ReL4：高性能异步微内核设计与实现

***

**** 年 * 月

ReL4：高性能异步微内核设计与实现

***

北京理工大学

中图分类号：TP316.7

UDC分类号：004.45

★ 特别类型
□ 交叉研究方向
□ 政府项目留学生

ReL4：高性能异步微内核设计与实现

作 者 姓 名 ***

学 院 名 称 计算机学院

指 导 教 师 ***

答辩委员会主席 ***

申 请 学 位 工学硕士

学 科 / 类 别 计算机科学与技术

学 位 授 予 单 位 北京理工大学

论 文 答 辩 日 期 **** 年 * 月

ReL4: Design and Implementation of High-performance Asynchronous Microkernel

Candidate Name:	***
School or Department:	School of Computer Science & Technology
Faculty Mentor:	***
Chair, Thesis Committee:	***
Degree Applied:	Master of Science in Engineering
Major:	Computer Science & Technology
Degree by:	Beijing Institute of Technology
The Date of Defence:	*, ****

研究成果声明

本人郑重声明：所提交的学位论文是我本人在指导教师的指导下独立完成的研究成果。文中所撰写内容符合以下学术规范（请勾选）：

□ 论文综述遵循“适当引用”的规范，全部引用的内容不超过50%。

□ 论文中的研究数据及结果不存在篡改、剽窃、抄袭、伪造等学术不端行为，并愿意承担因学术不端行为所带来的一切后果和法律责任。

□ 文中依法引用他人的成果，均已做出明确标注或得到许可。

□ 论文内容未包含法律意义上已属于他人的任何形式的研究成果，也不包含本人已用于其他学位申请的论文或成果。

□ 与本人一同工作的合作者对此研究工作所做的任何贡献均已在学位论文中作了明确的说明并表示了谢意。

特此声明。

签　　名：　　日期：

关于学位论文使用权的说明

本人完全了解北京理工大学有关保管、使用学位论文的规定，其中包括：

① 学校有权保管、并向有关部门送交学位论文的原件与复印件；

② 学校可以采用影印、缩印或其它复制手段复制并保存学位论文；

③ 学校可允许学位论文被查阅或借阅；

④ 学校可以学术交流为目的，复制赠送和交换学位论文；

⑤ 学校可以公布学位论文的全部或部分内容（保密学位论文在解密后遵守此规定）。

签　　名：　　日期：

导师签名：　　日期：

摘要

微内核将大部分系统服务运行在用户空间，相比于宏内核有更好的稳定性、扩展性和内核安全性，在微内核系统中，应用程序通过进程间通信 (IPC) 而非系统调用来请求服务，频繁的 IPC 造成的特权级切换将产生巨大开销，成为了系统的性能瓶颈。以 seL4 为代表的现代微内核将同步 IPC 作为主要通信手段，并以异步通知机制作为辅助手段来提升系统并发度，这些机制在一定程度上减少了 IPC 的开销，提升了系统性能，然而它们在设计上仍有三点不足：1）在支持同步 IPC 的情况下冗余地支持了异步通知机制，这违反了内核最小化原则；2）通知机制依赖内核的转发，会造成大量的特权级切换；3）系统调用和同步 IPC 会导致无关请求顺序执行，无法充分利用硬件资源。

针对以上三点缺陷，本文设计并实现了 ReL4——一套异步化的高性能微内核架构，它在兼容 seL4 基本系统调用的前提下，将同步 IPC 从内核中移除，采用纯异步的用户态 IPC 框架进行通信，保证了内核最小化原则；同时，ReL4 基于用户态中断的硬件支持，设计了无需内核转发的异步通知机制（U-notification），避免了通知机制造成的大量特权级切换；最后，ReL4 通过异步运行时来实现异步 IPC 和异步系统调用，避免了无关 IPC 和系统调用请求的顺序执行，在充分利用硬件资源，提升系统并发度的同时，提升系统易用性，并利用感知任务的中断控制器（TAIC），将任务调度的部分功能卸载到硬件，进一步提升低并发场景的性能。

本文在 FPGA 上实现了 ReL4 的原型系统，使用 seL4test 测试框架证明了 ReL4 的功能正确性和对 seL4 的兼容性，同时将 U-notification、异步 IPC 和异步系统调用分别与 seL4 进行了对比测试，并评估了真实的 TCP Server 在 ReL4 的性能表现。测试结果表明，相比于 seL4，ReL4 能够大幅减少系统中特权级切换的次数，在低并发场景下有着接近的性能，在高并发场景下拥有更卓越的表现，从而证明了 ReL4 架构的可行性和优越性。

关键词：微内核；异步；进程间通信；用户态中断

Abstract

Modern microkernels, which execute most system services in user space, offer superior stability, scalability, and kernel security compared to monolithic kernels. In such systems, applications request services via Inter-Process Communication (IPC) rather than system calls. However, frequent IPC-induced privilege level transitions incur significant overhead, becoming a performance bottleneck. Contemporary microkernels like seL4 address this by adopting synchronous IPC as the primary communication mechanism, supplemented by asynchronous notification mechanisms to enhance concurrency. While these approaches reduce IPC overhead and improve performance to some extent, they still exhibit three key limitations: (1) Redundant support for asynchronous notifications alongside synchronous IPC violates the principle of kernel minimization; (2) Notification mechanisms rely on kernel-mediated forwarding, causing excessive privilege level transitions; (3) System calls and synchronous IPC enforce sequential execution of unrelated requests, underutilizing hardware resources.

To address these deficiencies, we design and implement ReL4—a high-performance asynchronous microkernel architecture. While maintaining compatibility with seL4's essential system calls, ReL4 removes synchronous IPC from the kernel, employing a purely asynchronous user-space IPC framework to uphold the principle of kernel minimization. Additionally, leveraging hardware support for user-space interrupts, ReL4 introduces U-notification, an asynchronous notification mechanism that eliminates kernel-mediated forwarding and the associated privilege level transitions. Furthermore, ReL4 implements an asynchronous runtime to enable concurrent execution of asynchronous IPC and system calls, preventing the serialization of unrelated requests. This design not only enhances resource utilization and concurrency but also improves usability. To further optimize low-concurrency scenarios, ReL4 integrates the Task-Aware Interrupt Controller (TAIC), offloading parts of task scheduling to hardware.

We prototype ReL4 on an FPGA platform and validate its functionality and seL4 compatibility using the seL4test framework. Comparative evaluations of U-notification, asynchronous IPC, and asynchronous system calls against seL4 demonstrate significant reductions in privilege level transitions. Performance measurements show that ReL4 achieves compara-

ble performance to seL4 in low-concurrency scenarios and outperforms it in high-concurrency settings. These results confirm the feasibility and superiority of the ReL4 architecture.

