Lazarev N, Koufaty D, et al. LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling[C]//2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2024: 922-936.

[12] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, “Scheduling-context capabilities: a principled, light-weight operating-system mechanism for managing time,” in Proceedings of the Thirteenth EuroSys Conference, Porto Portugal: ACM, Apr. 2018, pp. 1–16. doi: 10.1145/3190508.3190539.

[13] T. Aswathanarayana, D. Niehaus, V. Subramonian, and C. Gill, “Design and Performance of Configurable Endsystem Scheduling Mechanisms,” in 11th IEEE Real Time and Embedded Technology and Applications Symposium, San Francisco, CA, USA: IEEE, 2005, pp. 32–43. doi: 10.1109/RTAS.2005.17.

[14] Baumann A, Barham P, Dagand P E, et al. The multikernel: a new OS architecture for scalable multicore systems[C]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009: 29-44.

[15] J. Li, S. Dinh, K. Kieselbach, K. Agrawal, C. Gill, and C. Lu, “Randomized Work Stealing for Large Scale Soft Real-Time Systems,” in 2016 IEEE Real-Time Systems Symposium (RTSS), Porto, Portugal: IEEE, Nov. 2016, pp. 203–214. doi: 10.1109/RTSS.2016.028.

[16] D. Ferry, Jing Li, M. Mahadevan, K. Agrawal, C. Gill, and Chenyang Lu, “A real-time scheduling service for parallel tasks,” in 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), Philadelphia, PA: IEEE, Apr. 2013, pp. 261–272. doi: 10.1109/RTAS.2013.6531098.

[17] Q. Wang and G. Parmer, “FJOS: Practical, predictable, and efficient system support for fork/join parallelism,” in 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), Berlin, Germany: IEEE, Apr. 2014, pp. 25–36. doi: 10.1109/RTAS.2014.6925988.

[18] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi, “Real-Time Scheduling and Analysis of OpenMP Task Systems with Tied Tasks,” in 2017 IEEE Real-Time Systems Symposium (RTSS), Paris: IEEE, Dec. 2017, pp. 92–103. doi: 10.1109/RTSS.2017.00016.

[19] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quinones, “Timing characterization of OpenMP4 tasking model,” in 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), Amsterdam, Netherlands: IEEE, Oct. 2015, pp. 157–166. doi: 10.1109/CASES.2015.7324556

[20] Broquedis F, Diakhate F, Thibault S, et al. Scheduling dynamic OpenMP applications over multicore architectures[C]//OpenMP in a New Era of

Parallelism: 4th International Workshop, IWOMP 2008 West Lafayette, IN, USA, May 12-14, 2008 Proceedings 4. Springer Berlin Heidelberg, 2008: 170-180.

[21] J. Stoess, “Towards effective user-controlled scheduling for microkernel-based systems,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 4, pp. 59–68, Jul. 2007, doi: 10.1145/1278901.1278910.

[22] G. Parmer and R. West, “Predictable Interrupt Management and Scheduling in the Composite Component-Based System,” in *2008 Real-Time Systems Symposium*, Barcelona, Spain: IEEE, Nov. 2008, pp. 232–243. doi: 10.1109/RTSS.2008.13.

[23] B. Ford and S. Susarla, “CPU Inheritance Scheduling”.

[24] Insik Shin and Insup Lee, “Periodic resource model for compositional real-time guarantees,” in *Proceedings. 2003 International Symposium on System-on-Chip (IEEE Cat. No.03EX748)*, Cancun, Mexico: IEEE Comput. Soc, 2003, pp. 2–13. doi: 10.1109/REAL.2003.1253249.

[25] M.-K. Yoon, M. Liu, H. Chen, J.-E. Kim, and Z. Shao, “Blinder: Partition-Oblivious Hierarchical Scheduling”.

[26] Yoon M K, Kim J E, Bradford R, et al. Timedice: Schedulability-preserving priority inversion for mitigating covert timing channels between real-time partitions[C]//2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2022: 453-465.

[27] Wei H, Shao Z, Huang Z, et al. RT-ROS: A real-time ROS architecture on multi-core processors[J]. *Future Generation Computer Systems*, 2016, 56: 171-178.

[28] Saito Y, Sato F, Azumi T, et al. Rosch: real-time scheduling framework for ros[C]//2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). IEEE, 2018: 52-58.

[29] Suzuki Y, Azumi T, Kato S, et al. Real-time ros extension on transparent cpu/gpu coordination mechanism[C]//2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC). IEEE, 2018: 184-192.

[30] Saito Y, Azumi T, Kato S, et al. Priority and synchronization support for ROS[C]//2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA). IEEE, 2016: 77-82.

[31] Blaß T, Casini D, Bozhko S, et al. A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance[C]//2021 IEEE Real-Time Systems Symposium (RTSS). IEEE, 2021: 41-53.

[32] Casini D, Blaß T, Lütkebohle I, et al. Response-time analysis of ROS 2 processing chains under reservation-based scheduling[C]//31st Euromicro Conference on Real-Time Systems. Schloss Dagstuhl, 2019: 1-23.

[33] Yang Y, Azumi T. Exploring real-time executor on ros 2[C]//2020 IEEE international conference on embedded software and systems (ICESS). IEEE,

2020: 1-8.

[34] Tang Y, Feng Z, Guan N, et al. Response time analysis and priority assignment of processing chains on ROS2 executors[C]//2020 IEEE Real-Time Systems Symposium (RTSS). IEEE, 2020: 231-243.

[35] Jiang X, Ji D, Guan N, et al. Real-time scheduling and analysis of processing chains on multi-threaded executor in ros 2[C]//2022 IEEE Real-Time Systems Symposium (RTSS). IEEE, 2022: 27-39.

[36] Choi H, Xiang Y, Kim H. PiCAS: New design of priority-driven chain-aware scheduling for ROS2[C]//2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2021: 251-263.

[37] Sobhani H, Choi H, Kim H. Timing Analysis and Priority-driven Enhancements of ROS 2 Multi-threaded Executors[C]//2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2023: 106-118.

[38] Liu S, Jiang X, Guan N, et al. RTeX: an Efficient and Timing-Predictable Multi-threaded Executor for ROS 2[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2024.

[39] X. Luo, X. Jiang, N. Guan, H. Liang, S. Liu and W. Yi, "Modeling and Analysis of Inter-Process Communication Delay in ROS 2," 2023 IEEE Real-Time Systems Symposium (RTSS), Taipei, Taiwan, 2023, pp. 198-209.

[40] D. Enright, Y. Xiang, H. Choi and H. Kim, "PAAM: A Framework for Coordinated and Priority-Driven Accelerator Management in ROS 2," 2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS), Hong Kong, Hong Kong, 2024, pp. 81-94, doi: 10.1109/RTAS61025.2024.00015.