中国科学技术大学
University of Science and Technology of China

本 科 毕 业 论 文

题　目 一种基于实时 Linux 的 ROS2 改进方案的研究

英　文 **Research on ROS2 Improvement Scheme Based on**

题　目 **Real-Time Linux**

院　系 少年班学院

姓　名 陶子扬　　学　号 **PB18000138**

导　师 张燕咏 教授

日　期 **2022 年 5 月 21 日**

摘　要

本文旨在优化机器人操作系统的实时任务响应能力，提高实际应用的性能。本文主要从以下两个方向开展研究，第一：Linux baseline 的某些机制导致的实时性缺陷，以及内核实时补丁 Preempt RT 对其的优化方法；第二：主流机器人操作系统 ROS2 的调度体系结构的实时性缺陷，以及一种改进的调度体系结构解决方案 ROS2_PiCAS 的优化方法。本文在通过实验，测试机器人应用在 baseline、ROS2_PiCAS 调度方案、ROS2_PiCAS 调度方案与实时补丁 Preempt RT 共同作用的情况下的端到端时延，证明了内核实时补丁 Preempt RT 和改进的调度体系结构解决方案 ROS2_PiCAS 确实大幅提升了系统的实时性。最后做出了总结与应用前景的展望。

关键词：实时性；Preenmpt RT 补丁；机器人操作系统；ROS2_PiCAS

ABSTRACT

This paper aims to optimize the real-time task response capability of robot operating system and improve the performance of practical applications. This paper mainly carries out research from the following two directions: First: real-time defects caused by some mechanism of Linux Baseline, and the optimization method of kernel real-time patch Preempt RT for it; Second, the real-time defects of the scheduling architecture of ROS2, the mainstream robot operating system, and the optimization method of ROS2_PiCAS, an improved scheduling architecture solution. In this paper, the end-to-end delay of robot application is tested under the situation of baseline, ROS2_PiCAS scheduling scheme, ROS2_PiCAS scheduling scheme and real-time patch Preempt RT. It is proved that kernel real-time patch Preempt RT and improved scheduling architecture solution ROS2_PiCAS can greatly improve the real-time performance of the system. Finally, the summary and application prospect are made.

Key Words: Real-time; Preenmpt RT patch; Robot operating system; ROS2_PiCAS

目 录

第一章　简介

第一节　实时性及其在 Linux 中的实现

一、实时概念

1. 实时性

实时性定义为计算或进程中的其他各种操作 (统称为任务) 必须保证在指定时间 (截止日期) 内响应的能力。我们定义相对截止时间为指定的任务完成时间与实际任务发起时间的差值。

2. 实时系统

实时系统是一个有明确的、固定的时限的系统约束。任务必须在给定的约束内完成，否则系统出错甚至引发崩溃。满足实时系统的工作方式又分为两种，硬实时和软实时。硬实时的含义是在一定条件下，内核保证可以满足任何约束；相反地，软实时的含义是内核努力使进程在它的限定时间内运行，但是不能保证总是满足这些约束。

二、linux baseline 实现实时性的机制

1. 抢占

如果系统中进程的数目比处理器的数目要多，那么就一定会有进程处于等待状态，换言之，注定有一些进程不能一直执行。如果处于等待状态的进程有实时性要求，那么在不加干扰的情况下，该进程很有可能会错过它的截止日期，从而引发系统出错或崩溃。为了避免这种情况，Linux 提供了进程抢占机制，用于停止一个进程的运行以便其他进程能够有运行的机会。进程在被抢占之前能够运行的时间被事先配置，这个时间被称为时间片 (timeslice)，时间片的设置可以起到避免个别进程独占系统资源的作用。

2. 优先级

linux 为进程提供了一组优先级，优先级高的进程先执行，低的后执行。linux baseline 内核提供了两种独立的优先级范围：第一种被称为 nice 值，范围从-20 到 19，nice 值越大，进程的优先级越低；第二种被称为实时优先级，范围从 0 到 99。在默认情况下，nice 值与实时优先级共享取值空间，nice 值的-20 到 19 与实时优先级的 100 到 140 直接对应，即配置了实时优先级的进程的优先级一定比

配置了 nice 值优先级的进程要高。

3. 调度

调度程序是 linux 内核的组成部分，它主要负责选取下一个要被执行的进程。调度程序会根据给定的策略决定何时让什么进程运行，比如时间片、优先级等等。

linux baseline 在实时性方面的机制体现为两种调度策略：SCHED_FIFO 和 SCHED_RR[1]，被配置为使用这两个调度策略的进程需要配置实时优先级。SCHED_FIFO 是一种简单的、先入先出的调度算法，它不使用时间片。只要 SCHED_FIFO 级别的进程处于可执行状态，就会一直执行，直到进程自己被阻塞或者被释放为止。SCHED_RR 与 SCHED_FIFO 基本一致，区别在于调度策略为 SCHED_RR 级的进程在耗尽事先分配给它的时间后就不能再执行。这两种实时算法的优先级分配都是静态的，这保证了给定优先级级别的实时进程总能抢占优先级比它低的进程。

Linux baseline 的实时调度算法实现的是一种软实时工作方式。

三、linux 的硬实时实现——Preempt RT 补丁

随着 Linux 越来越多的被应用于需要实时控制的环境，如工厂自动化处理、雷达系统、医疗系统、汽车控制系统等，这些系统的任务级响应时间要求在 10 到几百微秒的范围内，Linux baseline 已经无法保证这种响应水平[2]，这种问题由于 Linux 自身的一系列数据结构的不可抢占导致的。比如同步机制——锁的不可抢占性导致的优先级反转问题、中断处理的不可抢占性等，这些都导致了实时性要求高的任务被迫等待。

于是从 Linux 内核版本 2.6 开始，为了实现更好的实时响应性能，基于 Linux 内核的 Preempt RT 内核补丁开始被维护。它通过对一系列内核机制的修改，包括锁的优先级继承、中断线程化、RCU 机制优化等，实现了内核的几乎全部数据结构的抢占性，大幅提高了系统的实时响应能力。具体修改内容将在第二章论述。

第二节　机器人中间件 ROS 及改进方案

一、机器人中间件

在机器人系统中有大量的软件在同时运行，比如语音识别、图像识别、定位、路径规划、底盘控制等等，这些软件的不同进程之间需要传输不同的数据。基于目前软件基本都实现了模块化管理的情况，软件的开发越来越便利，但是对于通信的要求也越来越高。运行中的的进程之间需要考虑共享哪些信息，如何传输这些信息，如何维护消息队列等等问题。

基于上述功能需求，机器人中间件被引入到机器人系统中。中间件是介于应用和操作系统之间的软件，机器人的中间件广义上也属于操作系统，但是它与 Linux 等底层系统不一样，本质上是起到软硬件管理与分配、软件通信、进程调度的作用。机器人中间件使应用与系统、软件与硬件解耦，帮助上层应用的开发者更高效、灵活地开发软件。

目前机器人系统越来越多的应用于自动驾驶系统中，然而自动驾驶系统中软件的功能更加复杂，包括接收传感器信号、感知、路径规划、控制、高精度定位等等。并且自动驾驶的应用有着高性能、高并发的需求，这对于中间件的实时性、可靠性、确定性都有更高的要求，这也是目前各中间件厂家、研究人员所突破的方向之一。

二、ROS 与 ROS2

ROS 作为目前机器人主流的开源中间件，自从发布以来被广泛应用于各种机器人应用场景[3]。它的出现彻底改变了工业界和学术界的开发环境，为应用的开发提供了大量高效的工具，使得机器人应用的模块化开发更加高效、便捷。