Key Words: microkernel; asynchronous; inter-process communication; user-mode interrupt

目录

插图

表格

第 1 章　绪论

1.1　课题研究的背景和意义

随着科技的发展，微内核广泛应用于工控系统、嵌入式系统等领域[1]。相比于宏内核，微内核将内存管理、设备驱动、文件系统等与核心功能分离，运行在用户空间，这种隔离机制使得单个服务的故障不会直接影响到内核和其他服务，从而提升了系统的整体稳定性[2]；微内核通过精简核心功能，减少攻击面，从而提升内核安全性[3]；此外，模块化的设计使得系统更易于维护和升级[4]。在微内核系统中，应用程序通过进程间通信 (IPC) 而非系统调用来请求服务，这或许能够满足早期性能不敏感的软件需求，然而在对软件性能有着更高要求的今天，频繁 IPC 造成的特权级切换将产生巨大开销，成为系统的性能瓶颈[5]。

30 年前 Liedtke 提出的 L4[6] 重新设计了微内核系统，通过组合系统调用、快速路径、消息寄存器等优化手段，从硬件层到软件层对 IPC 进行了系统性优化，证明了微内核的 IPC 也可以很快[7]，之后以 seL4[8] 为代表的现代微内核的 IPC 框架也基本延续了 L4 的设计理念，以同步 IPC 作为主要的通信方式。然而同步 IPC 迫使单线程中上下文无关的请求以顺序的形式执行，系统只能通过多线程实现并发，而为了更好地利用硬件资源，现代微内核大多引入异步的通知机制来简化并发程序设计，提升多核的利用率，这违反了微内核的最小化设计原则，增加了内核的复杂性。

而随着软件复杂性的提升，用户希望系统级软件如数据库管理系统、网络服务器等能够快速处理大量系统调用和 IPC[9]，而微内核将操作系统的大部分服务（如网络协议栈、文件系统等）移到用户态，从而使得 IPC 数量和频率激增，特权级切换成为性能瓶颈。此外，新出现的硬件漏洞如 Meltdown[10] 和 Spectre[11] 漏洞促使内核使用 KPTI 补丁[12] 来分离用户程序和内核的页表，进一步增加了陷入内核的开销。最后，现代微内核的外设驱动往往存在于用户态，外设中断被转化为异步通知，需要用户态驱动主动陷入内核来进行接收，这在一定程度上成为了外设驱动的性能瓶颈[13]。综上所述，由 IPC 和通知机制引起的特权级切换已经成为制约系统性能的主要因素。

本文提出 ReL4，一个用 Rust 编写的高性能异步微内核，它将同步 IPC 从内核中移除，基于用户态中断技术设计了无需陷入内核的 U-notification 机制，在兼容 capability 机制的基础上改造微内核的通知机制，并利用改造后的 U-notification 和异步化编程

设计和实现了一套绕过内核的异步 IPC 和异步系统调用框架。ReL4 在设计理念上将内核最小化原则贯彻得更加彻底，并通过软硬件协同的方式进一步提升微内核的 IPC 性能，为下一代微内核的发展指出一个可能的方向。

1.2 国内外研究现状及发展趋势

1.2.1 微内核 IPC 的发展现状

现代微内核的进程间通信（IPC）优化研究可追溯至 Liedtke 提出的 L4 微内核架构。针对早期微内核 IPC 存在的性能瓶颈问题，L4 从硬件抽象层、系统架构设计和软件接口规范等多个维度进行了系统性重构。这些优化措施主要可归纳为内核执行路径优化和上下文切换优化两个关键方向。

在内核执行路径优化方面，L4 通过物理消息寄存器来传递短消息，有效规避了短消息场景下的内存拷贝开销。然而，随着存储器访问性能的持续提升，这种零拷贝优化带来的相对收益逐渐减弱。更为关键的是，物理寄存器的硬件依赖性导致平台移植困难，并可能干扰编译器的寄存器分配优化[14]。这一局限性促使现代微内核普遍转向虚拟消息寄存器方案。对于长消息传输，L4 采用临时内存映射机制避免数据拷贝，但该设计增加了内核处理缺页异常的复杂度[14]，因而被后续系统所摒弃。值得注意的是，L4 针对高频 IPC 场景设计了专门的快速路径（fast-path）机制[15]（如图1.1所示），通过简化参数解析和任务调度流程提升性能。然而，该优化对消息长度和任务优先级等参数有严格约束，且难以扩展至多核环境。

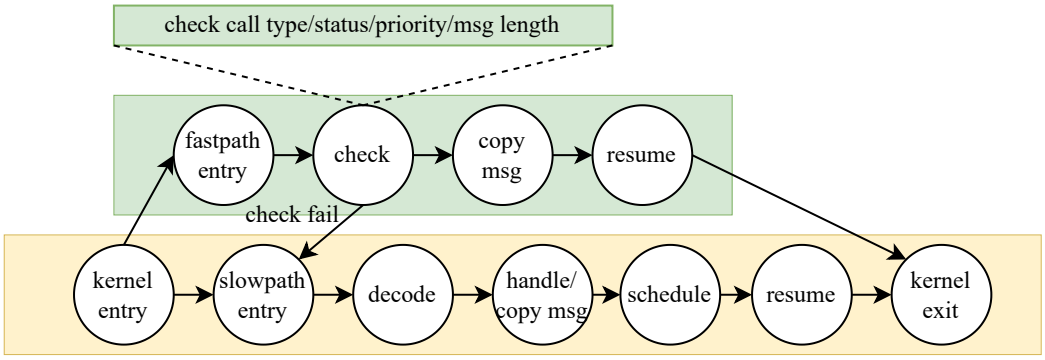

图 1.1 fast-path 优化示意图

在特权级切换优化方面，L4 进行了多项开创性探索。虽然其物理消息寄存器方案确实降低了上下文切换开销，但前文所述的局限性最终导致该设计被虚拟寄存器

方案取代[14]。L4 的其中一个创新性贡献在于发现了 IPC 通信中普遍存在的客户端-服务器模式特征，并据此设计了组合式系统调用接口：将 Send+Reply 整合为 Call 操作，将 Reply+Recv 合并为 ReplyRecv 操作。这种设计显著减少了特权级切换频率，至今仍是现代微内核的标准优化手段。此外，L4 采用地址空间标识符（ASID）机制来缓解 TLB 冲刷问题[16]，通过维护进程专属的快表项降低了上下文切换带来的性能损耗。然而，这些优化仍无法完全消除特权级切换导致的 TLB 污染和缓存失效问题[17]。

除此之外，L4 仅支持同步 IPC，对于多核架构，同步 IPC 会导致服务调用被顺序执行，导致资源浪费[18]。其次，同步 IPC 强制用户态以多线程的形式处理并发请求，导致了线程同步的复杂性。现代微内核在内核中引入异步通知机制，简化并发编程模型，却使得内核功能冗余，违反了内核最小化原则。

总而言之，现代微内核在单核环境下的 IPC 内核路径上的优化已经较为完善，在最理想的情况下仅需要两次特权级切换，然而对多核环境下，由于需要核间中断，IPC 无法进入快速路径，导致多核下的 IPC 内核路径依旧冗长。而现代微内核在特权级切换的优化方面仍然停留缓解的层面上，导致特权级切换会成为 IPC 的性能瓶颈。

1.2.2 特权级切换

特权级切换带来的性能开销可分解为直接和间接两个部分[19]。直接开销主要体现为上下文保存与恢复所需的额外指令执行周期，而间接开销则源于地址空间切换导致的缓存污染效应[20]，这种效应会显著降低后续指令的执行效率。如表1.1所示，基于 FPGA 平台的实测数据表明，在 seL4 微内核中执行一次 Call IPC 操作时，地址空间切换所产生的时间开销占比最高，其次是上下文切换和 fast-path 检查过程。特别值得注意的是，当 fast-path 检查未通过时，系统将转入 slow-path 处理流程，该流程涉及更复杂的消息解码和任务调度机制，会进一步加剧 IPC 性能的下降。针对这一关键性能瓶颈问题，学术界和工业界已从硬件架构优化和软件算法改进等多个维度展开了广泛而深入的研究探索。

从硬件出发的角度出发，大多数工作通过设计特殊的硬件或者特殊的指令来绕过内核实现 IPC。如 SkyBridge[21] 允许进程在 IPC 中直接切换到目标进程的虚拟地址空间并调用目标函数，它通过精心设计一个 Root Kernel 提供虚拟化的功能，通过 VMFUNC 地址空间的直接切换[22]，并通过其他一系列软件手段来保证安全性，但这种方案仅适用于虚拟化环境中。XPC[23] 则直接使用硬件来提供一个无需经过内核的

表 1.1　seL4 一次 IPC 中各操作的开销占比

操作	占比
保存和恢复上下文	14.5%
地址空间切换	59.4%
fast - path 检查	20.1%
消息拷贝	1.5%
其他	4.5%

同步功能调用，并提供一种新的空间映射机制用于调用者与被调用者之间的零拷贝消息传递，然而该方案没有相应的硬件标准，也没有一款通用的处理器对其进行支持。这些方法都基于特殊的环境或者没有标准化的硬件来实现，适用范围有限。

从软件出发的角度出发，相关工作主要分为两类：第一类方法通过将用户态和内核态的功能扁平化来减少内核与用户态的切换开销，如 unikernel[24–26] 将所有用户态代码都映射到内核态执行，Userspace Bypass[27] 通过动态二进制分析将两个系统调用之间的用户态代码移入内核态执行，从而减少陷入内核的次数，kernel bypass[28,29] 则通过将硬件驱动（传统内核的功能）移入用户态，从而减少上下文的切换。这些方法要么需要特殊的硬件支持，要么难以与微内核的设计理念兼容。第二类方法则是允许用户空间对多个系统调用请求排队，并通过一次提交将他们注册给内核，如 FlexSC[30] 通过在用户态设计一个用户态线程的运行时，将用户态线程发起的系统调用自动收集，然后陷入内核态进行批量执行。该方法虽然可以有效的减少陷入内核的次数，但如何设置提交的时机难以把握，过短的提交间隔将导致切换次数增加，过长的提交间隔则会导致 CPU 空转。

虽然现有工作难以广泛且有效地应用到微内核中，但他们的思路值得借鉴，他们的缺陷驱使研究者去寻求更好的方案。在硬件方面，一种新型的硬件技术方案——用户态中断[31,32] 逐渐被各个硬件平台（x86，RISC-V）采纳，它通过在 CPU 中新增中断代理机制和用户态中断的状态寄存器，当中断代理机制检测到状态寄存器发生变化时，会将中断以硬件转发的形式传递给用户态程序，从而绕过内核。该硬件方案已经在 Sapphire Rapids x86 处理器上和 RISCV 的 N 扩展中有了一定的支持，适用范围更加广泛。而在软件方面，异步被广泛用于请求合并和开销均摊，传统类 Unix 系统提供的类似 select IO 多路复用接口相对简陋，迫使用户态代码采用事件分发的编程范式来处理异步事件，代码相对复杂，可读性较弱。而新兴的 Rust[33,34] 语言对异步有着良好的支持，其零成本抽象的设计也让它作为系统编程语言有着强大的竞争力。使用 Rust 进行内核和用户态基础库的开发，可以更好地对异步接口进行抽象，改善接口的

表 1.2　国内外研究现状汇总

优化方法	详细分类	实例	缺点
减少内核路径	临时地址映射	[6]	上下文切换开销已经成为性能瓶颈
	快速路径	[6,8,35,36]	
	消息寄存器		
减少上下文切换开销	消息寄存器	[6,8,35,36]	
	组合系统调用		无法从根本上消除切换开销
	ASID 机制	[8]	
	统一地址空间	[24–29]	与微内核设计理念相悖，无法有效地实施到微内核中
	批量系统调用	[30]	
硬件优化	虚拟化指令	[21]	仅适用于虚拟化环境
	直接硬件辅助	[23]	没有硬件标准，没有通用硬件的支持
		[37,38]	—

易用性和代码的可读性。

1.2.3　Rust 语言与异步编程机制

在现代计算机系统中，随着多任务处理和并发编程的普及[39]，程序需要处理的任务越来越复杂，任务之间的依赖关系也变得越来越松散。传统的同步机制，即任务按照严格的顺序执行，前一个任务完成后才能开始后一个任务，已经无法满足高效处理多个并发任务的需求[40]。因此，异步机制应运而生[41]，它允许任务在不阻塞其他任务的情况下执行，从而提高了系统的整体性能和响应速度。传统操作系统大多使用 C 语言作为主要语言来实现异步[42]，用于应对内核场景下的复杂场景，其虽然相对灵活，但也存在一些固有缺陷：

- 编程困难：异步编程需要在代码中处理多个任务之间的协作和同步，这增加了编程的复杂性。开发者需要仔细考虑异步任务之间的依赖关系、执行顺序以及错误处理，可能导致代码难以理解和维护。
- 调试困难：异步编程的执行流程是非线性的，这增加了调试的难度。由于异步任务可能在不同时间点触发和执行，调试时需要跟踪多个任务的状态和交互，使得调试过程更加繁琐和耗时。
- 代码可读性下降：异步编程通常涉及回调函数、事件监听等结构，这些结构可能导致代码的可读性下降。回调函数的嵌套使用（即“回调地狱”）可能使代码结构变得混乱，难以阅读和理解。
- 代码可维护性下降：异步编程中的任务依赖关系和同步机制可能随着代码的发展而变得更加复杂，这增加了代码维护的难度。当需要修改或添加新的异步任务时，开发者需要仔细考虑对现有代码的影响，并确保新任务与现有任务之间