然而在过去几十年里，许多场景已经表明 ROS 对于实时性、确定性敏感的应用程序的实时支持存在缺陷，这推动了 ROS 的第二代版本——ROS2 的开发。2017 年 ROS2 发布，它在继承了 ROS 的大部分特性的同时引入了新的结构，包括采用数据分发服务 (Data Distribution Service) 来实现节点之间的通信。数据分发服务是工业界的分布式通信中间件协议，它规定节点通信需要使用发布/订阅结构，每个需要通信的节点对共同指向一个被称为 topic 的数据结构，通过发布者向 topic 传入数据，订阅者从 topic 提取数据来完成通信。这种设计实现了通信节点的解耦，使得数据可以灵活、实时、高效的分发。

三、ROS2 的改进方案——ROS2_PiCAS

ROS2 创新性的使用数据分发服务来提高数据传输的性能，但是对于其实时性能仍然有很大的研究空间。在具体应用场景中，应用程序由回调 (callback) 函数组成，这些函数包含了应用程序的运算过程。多个应用程序之间通常形成一个链，它由一组具有数据依赖的 callback 函数组成，每个回调的输入都依赖一些特定回调输出的数据。每个回调函数都需要一定的执行时间，并且回调之间具有数据通信关系。因此我们有数据链端到端的延迟的概念，即从数据链头部收到输入开始到数据链尾部输出为止这段时间。端到端延迟上限对于有严苛的实时要求的场景是至关重要的，比如对于自动驾驶系统，它的数据链的过长的端到端延迟可能会导致追尾事故。

然而 ROS2 内部没有定义 chain 的任何属性，而且调度程序中也没有考虑到 chain 的时间和资源需求。ROS2_PiCAS 提供了一种改进的调度体系结构，大幅提升了端到端的响应能力。具体方案将在第三章论述。

第二章　Linux 实时补丁 Preempt RT

第一节　Linux baseline 在实时性方面的问题

一、锁

1. 锁的作用与机制

由于 Linux 内核独特的抢占机制以及自 2.0 版本以来的对称处理器架构，共享资源的线程有可能并发执行对资源的访问和操作，各线程之间就有可能发生覆盖各自共享资源的情况，引发了一致性问题。称访问和操作共享资源的代码段为临界区，给临界区加锁的目的就是保证临界区在任意时刻只能被一个线程访问，从而避免共享资源的恶性覆盖。

Linux 提供了如下几种不同机制的锁：

（1）自旋锁

自旋锁最多只能被一个线程持有。如果一个线程试图获取被使用的自旋锁，那么该线程不会立刻放弃 CPU 时间片，而是一直循环尝试获得锁直到获取为止。自旋锁不涉及操作系统的上下文切换，适用于被短时间持有的情况，并且禁止抢占。自旋锁不可递归，如果试图得到一个正被持有的自旋锁，用户必须自旋等待自己释放这个锁，但是自己正处于忙等待中，自旋锁永远不可能被释放，这样就陷入了死锁。所以使用自旋锁时应禁止本地中断。

（2）互斥锁

互斥锁 (mutex) 一次只能有一个线程持有，其他想获取正在被持有的互斥锁的线程会主动放弃 CPU 进入睡眠状态，直到锁的状态改变时再被操作系统的调度器唤醒。为了实现互斥锁的状态发生改变时唤醒阻塞的线程或者进程，需要把互斥锁交给操作系统管理，所以互斥锁在加锁操作时涉及上下文的切换。互斥锁适用于被长时间持有的情况，且不会禁止内核抢占。

（3）读写锁

对某个数据结构的操作可以被划分为读写两种类别时，可以使用读写锁。多个读任务可以并发的共同持有同一个读者锁，但是用于写的锁最多只能被一个写任务持有，并且此时并发的读操作会睡眠；读任务持有锁时，并发的写操作也必须等待。多个读者可以同时获得同一个读锁，也可以线程递归地获得同一个读锁，这一特性使得读操作使用锁时不用禁用本地中断。在读任务持有锁时，写操

作会像自旋锁一样自旋等待，所以有大量读操作时，写操作会浪费 CPU 资源。

（4）RCU

RCU，全称为 Read-Copy-Update，是一种改进的读写锁机制，实现了读写的并行。

执行 RCU 机制的过程如下：要修改共享数据时，系统先分配一段新的内存空间来存放旧数据的一个 copy，然后负责写操作的 writer 修改这个 copy。正在执行读操作的 reader 读取的是旧数据的值，在此时完全不受影响。修改完成后 writer 修改原节点的前驱的指针域指向这个 copy。等所有旧数据区的 reader 都完成了相关操作，writer 释放旧数据内存区域。

writer 需要等待所有正在读的 reader 读取完成后，将旧内存释放。如果读取时间过长，则会让 writer 长时间等待。所以 RCU 使用基于回调机制的 call_rcu() 函数来等待读取完成，传递给 call_rcu() 的形式参数的回调函数的正是释放旧数据内存区域的函数。

RCU 机制涉及到 grace period 的概念。grace period 是一段时间，它从作为 writer 的 CPU 准备释放一个对象时开始，释放的具体操作 (callback) 被填入这个对象内含的 rcu_head() 中，然后被 rcu_head() 带着加入到待执行的链表中, 到读取 writer 修改的共享数据的 reader 全部退出临界区时结束。

RCU 判断 grace period 结束的方法如下：在不可抢占的 RCU 中，reader 进入临界区时，reader 所在的 CPU 上的调度是关闭的，直到退出临界区后，调度才会重新打开。并且临界区内的代码是被要求不能睡眠/阻塞的，因此不会发生线程切换。如果接下来该 CPU 开始执行其他任务，那么说明发生了线程切换，即开启了调度，进而说明该 CPU 退出了临界区。另外由于 RCU 临界区代码一定是在内核态执行，在 tick 时钟中断时，如果识别出该 CPU 已经在 userspace 执行代码，同样可以判断临界区已经退出。

因此 writer 在实际实现时被划分为 updater 和 reclaimer。updater 更新 copy 的数据，并且修改原节点的前驱的指针域指向这个 copy，并移除对原节点的引用，然后进入 grace period。grace period 结束以后，reclaimer 释放原节点。

通过以上机制我们可以看出，reader 和 updater 可以实现真正的并行，updater 和 reclaimer 也可以并行，updater 和 updater 不能并行，reclaimer 和 reader 能不能并行取决于 reclaimer 对应的 grace period 是否包含相关的 reader。

2. 锁存在的问题

（1）自旋锁性能

如果一个线程试图获取被使用的自旋锁，它会一直循环尝试获得锁直到获取为止。自旋锁适用于被短时间持有的情况，但是如果被长时间持有，循环会占用 CPU 很多利用率，导致性能下降。

（2）优先级反转

优先级反转是指：在争用互斥锁的某些情况下，高优先级的进程被较低优先级的进程阻塞。我们用图 2.1 来描述这个现象：

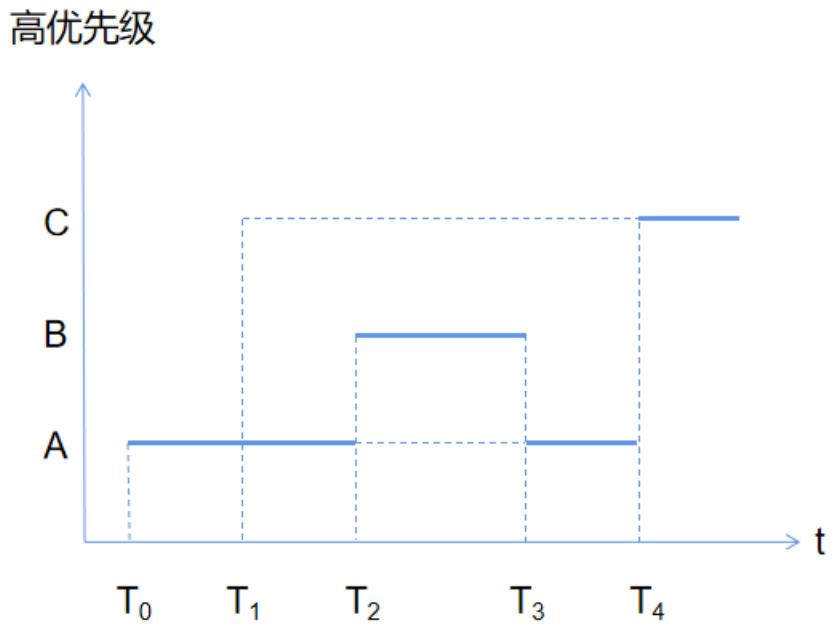

图 2.1　优先级反转

如图 2.1，A、B、C 是三个不同进程，优先级顺序为 A<B<C，其中 A、C 争用同一个互斥锁。在时刻 T_0，A 进入临界区，持有互斥锁；时刻 T_1 时，C 试图获得锁，但锁已经被 A 持有，C 只好等待；时刻 T_2 时，B 进程进入运行状态，由于优先级高于 A，B 立刻抢占 A 获得 CPU 使用权；时刻 T_3 时，B 进程执行结束，A 继续执行剩余部分；时刻 T_4 时，A 进程执行结束，释放互斥锁，此时 C 终于等待到互斥锁释放，开始执行。

由上述过程可以看出，进程 C 比进程 B 优先级高，却被一直被 B 阻塞，这导致实时任务 C 很有可能错过它的截止时间。

（3）RCU 性能

RCU 回调在软中断上下文中执行，并且不可抢占，这种回调通常会释放内存，而内存操作会导致较大延迟。

二、中断

Linux 通过中断使硬件与处理器通信。在响应特定中断时，内核会运行一个函数，这个函数被称为中断处理程序。中断处理程序运行在中断上下文中，与进程上下文不同，中断上下文不可睡眠。所以执行中断处理程序地同时，会禁用其他硬件中断、软中断，并且自身不可被抢占。

Linux 会把需要中断任务即刻响应地部分分配给中断处理程序执行，其他部分则放到称为下半部的程序中执行。下半部也运行在中断上下文中，同样，下半部也禁用其他硬件中断、软中断，并且自身不可被抢占 (唯一可被中断处理程序抢占)。在 Linux baseline 中，中断处理程序的优先级比任何实时任务都要高。

中断处理程序与下半部的不可抢占性对实时性产生了显著的影响，导致本应该优先得到执行的任务停留在饥饿状态。

第二节　Preempt RT 补丁的改进内容

一、rt_mutex

rt_mutex 是 Preempt RT 补丁中的新的锁机制，rt_mutex 与普通 mutex 的区别在于其增加了优先级继承的特性。这种技术要求低优先级任务继承它们共享的资源上任何高优先级任务的优先级。一旦高优先级的任务开始等待，这个优先级的改变就应该发生，应该在资源被释放时结束。我们用图 2.2 来解释这个特性：

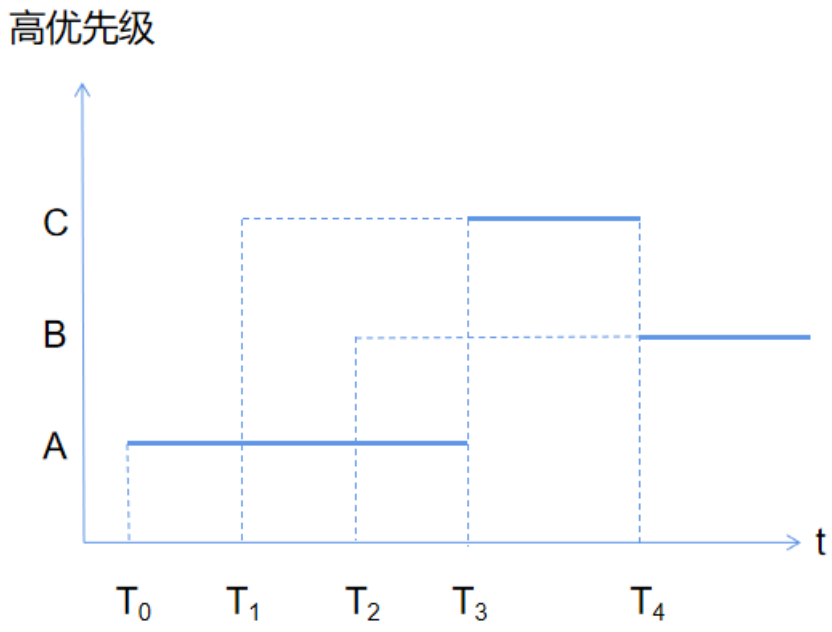

图 2.2　优先级继承

如图 2.2，A、B、C 是三个不同进程，优先级顺序为 A<B<C，其中 A、C 争用同一个互斥锁。在时刻 T_0，A 进入临界区，持有互斥锁；时刻 T_1 时，C 试图获得锁，但锁已经被 A 持有，C 只好等待，并且此时 A 检测到有更高优先级任务 C 在等待自己持有的互斥锁，于是它立即将自己的优先级设为与 C 相同；时刻 T_2 时，B 进程进入等待队列，由于优先级低于 A(C) 此时的优先级，B 继续等待；时刻 T_3 时，A 进程执行结束，释放互斥锁，调度器选择等待队列中优先最高的 C 进程开始执行，同时 A 进程将它的优先级还原；时刻 T_4 时，C 进程执行结束，B 进程开始执行。

我们看到 rt_mutex 地优先级继承特性完美地解决了优先级反转的问题。在 Preempt RT 补丁中，几乎所有的自旋锁和互斥锁被隐式的转化为 rt_mutex，rt_mutex 对于锁的优化大幅提升了系统实时能力。

二、RCU 线程化和可抢占性

我们已经知道 RCU 回调在软中断上下文中执行会导致较大延迟。实际上 Linux 可以选择在指定的 cpu 上使用内核线程执行回调函数，并这些内核线程可以根据需要分配优先级。Preempt RT 补丁会使 RCU 用 kthread 替换掉了大部分软中断的执行。

Preempt RT 补丁也可以配置 RCU 的读可抢占性。但是可抢占 RCU 的缺点是，低优先级的任务可能会在 RCU 读临界区中间被抢占，如果系统高优先级任务一直抢占，那么低优先级任务可能永远不会恢复，也不可能离开其临界区，这导致了 RCU grace period 无限延长。解决这个问题的方法被称为优先级提升 (Priority Boosting)，在默认情况下将被当前 grace period 阻塞超过半秒的任务提升到实时优先级 1。

三、中断处理线程化

Preempt RT 补丁中，中断处理程序用在线程上下文中执行，并且其优先级一定比实时优先级最高的任务低。默认情况下，中断处理程序实时优先级设置为 50，调度策略设置为 SCHED_FIFO。基于线程的中断处理程序使中断能在线程上下文中可被抢占地运行，因此显著减少了内核的处理延迟和优先抢占延迟。

第三章　ROS2 调度体系结构的改进方案——ROS2_PiCAS

第一节　ROS2 调度体系结构及存在的问题

一、ROS2 调度体系结构

ROS2 作为广义上的操作系统，本身具有调度、通信、资源分配等功能。本文主要关注其调度体系结构的设计与可改进方向。

1. callback

回调 (callback) 是 ROS2 中的最小调度实体，本质上是包含各种特定功能的函数。callback 被操作系统加入等待队列，在合适时候被唤醒执行。

callback 分为时间触发与事件触发两种。时间触发的 callback 根据系统的定时器周期性的被执行，事件触发的 callback 需要满足一定条件 (例如输入数据产生) 才能被执行。事实上，ROS2 正是通过事件触发的 callback 实现了发布/订阅的通信结构[4]。

2. node

节点 (node) 由应用程序产生，是一个 callback 的集合。在应用程序中对 node 内的内容进行模块化设计，便于为需要设计的功能归类。具体应用程序应包含多个 node，每个 node 包含多个 callback。在面向对象语言中，node 体现为类的实例，而 callback 体现为 node 的方法。