的正确同步和协作。

而相对高阶的语言提供了对异步的原生支持，如 C++[43] 和 Rust[44]，在语言层面对异步编程提供了支持，不依赖操作系统以及标准库，有着良好的移植性和兼容性。本节我们将介绍 Rust 语言对异步的支持[45]。

Rust 语言的异步编程模型体现了一种系统级的创新设计，其核心思想是通过编译时状态机转换来实现高效的异步执行[46]。该模型建立在 Future 这一基础抽象之上，Future trait 定义的 poll 机制采用了一种独特的延迟执行模式，这种设计使得异步任务能够被精确控制，而非传统的事件回调模式[47]。值得注意的是，Rust 的异步机制通过编译器生成的状态机转换，在保证零成本抽象的同时[48]，实现了与手动优化代码相当的性能表现。

在运行时支持方面，Rust 采用了模块化的设计哲学[49]。异步运行时被明确划分为 Executor 和 Reactor 两个核心组件[50]，这种关注点分离的设计带来了显著的架构优势。Executor 负责调度逻辑，而 Reactor 处理 I/O 事件通知，二者的协同工作通过高效的 wake 机制实现。特别值得关注的是，Rust 的标准库刻意不绑定特定的运行时实现[51]，这种设计决策赋予了开发者根据应用场景选择最优运行时的灵活性。从系统编程的角度看，这种模块化设计使得异步 Rust 能够适应从嵌入式系统到分布式服务的各种应用场景。

语言层面的 async/await 语法为异步编程提供了重要的工程实践支持[45]。这一语法糖不仅大幅提升了代码的可读性和可维护性，更重要的是，它使得编译器能够进行深度的优化。通过将异步控制流转换为状态机，Rust 实现了与手动编写回调代码相当的性能，同时避免了回调地狱等常见的异步编程陷阱[47]。从类型系统的角度来看，async 函数返回的 Future 保持了严格的类型约束，这使得编译器能够在编译期捕获大量潜在的逻辑错误，显著提升了异步代码的可靠性。

此外，Rust 语言的设计哲学与微内核架构在系统安全性和可靠性方面展现出深刻的协同效应。从内存管理机制来看，Rust 的所有权系统通过编译时的严格检查[33]，为微内核的关键组件提供了静态的内存安全保障。这种编译期验证机制与微内核最小化特权域的设计原则高度契合，使得内核开发者能够在保证性能的前提下，有效预防各类内存安全漏洞。

基于上述工程实践考量，本文选择 Rust 作为 ReL4 微内核的主要实现语言。这一选择不仅能够充分利用 Rust 在系统编程领域的独特优势，更重要的是，Rust 的现代

化语言特性为构建安全、高效的异步微内核提供了更加理想的开发工具链。

1.2.4 用户态中断与 TAIC 加速器

现代处理器通过多特权级设计实现安全隔离[52]，但特权级切换带来的性能开销已成为系统优化的关键瓶颈。传统的中断处理机制依赖于内核介入[42]，导致频繁的特权级切换和上下文保存恢复操作，不仅产生直接的指令执行开销，还会因页表切换和缓存失效引入显著的间接性能损失[20]。

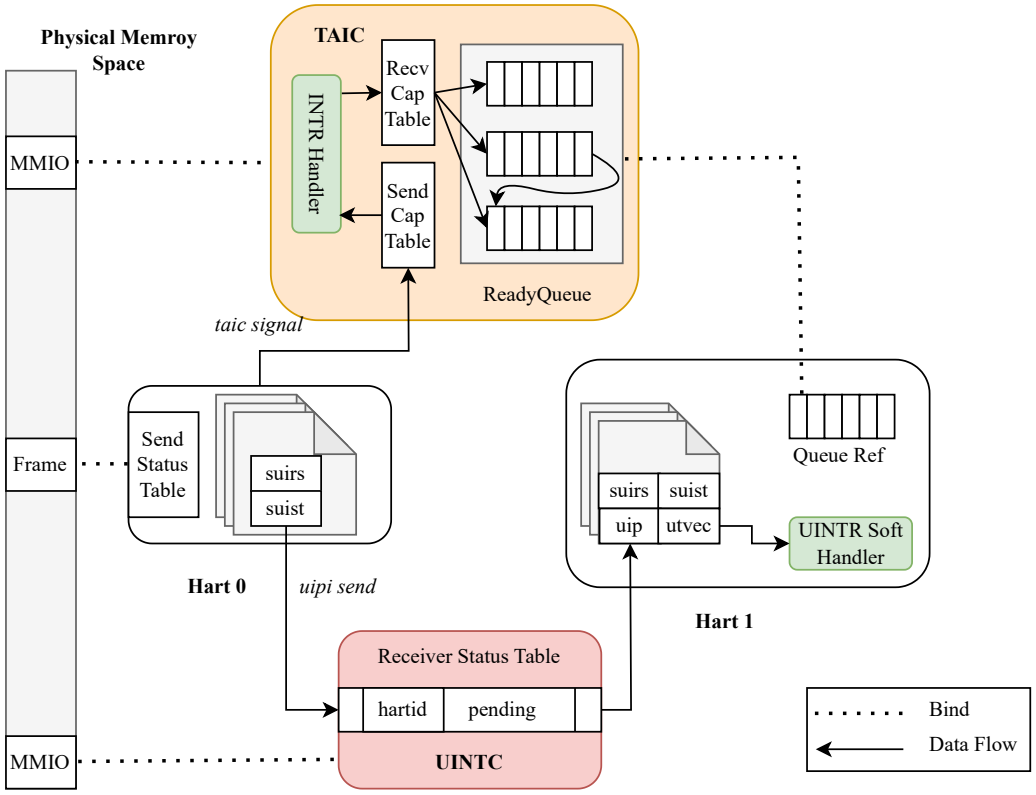

图 1.2 用户态中断和 TAIC 加速器的示意图

为降低这一开销，用户态中断（User-Level Interrupt）[37] 机制应运而生。该技术通过在 CPU 中引入中断代理和专用状态寄存器，使中断能够直接在用户态处理，完全绕过内核参与。以 RISC-V N 扩展的实现为例，如1.2所示，用户态中断的核心创新在于将中断状态管理分为操作系统维护的发送状态表和硬件控制器（UINTC）管理的接收状态表。当触发中断时，硬件通过专用指令自动完成状态查询和处理器间中断传递，最终直接跳转到用户态注册的处理函数执行。这一设计不仅消除了内核转发的软