3. executor

执行器 (executor) 是在 CPU 上执行的、OS 层面调度的实体，具体体现为线程。在 ROS2 的调度程序中，executor 被 CFS 调度算法执行。callback 需要通过 node 绑定到 executor 上来完成执行，node 是 executor 的最小分配单元。在 ROS2 的调度程序中，executor 被 CFS 调度算法执行。因此，所有在同一个 node 中的 callback 将被同一个 executor 执行。executor 内的 callback 调度与普通优先级实时调度不同，callback 的优先级由类型决定，时间触发的 callback 的优先级比事件触发的 callback 的优先级高，所有 callback 被不可抢占式执行。

二、ROS2 调度体系结构存在的问题

我们首先介绍 chain 的概念。数据链 (chain) 由一组有先后顺序的 callback 组成，即这些 callback 呈链式结构，并且除了起始 callback，链中的 callback 都是数据触发。对于时间触发的 chain，起始 callback 是时间触发的 callback；对于事件触发的 chain，起始 callback 是事件触发的 callback。

Chain 的具体构成由应用程序决定，在 ROS2 里没有实体承载，只是一种数据结构的抽象。ROS2 没有定义 chain 的任何属性，而且调度程序中也没有考虑到 chain 的时间和资源需求。由于在 ROS2 中 executor 直接被操作系统的 CFS 调度算法执行，导致无法优先考虑包含高实时性要求 chain 的 executor。更严重的问题是，由于时间触发的 callback 的优先级比事件触发的 callback 的优先级高，ROS2 会先调度所有时间触发的 callback，这导致实时性要求高的 chain 的端到端时延被拉长。

另一方面，ROS2 的简易调度算法可能会产生 chain 的自干扰，即如果 chain 的两个实例被同一个 CPU 执行，chain 的前一个实例未能下一个实例开始执行之前执行完毕的情况。那么在下一个实例运行时，一定会被前一个实例抢占，这造成了下一个实例不该有的延迟。

由于链的端到端延迟对实时系统的性能有重大影响，chain 的实时性调度、资源分配需要被重点关注。因此需要信息量更加丰富的调度单元以及高效的调度算法。

第二节　ROS2_PiCAS 改进方案

PiCAS 全称为 a Priority-driven Chain Aware Scheduler。改进方案分为两方面，分别为调度单元和调度算法。

一、调度单元

ROS2_PiCAS 在 ROS2 原有的调度单元上增加了以下调整：

1. callback

为每个 callback 分配属性集合，包含周期、相对时限、最大执行时间、优先级，其中周期与 callback 所属的 chain 的周期相同，最大执行时间由预先执行测试确定，优先级由 ROS_PiCAS 分配算法确定。

2. executor

为每个 executor 分配实时优先级，并设置调度策略为 SCHED_FIFO。优先级由 ROS_PiCAS 分配算法确定。

3. chain

每个 chain 都被人为赋予优先级，理论上按照 chain 在完整应用系统中的重要性高低决定优先级的高低。

二、调度算法

1. 核心思想与理论证明

我们的调度目标主要有以下两点：高优先级 chain 应该比低优先级 chain 先执行；如果 chain 的两个实例被同一个 CPU 执行，chain 的前一个实例应该在下一个实例开始执行之前执行完毕，这是为了避免同一个 chain 的实例之间的自干扰。我们的调度算法实现以上目标的方式主要依赖如下定理：

定理：如果 chain 的两个实例在同一个 CPU 上执行，chain 的前一个实例保证在下一个实例开始执行之前执行完毕当以下条件能被满足时成立：chain 的数据依赖下游 callback 的优先级比上游 callback 高；chain 的下游 callback 在比上游 callback 优先级更高的 executor 上执行。

证明：反证法。假设某个 chain 实例的 callback c_1 比前一个实例的 callback c_j 先执行 (这里 c_1 是这个 chain 的起始 callback)，那么存在三种情况。第一，c_1 比 c_j 优先级高，且这两个实例在同一个 executor 上；第二，c_1 比 c_j 优先级低，但是 c_1 运行在比 c_j 优先级更高的 executor 上；第三，c_1 比 c_j 优先级高，且 c_1 运行在比 c_j 优先级更高的 executor 上。这都与定理中的两个条件相悖，故得证。

2. callback、executor 优先级分配算法

（1）callback

对于事件触发的 chain 中的 callbacks，按在 chain 中的先后顺序分配，越靠后优先级越高；对于事件触发的 chain 中的 callbacks，事件触发的 callback 按 chain 中的顺分配序，越靠后优先级越高，时间触发的 callback 分配比所有事件触发的 callback 低的优先级。

对于多个事件触发的 chains 中的 callbacks，高优先级的 chain 中的所有 callbacks 分配比低优先级的 chain 中的所有 callbacks 高的优先级。事件触发的 Chain

内的 callbacks 优先级按前述方式分配。

对于多个时间触发的 chains 中的 callbacks，高优先级的 chain 中的时间触发的 callback 分配比低优先级的 chain 中的时间触发的 callback 高的优先级。时间触发的 Chain 内的 callbacks 优先级按前述方式分配。

（2）executor

对于包含同一个 chain 中 callback 的不同 executor，包含数据依赖上游的 callback 的 executor 分配比包含下游 callback 的 executor 低的优先级。对于包含不同 chain 中 callback 的不同 executor，包含高优先级的 chain 的 executor 分配比包含低优先级的 chain 的 executor 高的优先级。

3. node 绑定 executor、executor 绑定 CPU 算法

我们的核心思想是将与同一个 chain 关联的 nodes 尽可能分配到同一个 CPU 上，这样能减少 CPU 之间的通信开销。算法过程如下：

维护一个所有待分配的 node 集合 N^*，将 N^* 中的 node 按其包含的最高优先级的 callback 按降序排列。对于当前存在的包含的 callback 未被分配的最高优先级的 chain C，选择出 N^* 中与 C 关联的所有的 node 的集合 N。在线程池中尝试找出空闲的 executor e 分配给 N 中的 node，以下分三种情况：

（1）情况 A：有空闲的 executor e

此时把 N 中的 node 绑定到 e，尝试将 e 绑定到一个可用的 CPU。判断 CPU 可用意味着原 CPU 利用率加上 e 的利用率不超过 1。如果没有可用 CPU，从 N 中删除包含最低优先级 callback 的 node n，将其移回 N^*，然后再寻找可用 CPU，如此循环直到 N 中只有一个 node，如果此时仍然没有可用 CPU，转到情况 C。对于找到的可用 CPU，将 N 分配到满足 executor 优先级分配策略的最高优先级 executor，将此 executor 分配到最低利用率的 CPU。如果找不到满足策略的 executor，转到情况 C。

（2）情况 B：没有空闲的 executor

此时尝试找出一个对 N 可用的、已经被绑定到 CPU 核 P_k 的非空闲 executor e_m，判断可用意味着原 CPU 利用率加上被绑定 N 中的 node 后的 e_m 的利用率不超过 1。如果没有可用 executor，从 N 中移除包含最低优先级 callback 的 node n，将其移回 N^*，再寻找可用 executor，如此循环直到 N 中只有一个 node，如果此时仍然没有可用 executor，转到情况 C。对于找到的可用 executor，将 N 分配到满足 callback、executor 优先级分配策略的、最低利用率的 executor。如果找不到满足策略的 executor，转到情况 C。

（3）情况 C

有两种情形会导致情况 C：第一，找到了可用 CPU，但是不满足 executor 优先级分配策略，这种情况下，把这个 CPU 上所有 executor 中的 node 都重新分配给一个 executor，这样优先级分配策略被满足；第二：所有 CPU 利用率都超过 1，这种情况下直接把 N 分配到利用率最低的 CPU。

第四章　带 Preempt RT 补丁的 ROS2_PiCAS 实时性实验

第一节　实验设计

一、软硬件环境

实验硬件环境为 NVIDIA Xavier NX，CPU 个数为 8 核，附带 GPU。操作系统为 xavier 原生系统 tegra，Linux 内核版本为 4.9，ROS2 版本为 foxy。

二、实验方案

实验在无 Preempt RT 补丁的 ROS2、无 Preempt RT 补丁的 ROS2_PiCAS、有 Preempt RT 补丁的 ROS2_PiCAS 上分别测试不同情景下的端到端时延。为了更明显地观察到区分度，我们将所有 CPU 地负载拉满到接近 100%。

具体不同情景为：单个 chain 运行在一个 executor 上，多个 chain 运行在同一个 executor 上，多个 chain 且每个 chain 绑定到一个 executor 上，多个 chain 跨 executor 绑定。

第二节　实验结果与分析

一、单 chain 绑定一个 executor

本例测试一个 chain 绑定到一个 executor 的端到端时延。Chain 由四个 callback 组成，从数据链上游到下游依次为时间触发的 callback_t，执行时间设置为 2ms；由 callback_t 触发的事件触发 callback1，执行时间设置为 2ms；由 callback1 触发的事件触发 callback2，执行时间设置为 2ms；由 callback2 触发的事件触发 callback3，执行时间设置为 2ms。Chain 的结构图以及 callback 的绑定策略如图 4.1 所示。用定时器触发 50 次 chain，chain 的端到端时延如图 4.2 所示。

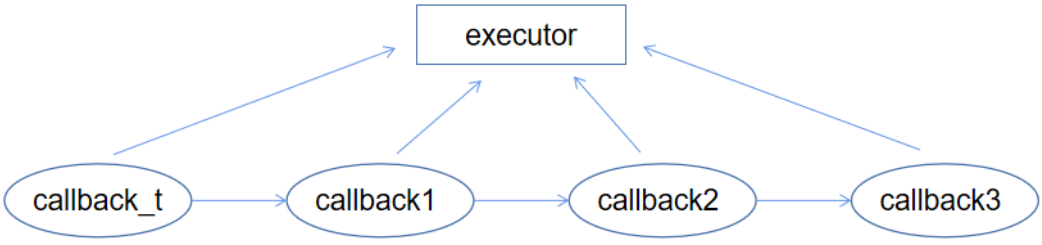

图 4.1　Chain 的结构图以及 callback 的绑定策略

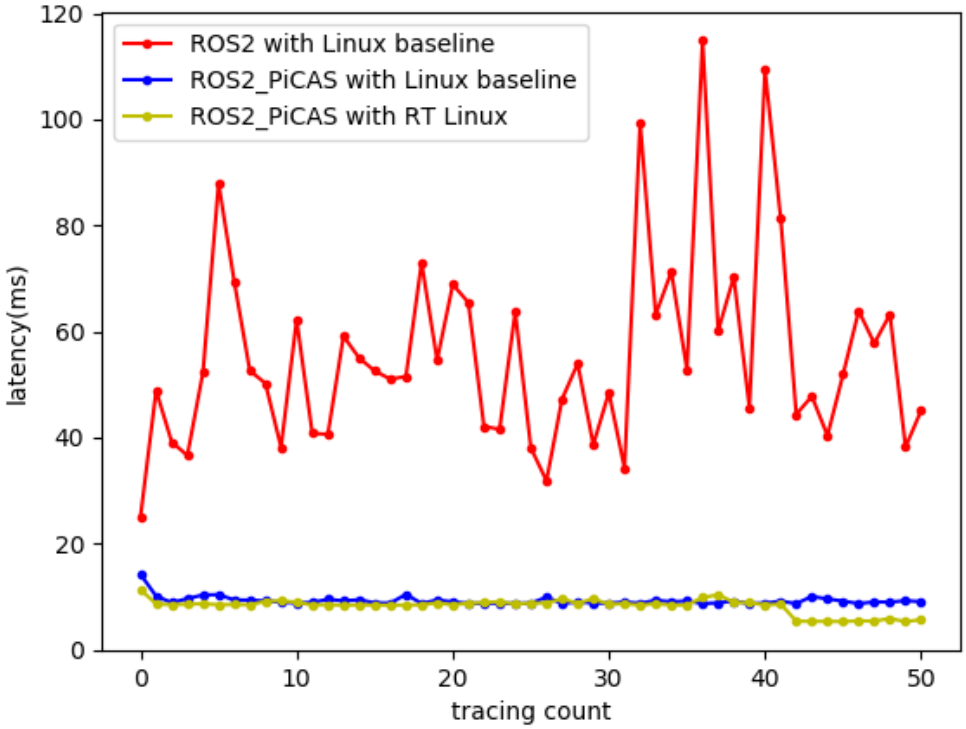

图 4.2　Chain 的端到端时延

可以看出，ROS2_PiCAS 大幅提升了稳定性和实时性。这是由于 ROS2_PiCAS 规定了有依赖关系 callback 的优先级，以及 executor 的实时优先级，提高了在系统中的实时响应能力。

二、多个 chain 绑定同一个 executor

本例测试两个 chain 绑定到同一个 executor 的端到端时延。Chain1 和 chain2 结构相同，都由四个 callback 组成，从数据链上游到下游依次为时间触发的 callback_t，执行时间设置为 2ms；由 callback_t 触发的事件触发 callback1，执行时间设置为 2ms；由 callback1 触发的事件触发 callback2，执行时间设置为 2ms；由 callback2 触发的事件触发 callback3，执行时间设置为 2ms。并且假设 Chain2 比 Chain1 的优先级高。Chain 的结构图以及 callback 的绑定策略如图 4.3 所示。用定时器触发 50 次 chain1 和 chain2，chain1 的端到端时延如图 4.4 所示，chain2 的端到端时延如图 4.5 所示。