件开销，还保持了良好的缓存局部性。

在 UINTC 基础上的进一步优化催生了任务感知中断控制器（TAIC）[53]。与基础用户态中断相比，TAIC 的创新性体现在两个方面：首先，它将所有状态表统一交由硬件管理，通过 MMIO 寄存器访问替代内存查表操作，减少了中断发送阶段的内存访问延迟；其次，该控制器深度整合了任务调度功能，能够在中断到达时自动将预注册的处理任务加入硬件就绪队列，完全消除了软件唤醒的开销。这种硬件-软件协同设计使得中断处理流程更加高效，在保持用户态执行优势的同时，进一步减少了 CPU 执行流的打断频率，为实时性要求高的应用场景提供了更优的解决方案。

从系统架构演进的角度看，从传统内核中断到用户态中断，再到任务感知中断控制器的发展，体现了中断处理机制从软件主导到硬件加速的转变趋势[54]。这种转变不仅降低了中断延迟，更重要的是通过硬件原语支持，为构建更高效的异步任务处理框架奠定了基础。

1.3 主要研究内容

本文以免费开源的 seL4 的 IPC 系统主要研究对象，介绍了 seL4 中 IPC 的设计细节以及相关设计缺陷，针对 seL4 的相关缺陷，基于用户态中断，给出了高性能异步微内核的整体设计方案，包含了无需内核转发的通知机制 U-notification、异步 IPC 与异步系统调用，并基于 Rust 语言在 RISC-V 平台实现了 ReL4 原型系统。在 FPGA 上将 U-notification、异步 IPC 和异步系统调用分别与 seL4 进行了对比测试，并评估了 ReL4 在真实的 TCP Server Benchmark 上的性能表现。论文内容结构安排如下：

第 1 章，绪论部分，阐述了本课题的研究背景及意义，国内外研究现状和发展趋势，最后概述了本课题的研究内容和对内容的组织结构。

第 2 章，介绍了 seL4 系统的发展历史和主要特征，并对 seL4 中的基本概念以及 IPC 设计进行了系统性阐述。

第 3 章，系统设计部分。首先介绍了 ReL4 与 seL4 的关系，ReL4 的特点以及设计目标，然后介绍了 ReL4 如何对系统中各个部分的通知机制进行支持和优化，最后介绍了用于保证兼容性和提升易用性的异步运行时的组成部分和设计思路。

第 4 章，系统实现部分。主要介绍了新增的系统调用，以及基于 U-notification 的异步运行时如何实现异步 IPC 和异步系统调用，并讨论了 ReL4 在通知机制、IPC 机制和系统调用中对 seL4 的兼容程度。

第 5 章，系统评估部分。首先介绍了 ReL4 的测试方案，然后针对 ReL4 的各个机制进行了性能评估和分析，最后测试了 ReL4 在真实的 TCP Server Benchmark 上的性能表现。

第 6 章，总结与展望，对本次设计和论文工作进行了总结，并提出了进一步的研究方向与对未来工作的展望。

第 2 章　seL4 介绍

2.1　seL4 的发展历史及主要特征

seL4 是一款具有创新性和里程碑意义的微内核[8]。seL4 项目始于 2006 年的澳大利亚悉尼大学，其目标是创建一个经过形式化验证的微内核，从而确保内核的安全性和可靠性。在 2009 年，seL4 正式发布了针对 arm 11 处理器的功能正确性的形式化证明[8]，是全球首个经过完整形式化验证的微内核。在 2014 年，seL4 正式开源，得到了来自开源社区的广泛关注，seL4 的生态蓬勃发展。在接下来的几年里，seL4 陆续完成了对不同 CPU 和不同指令集架构的验证和支持、虚拟化支持[55] 等，并在应用生态领域有了丰富的支持。

seL4 的设计遵循一下几个原则[56]：

- Verification：截止目前（2025 年 3 月），seL4 依然是第一个经过形式化验证的内核，形式化验证对 seL4 是个坚持不懈的努力目标，为了验证方便，禁止在内核里并发处理，不允许在内核态的大部分场景里再次发生中断。
- Minimality：一方面最小化原则是 L4 家族的根本设计理念，另一方面，最小化也是方便 seL4 做形式化验证的重要条件，seL4 内核除了中断控制器、定时器、MMU 相关的一点硬件驱动代码，其它驱动都在用户空间运行。
- Policy freedom：seL4 对于大部分资源分配策略都移到了用户态进行定制，通过 Capabiltiy 进行管理。
- Performance：虽然极度关注安全、可形式化验证，seL4 着重对热点路径的优化，因此依然有着突出的性能优势。
- Security：seL4 在安全性设计上遵循最小权限原则（Least privilege），通过 capability 机制来保证任何组件只拥有完成其工作所需的权限。

2.2　seL4 的基本组成

seL4 采用严格的最小化设计原则，将传统操作系统内核的功能进行解耦和重构，形成了具有高度安全性的微内核架构。如图2.1所示，该架构通过精心的功能划分，在内核空间仅保留最基础的硬件抽象层和核心机制，而将传统操作系统服务移至用户空

间实现。

在内核设计上，seL4 内核主要实现了五个关键子系统：虚拟内存管理负责地址空间隔离和保护，进程间通信提供基于消息传递的交互机制，通知机制处理异步事件传递，任务调度管理系统执行流，中断管理则负责硬件中断的初始分发。这些子系统通过统一的能力（Capability）机制进行安全管控[57]，所有资源的访问都必须经过严格的能力验证，从而构建起系统的安全基础。

用户空间的设计体现了 seL4 的架构创新，它将传统内核中的设备驱动、网络协议栈等系统服务完全移出内核，作为普通用户进程运行。应用程序通过同步 IPC 机制请求服务，而硬件中断和软件信号则经过内核的初步处理后，通过异步通知机制传递给相应的用户态驱动处理。这种设计不仅大幅减小了内核的代码规模，更重要的是显著降低了系统的可信计算基（TCB）[58]，同时保持了良好的性能特性。实测表明，经过优化的 IPC 机制使得 seL4 在保持微内核安全优势的同时，系统调用性能可以达到传统宏内核的 80%[14] 以上，实现了安全性与性能的良好平衡。

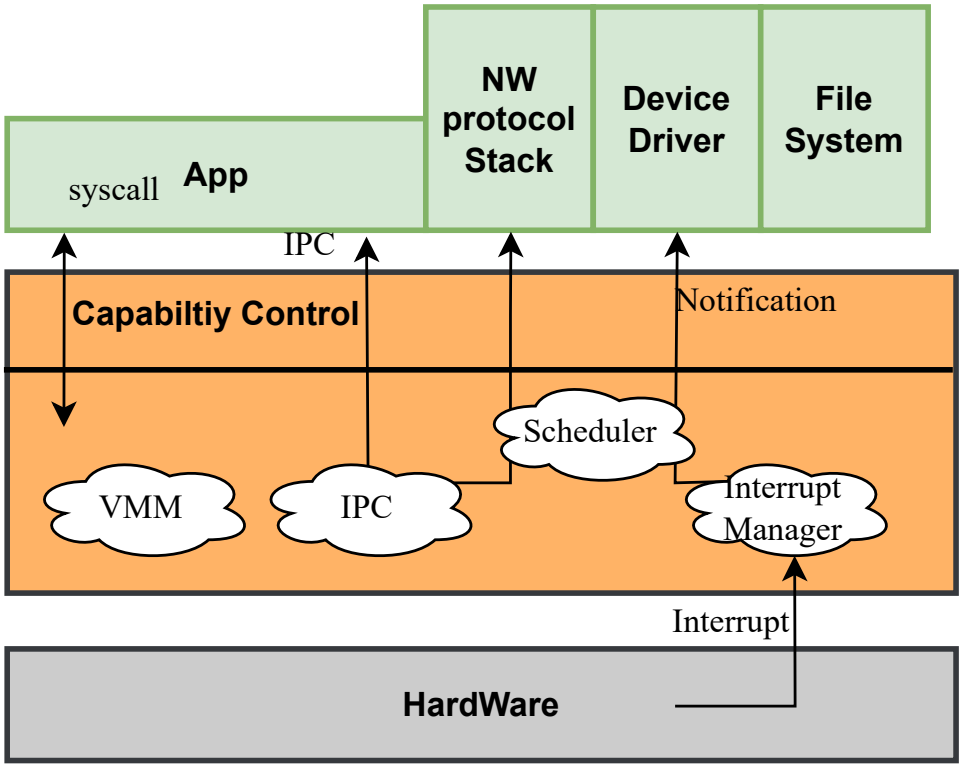

图 2.1　seL4 的系统结构图

2.2.1 内核对象与 Capability 机制

seL4 采用基于内核对象的能力（Capability）安全模型构建其访问控制体系。在该模型中，所有系统资源（如内存区域、通信端点等）都被抽象为内核对象，由内核统一管理其状态和生命周期。如表2.1所示，这些内核对象构成了系统资源的软件抽象，形成了系统状态的核心组成部分。

表 2.1 seL4 中的主要内核对象

内核对象	作用
线程控制块（TCB）	内核调度的基本单位，保存了用户任务运行所需的上下文。
能力空间（CSpace）	访问能力的集合，维护了各个内核对象的访问能力和对应权限。
地址空间（VSpace）	地址空间，维护了一段虚拟地址和物理地址的映射关系。
物理页框（Frame）	对应一个物理页，维护了物理页号和访问权限.
端点（Endpoint）	同步 IPC 的桥梁，维护了一个消息收发的状态机。
通知对象（Notification）	通知机制的桥梁，维护了一个通知状态位。
无类型对象（Untyped）	物理内存管理的承载者，通过系统调用转化为其他内核对象。

为确保系统安全性，seL4 实现了严格的访问隔离机制。用户态程序不能直接访问内核对象，而必须通过能力句柄（Capability handler）间接操作。每个能力句柄实际上是一个经过验证的引用，内核在其对应的 Capability 数据结构中维护了目标内核对象的物理地址及精确的访问权限集。当用户态发起系统调用时，内核会通过能力查找机制定位对应的 Capability，进行权限验证后才会执行相应操作。这种显式的授权机制确保了所有资源访问都必须经过严格的权限检查。

Capability 机制支持两种基本的权限传播方式：转移（Transfer）和派生（Derivation）。转移操作实现了权限的完全移交，原持有者将失去对应资源的访问权；而派生操作则允许创建具有受限权限的新 Capability，这些权限必须是原 Capability 权限的真子集。通过这种派生关系，系统内自然形成了一个层次化的能力树（Capability Tree）结构。值得注意的是，该模型提供了动态的权限回收机制——任何父节点都可以通过 revoke 操作递归地撤销其所有子节点的访问权限。这种精密的权限传播与回收机制，使得 seL4 能够构建可验证的、最小特权原则的系统安全架构。

2.2.2 内存管理架构

seL4 采用了一种创新的分离式内存管理架构，这种设计在保证系统安全性的同时，实现了管理权限的合理分配。如图2.2所示，该架构将内存管理明确划分为两个层

次：物理内存完全交由用户态管理，而虚拟内存则由内核态统一维护。这种分离式设计体现了微内核架构的最小特权原则，既降低了内核的复杂度，又为用户态提供了灵活的内存管理能力。

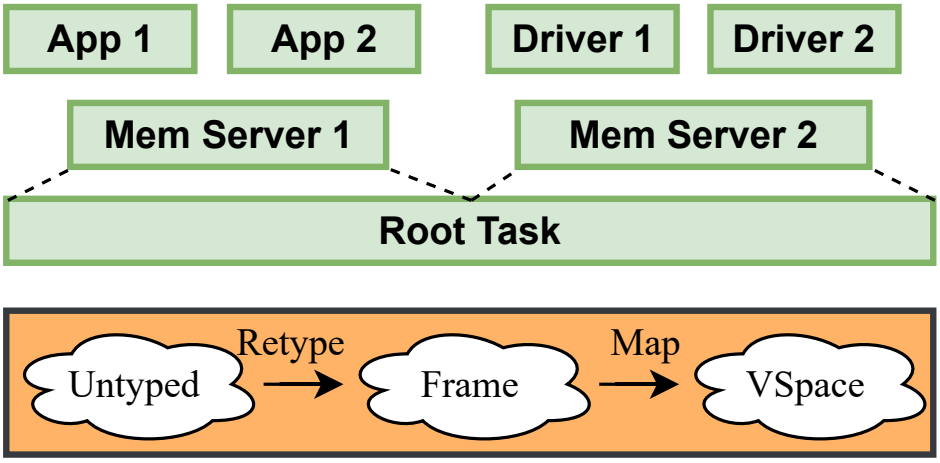

图 2.2　seL4 的内存管理

在物理内存管理方面，seL4 引入了一种独特的 Untyped 对象机制。这些对象代表了系统初始化时可用的原始物理内存区域，具有以下重要特性：

- 它们在系统启动时由内核统一创建；
- 所有 Untyped 对象的能力都被集中授予 Root Task 用户进程；
- 用户态可以通过 Retype 系统调用对这些对象进行精细化操作。

具体来说，Retype 操作支持两种转换方式：一是将大块 Untyped 对象分割为更小的 Untyped 区域，二是将其转换为特定类型的内核对象（如 Frame、CNodes 等）。这种设计形成了一个完全由用户态主导的分布式递归内存管理体系，使得物理内存资源可以根据需要动态分配给不同的用户态组件。值得注意的是，这种机制不仅实现了物理内存的灵活管理，还通过能力系统确保了内存分配的安全性。