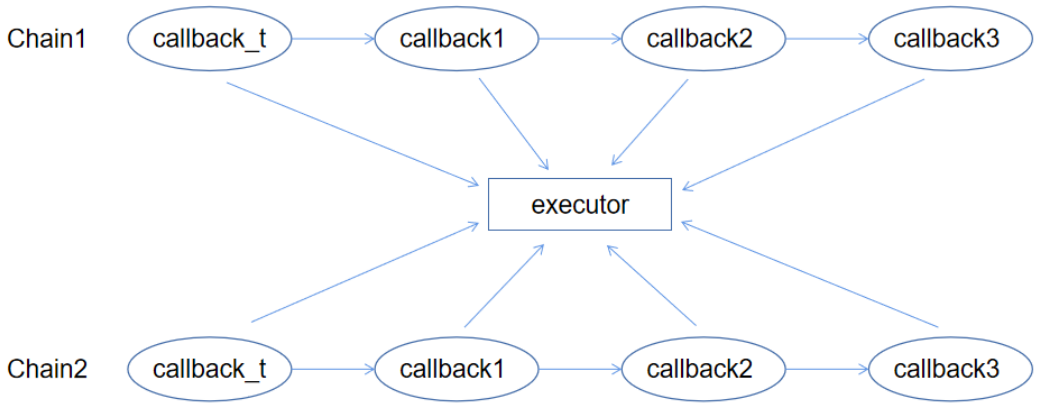

图 4.3　Chain 的结构图以及 callback 的绑定策略

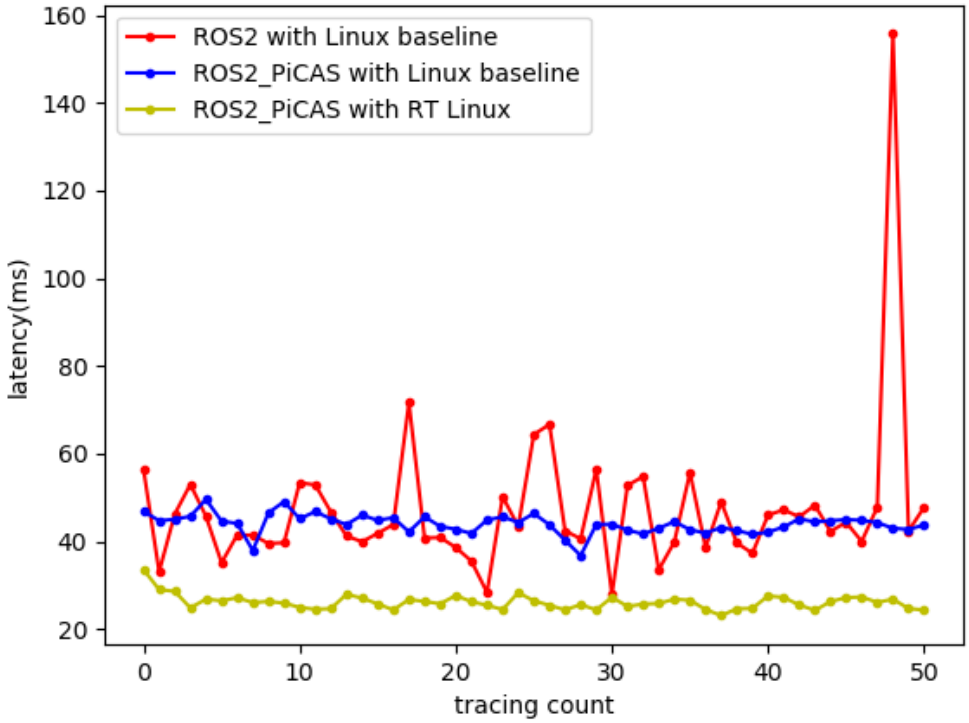

图 4.4　Chain1 的端到端时延

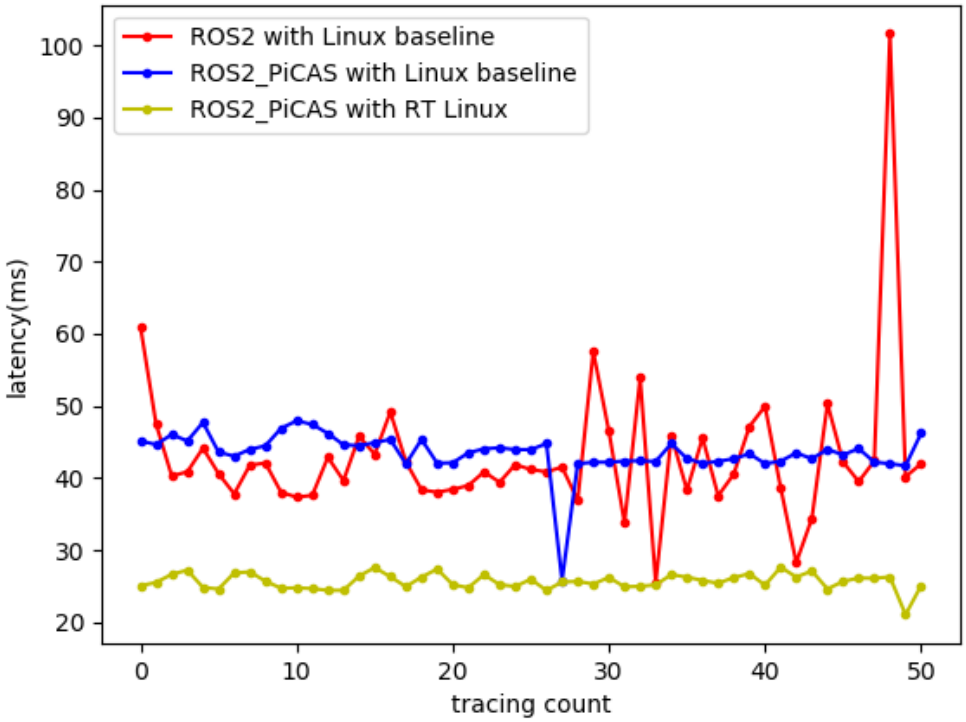

图 4.5　Chain2 的端到端时延

可以看出，ROS2_PiCAS 大幅提升了稳定性，Preempt RT 补丁大幅提升了实时性。这是由于 ROS2_PiCAS 规定了有依赖关系 callback 的优先级，以及 executor 的实时优先级，避免了 chains 在系统中的自干扰问题。Preempt RT 补丁避免了被各种其他中断程序持续打断的情况，提高了响应速度。

三、多个 chain 分别绑定一个 executor

本例测试两个 chain 分别绑定一个 executor 的端到端时延。Chain1 和 chain2 结构相同，都由四个 callback 组成，从数据链上游到下游依次为时间触发的 callback_t，执行时间设置为 2ms；由 callback_t 触发的事件触发 callback1，执行时间设置为 2ms；由 callback1 触发的事件触发 callback2，执行时间设置为 2ms；由 callback2 触发的事件触发 callback3，执行时间设置为 2ms。并且假设 Chain2 比 Chain1 的优先级高。Chain 的结构图以及 callback 的绑定策略如图 4.6 所示。用定时器触发 50 次 chain1 和 chain2，chain1 的端到端时延如图 4.7 所示，chain2 的端到端时延如图 4.8 所示。