虚拟内存管理方面，seL4 为每个进程维护一个 VSpace 内核对象，该对象作为地址空间的核心抽象，具有以下关键功能：

- 关联着进程的根页表结构；
- 通过能力机制控制着地址空间的访问权限；
- 为内存隔离提供了硬件级的保护；

用户态程序通过 map/unmap 系统调用可以动态调整 Frame 对象与虚拟地址的绑定关系，这些操作需要经过严格的能力检查。具体而言，map 操作需要调用者同时拥有目标 VSpace 和 Frame 对象的能力，且映射范围必须在 VSpace 的有效区域内。这种设计既保证了内存访问的安全性，又为用户态提供了充分的灵活性。

seL4 这种分离式内存架构显著降低了内核的复杂度，将内存管理的策略性决策交由用户态实现，同时支持多种用户态内存管理策略的共存，不同的应用程序可以采用最适合自己的内存分配方案。这种架构设计不仅是内核最小化原则的体现，同时也是系统灵活性和可扩展性的重要支持。

2.2.3 任务调度

seL4 采用基于线程的轻量级任务调度模型，其设计体现了微内核架构的简约性和灵活性。在该模型中，线程作为最基本的执行单元和调度实体，包含了完整的执行上下文和独立的能力空间。值得注意的是，seL4 对传统操作系统中的进程概念进行了弱化，通常情况下，我们将含有相同能力空间和地址空间的称为同一个进程，将进程抽象为资源分配的基本单位，因此在 seL4 中，进程只作为一个逻辑概念存在，内核的设计和实现只涉及到线程，这种设计选择使得内核实现更加精简，同时保持了足够的抽象能力来支持复杂的系统构建。

seL4 的调度器实现采用了双层次的调度策略，将优先级调度与事件驱动的抢占调度有机结合。优先级调度基于经典的固定优先级算法，当出现以下两种情况时会触发调度决策：一是当前线程耗尽分配的时间片；二是线程因等待某些事件而主动阻塞。在调度时机到来时，内核会从就绪队列中选择优先级最高的线程进行上下文切换。另一方面，抢占调度机制主要应用于同步 IPC 场景，当线程间的通信操作导致更高优先级的线程变为可执行状态时，在不违反基本调度原则的前提下，内核会立即进行任务切换以保证系统的响应性。这种混合调度策略在保证系统确定性的同时，也优化了任务间的交互性能，具体的同步 IPC 流程描述参考2.2.4

线程的状态转换图如2.3所示，当线程陷入内核时，从 Running 切换为 ReStart 状态，即候选状态，当所有检查（如地址空间、优先级校验等）都通过之后，从候选状态重新切换为 Running 状态并返回用户态，如果有更高优先级的线程存在，则会导致当前线程被抢占，转化为 Ready 状态，当检查出非法状态时，线程状态会被设置为 Inactive，而在 IPC 通信过程中，可能会产生线程阻塞，此时根据不同的阻塞对象，线

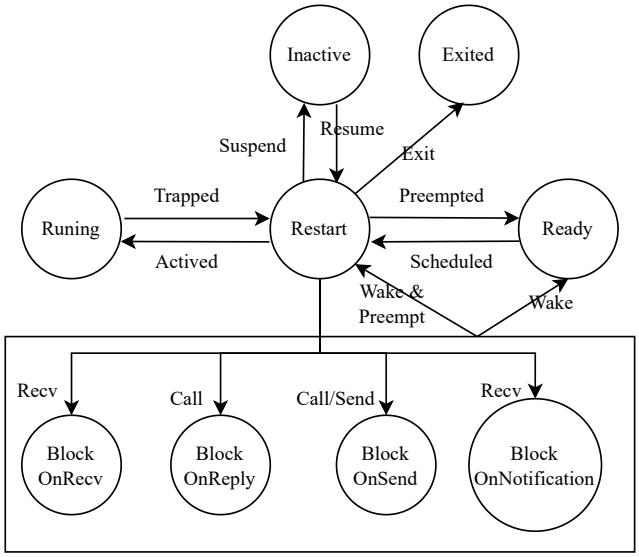

图 2.3　seL4 的线程状态转换

程状态会被设置为相应的阻塞状态。

为了满足实时系统的严格要求，seL4 还提供了 MCS（Mixed-Criticality Systems）扩展[59] 支持。该扩展通过三个关键机制来保证关键任务的实时性：内核为关键任务预留专用的时间片资源；系统支持运行时动态调整任务优先级；MCS 模式切换机制允许系统在不同关键性级别之间进行快速转换。

2.2.4　同步 IPC 和通知机制

seL4 微内核设计了一套完整的进程间通信 (IPC) 框架，该框架由同步 IPC 和异步通知机制共同构成，二者在语义和实现上存在显著差异。这种双模式设计使系统能够同时满足确定性通信和高效事件通知的需求。

同步 IPC 机制实现了严格的请求-响应语义，其核心特征在于通信双方的强同步性。如2.4所示，当客户端发起 IPC 调用时，系统会立即将调用线程阻塞，直至服务端完成请求处理并返回响应。这种同步性通过 Endpoint 内核对象实现，该对象维护着精确的状态机和阻塞队列。值得注意的是，同步 IPC 与任务调度器深度耦合，在通信过程中可能触发优先级驱动的任务切换。例如，当高优先级服务端线程被唤醒时，内核会立即执行上下文切换，这种设计保证了系统的时间确定性，使其特别适合需要严格时序保证的关键任务。

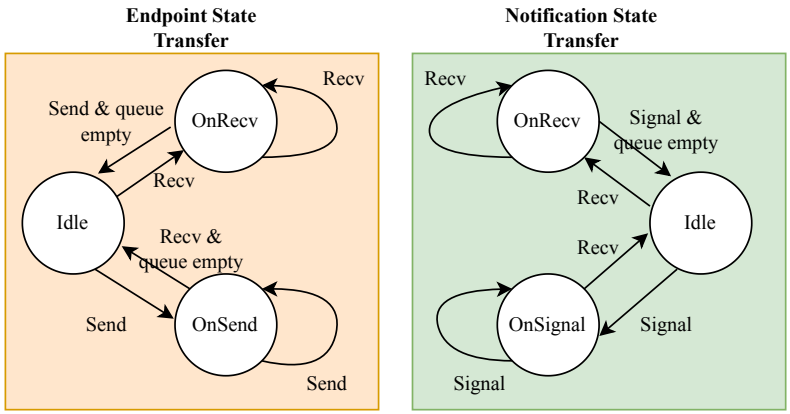

图 2.4　seL4 的 IPC 相关内核对象状态转换

相比之下，异步通知机制采用了完全不同的通信范式。该机制基于 Notification 内核对象实现，其最显著的特点是发送操作的异步性。通知发送方可以立即继续执行后续指令，而不需要等待接收方的响应。这种非阻塞特性显著提升了系统的并发处理能力，使其能够高效处理大量事件通知。然而，接收方仍然保持同步的接收模式，需要通过显式的系统调用获取通知内容。这种不对称设计在保证事件处理可靠性的同时，避免了纯异步模型可能带来的复杂性问题。