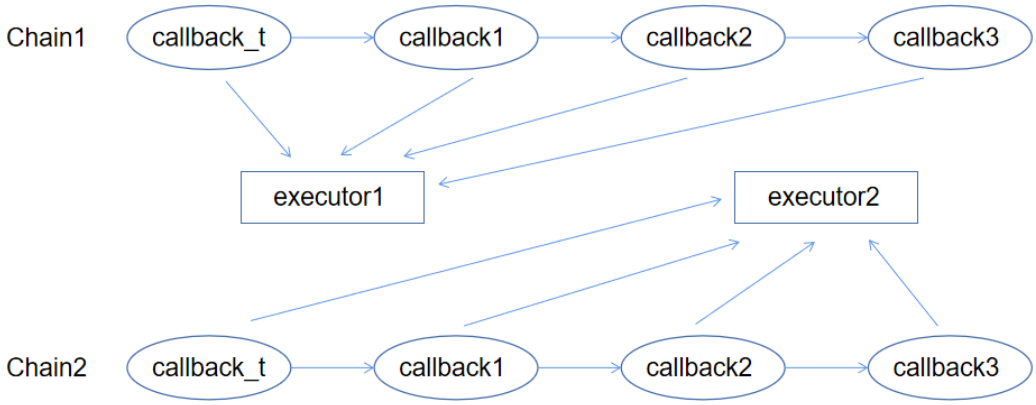

图 4.6　Chain 的结构图以及 callback 的绑定策略

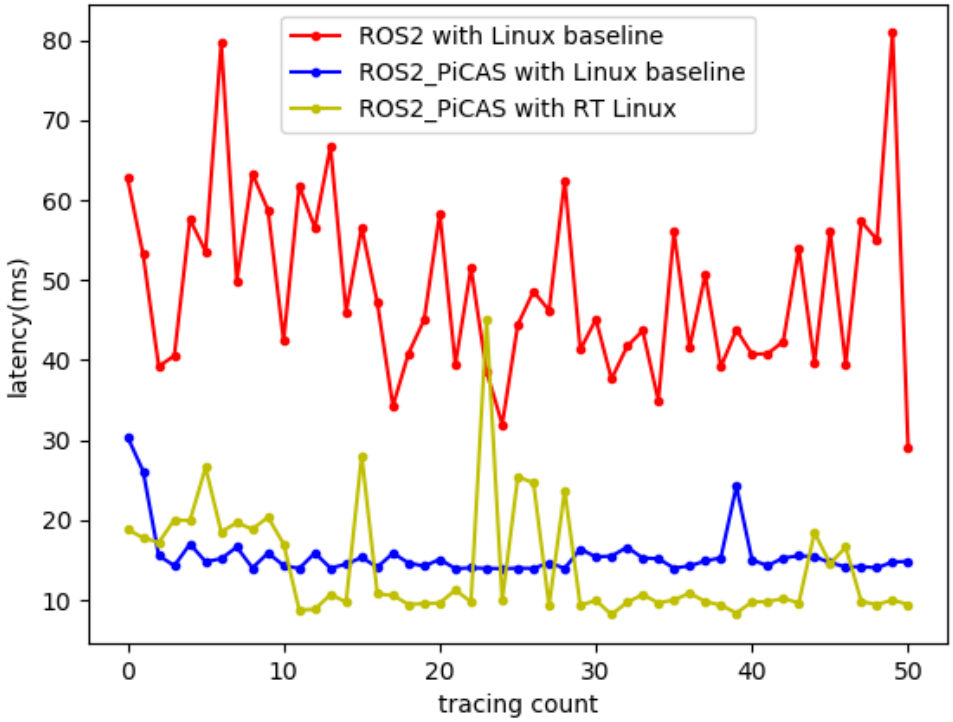

图 4.7　Chain1 的端到端时延

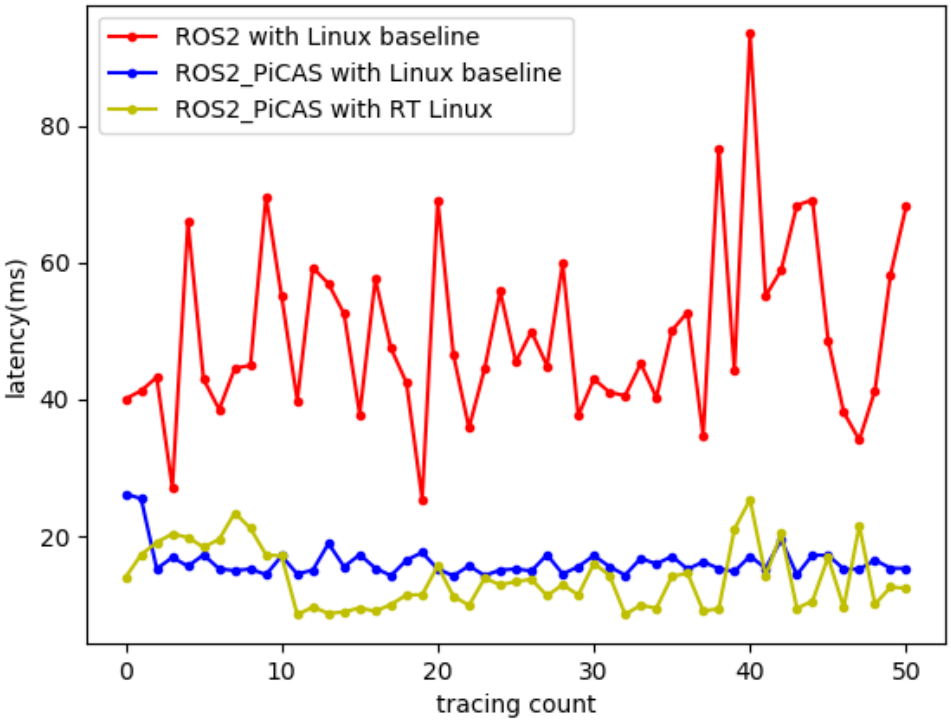

图 4.8　Chain2 的端到端时延

可以看出，ROS2_PiCAS 大幅提升了稳定性和实时性。

四、多个 chain 跨 executor 绑定

本例测试两个 chain 跨 executor 绑定的端到端时延。Chain1 和 chain2 结构相同，都由四个 callback 组成，从数据链上游到下游依次为时间触发的 callback_t，执行时间设置为 2ms；由 callback_t 触发的事件触发 callback1，执行时间设置为 2ms；由 callback1 触发的事件触发 callback2，执行时间设置为 2ms；由 callback2 触发的事件触发 callback3，执行时间设置为 2ms。通过将 Chain1 的前两个 callback 与 Chain2 的后两个 callback 绑定到一个 executor 上，将 Chain1 的后两个 callback 与 Chain2 的前两个 callback 绑定到一个 executor 上来实现 chain 的跨 executor 绑定。并且假设 Chain2 比 Chain1 的优先级高。Chain 的结构图以及 callback 的绑定策略如图 4.9 所示。用定时器触发 50 次 chain1 和 chain2，chain1 的端到端时延如图 4.10 所示，chain2 的端到端时延如图 4.11 所示。

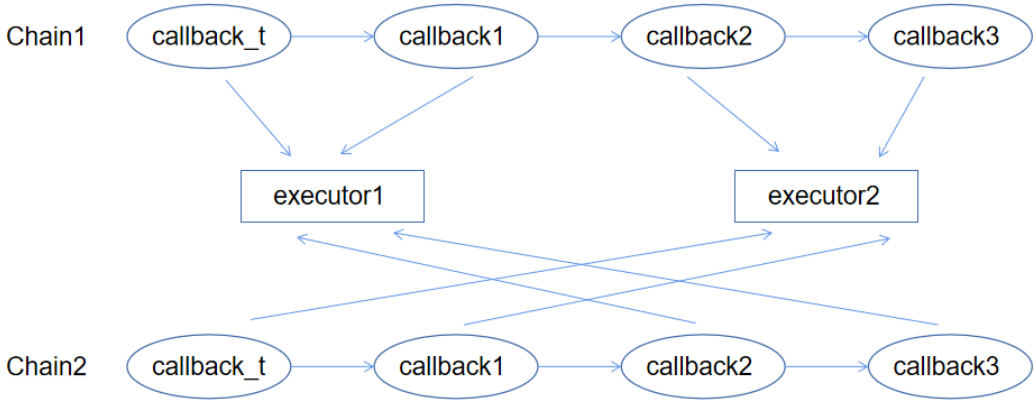

图 4.9　Chain 的结构图以及 callback 的绑定策略

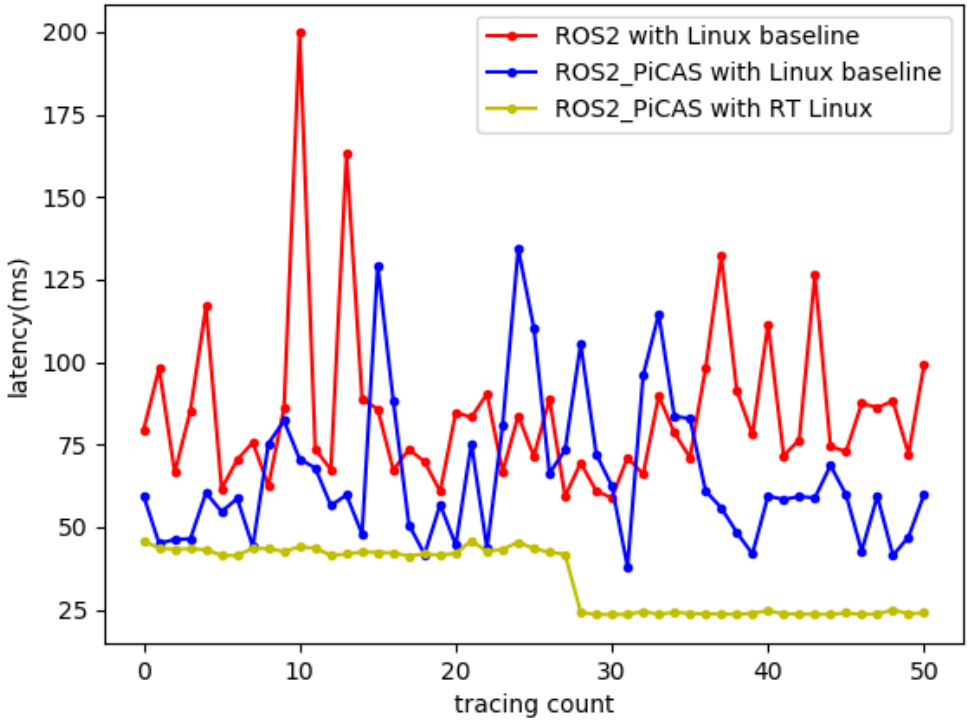

图 4.10　Chain1 的端到端时延

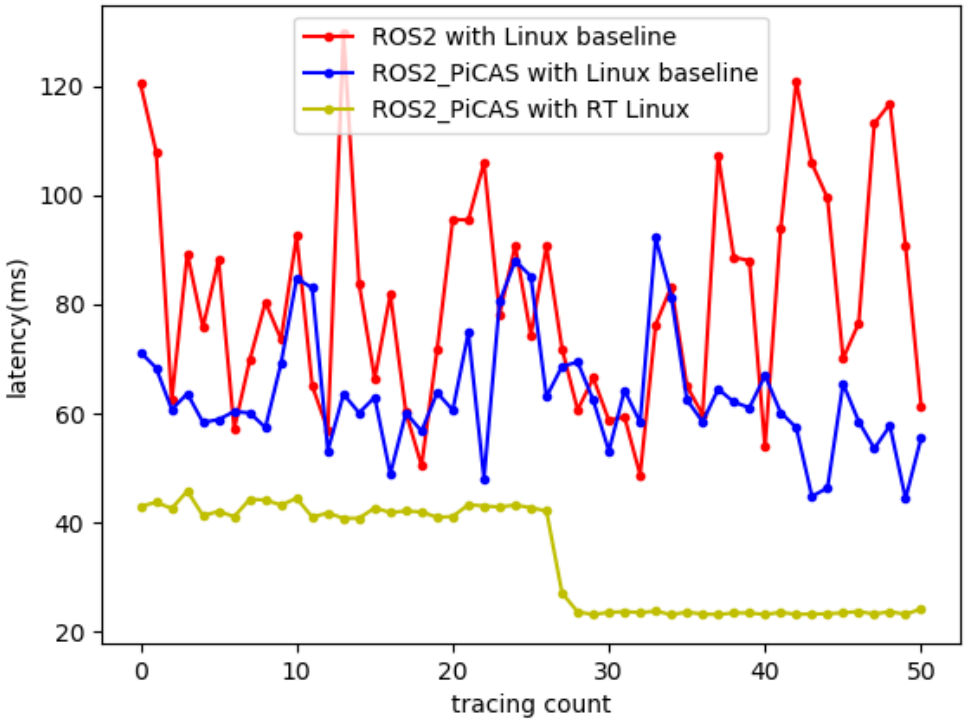

图 4.11　Chain2 的端到端时延

可以看出，ROS2_PiCAS 提升了实时性，Preempt RT 补丁大幅提升了稳定性与实时性。这是由于 ROS2_PiCAS 规定了有依赖关系 callback 的优先级，以及 executor 的实时优先级，避免了一些 chains 在系统中自干扰的发生。Preempt RT 补丁避免了被各种其他中断程序持续打断的情况，提高了响应速度。

第五章　总结与展望

本文通过阐述 Linux baseline 的实时性缺陷以及 Linux 的内核实时补丁 Preempt RT 的原理、机器人操作系统的调度体系结构的实时性缺陷以及引入了一种全新的调度体系结构 ROS2_PiCAS，总结了增强机器人系统实时性的一种解决方案。最后以实验表明，在启用 Linux 内核实时补丁以及改进 ROS2 的调度体系结构相结合的解决方案确实能大幅提高机器人系统的实时性能。本文总结的解决方案有助于各种机器人系统应用的运行，例如提高室内自动送餐机器人的稳定性，缩短自动驾驶系统的事件响应时间来提高安全性等等，在未来将拥有更广泛的应用场景。

参 考 文 献

[1] ARTHUR S, EMDE C, MC GUIRE N. Assessment of the realtime preemption patches (rt-preempt) and their impact on the general purpose performance of the system[C]//Proceedings of the 9th Real-Time Linux Workshop. 2007.

[2] DIETRICH S T, WALKER D. The evolution of real-time linux[C]//7th RTL Workshop. Citeseer, 2005.

[3] 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.

[4] 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.

致　谢

这篇毕业论文能以这样的完成形态呈现，有一些人的帮助不可忽视，在深表谢意。

首先是我的导师张燕咏教授。在研究学习期间，我有幸得到了我的导师张燕咏教授的指导，张老师深厚的学术功底和敏锐的科学洞察力在耳濡目染中影响了我很多。另外，她给予了学生充分的选择空间，一直教育我们以兴趣为导向进行科研，所以我能在充分的调研后选择系统作为研究方向。最重要的是，张老师一直强调沟通的重要性。我本来是一个内向、不太会沟通的人，在张老师一再强调沟通的重要性之后，我慢慢的尝试着提问，表明我的诉求，明确对方表达的内容，而不是只会说“好的”。我体会到了沟通带来的便利性，它不仅可以让人避免走弯路，大幅节省时间，还可以加深彼此的了解。

接着是同实验室的师兄祝含颀学长。他给予了这篇论文的很多思路，给了我具体的研究方向与建议。另一方面，他不仅授我以鱼，还授我以渔。他教给我我这个学术门外汉很多学术研究的基本方法，如何精确检索论文，如何写综述，如何搜索自己的问题等等。这些方法论在学术道路上将给予我很大的便利，非常感谢师兄的耐心教导。

其次是我实习的同事和领导们。他们虽然没有给予我学术上的帮助，但是在与人处事和工作呈现方面给了我很多宝贵建议。我在实习前一直处于象牙塔内的自我封闭状态，来这里实习之后我观察并学习到了真正工作的人如何相处。他们还教育我汇报应该如何更加直观、易懂，如何表达的清晰、有条理，这可以说改变了我的思维模式。从前我一直处于一片以自我为中心的混沌中，来这里后，我慢慢学着如何让别人听懂我的想法，如何处理人与人之间基本的关系，如何保持消息的同步等等。这些东西都使我终生受用。

最后，我最想感谢的是我自己。准备这篇论文开始时，我处于心理上的最低谷。但我争取到了导师和校外一家公司的成立联合实验室的会议的旁听机会，恰巧在会上我听到了联合实验室在杭州需要实习生的需求，会后我跟师兄了我想去实习的诉求，后来导师直接推荐我去实习，于是这家公司成了我第一家入职的企业。这都是我个人争取的结果。接着我孤身一人来到没有一个熟人的杭州，自己挑选、办理住所，添置生活用品，没有寻求任何人的帮助。入职以后我默默观察，学习身边同事的处事方式，直面高层领导和技术大牛，这些都拓宽了我的眼

界，沉淀了我的心性。其实这对于一个象牙塔中的自闭学生是非常大的冲击，但我都默默一个人扛下了一切，我知道这些痛苦是成长中所必须经历的。在写这篇论文之际，我最想要感谢的就是我自己，虽然还有非常大的进步空间，但你已经改变很多了。

在致谢写到尾声的时候，我的本科四年也将要结束了。这四年对我而言，既短暂又漫长。本科四年是我成长的四年，在这四年里，我一遍遍的建立、否定三观，一遍遍的探寻自我和学术的深度，一遍遍的迷茫又清醒。四年前我只是一个懵懂的孩子，意气风发，却又不知对自己负责，只会一头莽撞的向前。现在的我依旧眼前一片浓烟，但我已经知道如何拿稳我的探照灯，并且依稀有了一些方向。我正满怀期待的步入人生的下一个阶段，而这四年的经历将是我一生中极为宝贵的珍藏。

2022 年 5 月