从实现层面看，两种机制的主要区别体现在三个方面：首先，同步 IPC 需要维护完整的调用上下文，而通知机制仅需管理信号位图；其次，同步 IPC 涉及双向数据传输，而通知主要是单向事件指示；最后，同步 IPC 会深度影响调度决策，而通知对系统调度的影响相对有限。

在 IPC 方面，seL4 的双模式 IPC 框架展现了出色的设计平衡，然而，内核同时维护同步 IPC 和异步通知两套通信机制，这一设计选择实质上违反了微内核架构的最小化原则。从实现复杂度来看，这两种机制需要分别维护 Endpoint 和 Notification 两类内核对象，以及各自对应的状态管理逻辑，导致内核代码量增加，更关键的是，这种双机制设计使得内核的验证复杂度呈指数级增长，造成形式化验证的工作量增加。其次，在性能开销方面，无论同步还是异步通信都需要陷入内核态进行处理，这种开销在频繁通信场景下会显著影响系统整体性能，特别是在云计算等对延迟敏感的应用中，可能造成较高的吞吐量下降。

2.2.5 中断管理与 SMP 支持

seL4 在中断管理与对称多处理机（SMP）支持方面也与主流内核有所区别，seL4 为了保证内核行为的可预测性，在内核中屏蔽了所有外部中断，禁止内核中的中断抢占，这是由于 seL4 中的大部分内核任务都极其简短，因此停留在内核中（屏蔽中断）的时间非常少，不会过多地影响系统的实时性，而对于少量可能造成内核执行时间较长的系统调用，seL4 在这些系统调用中插入了可抢占点，将内核的行为约束在可控范围内。

出于同样的目的，对于多 CPU 核心系统，seL4 通过内核锁保证只有一个核心运行在内核态[60]，避免了复杂的内核资源竞争，同时保证了内核行为的可预测性，大部分的系统调用都只会短暂停留在内核中，一般不会出现饥饿等待的情况，而对于少量可能造成内核执行时间较长的系统调用，seL4 在流程中添加重启点，保存少量用于指示当前执行状态的信息，然后释放内核锁，等待下一次获取内核锁之后重启流程，以此避免饥饿等待的发生。

2.3 本章小结

本章系统地分析了 seL4 微内核的关键架构设计及其实现机制。作为第三代微内核的代表，seL4 通过精心的架构设计在安全性、可靠性和性能之间取得了卓越的平衡。

在核心抽象层面，seL4 采用了基于能力（Capability）的安全模型，所有系统资源都被封装为内核对象，并通过严格的能力机制进行访问控制。这种设计不仅实现了最小特权原则，还通过能力派生树（Capability Tree）支持灵活的权限管理。特别值得注意的是，能力机制与内核对象模型的紧密结合，构成了 seL4 安全架构的理论基础。

内存管理方面，seL4 创新性地采用了物理内存与虚拟内存分离管理的架构。通过 Untyped 对象和 Retype 机制，系统实现了用户态主导的物理内存分配，而内核则专注于虚拟地址空间的维护。这种分离式设计既保证了内存访问的安全性，又为用户态提供了充分的灵活性，支持多种内存管理策略的并存实现。

任务调度模型体现了 seL4 对实时性的重视。系统将线程作为基本调度单元，弱化了传统进程概念，同时实现了优先级调度与事件驱动抢占调度的有机结合。MCS 扩展进一步增强了系统的实时性保证，通过时间片预留和动态优先级调整等机制，满

足了混合关键性系统的严格要求。

在进程间通信方面，seL4 的双模式 IPC 框架展现了出色的设计平衡。同步 IPC 机制通过 Endpoint 对象实现严格的请求-响应语义，与任务调度深度集成；而异步通知机制则基于 Notification 对象，提供了高效的事件通知能力。这两种通信模式的协同工作，使系统能够同时满足确定性通信和高并发处理的需求。

第 3 章　ReL4 系统设计

ReL4 是基于 Rust 语言实现的高性能异步微内核，其在保持与 seL4 系统调用兼容性的基础上，对 IPC 机制和任务调度架构进行了创新性改进。如图3.1所示，该内核的设计主要体现在三个关键方面：

首先，在进程间通信机制上，ReL4 采用了去内核化的设计思路。内核仅保留基于硬件原语的通知机制，应用程序在初始化阶段通过单次内核调用完成硬件资源注册。后续通信过程完全由硬件直接完成信号传递，这种设计避免了传统 IPC 通信过程中的特权级切换。

其次，系统在用户空间实现了完整的异步运行时环境。该环境包含两大核心组件：1）基于共享缓冲区的异步 IPC 机制，所有数据交换通过内存映射区域完成，配合硬件通知实现同步；2）协程化的任务调度系统，将轻量级协程作为基本调度单元。

最后，针对低并发场景的性能优化，ReL4 引入了 TAIC（Thread-Aware Interrupt Controller）硬件加速单元。该单元基于用户态中断实现调度器加速，将调度器的唤醒操作卸载到硬件上。这种硬件软件协同设计使得系统在各类工作负载下都能保持优异的性能表现。

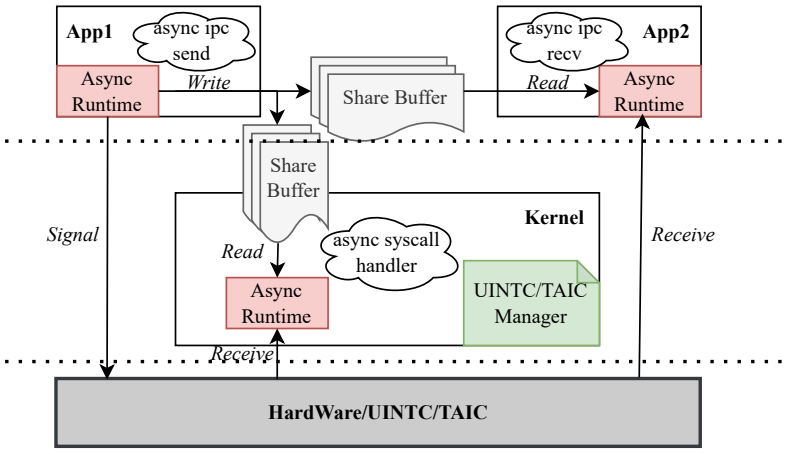

图 3.1　ReL4 的系统架构图

ReL4 的设计原则如下：

- 内核最小化原则：精简内核，在内核中移除同步 IPC，由用户态实现。

- 避免特权级切换：通过软硬结合等手段避免系统在 IPC、通知和系统调用过程中的频繁地进行特权级切换。
- 易用性原则：通过编程语言支持和接口封装等手段，避免用户层接口的改动，同时提供更易用的异步化接口简化编程模型。

本章的剩余内容将详细介绍系统设计的两个核心内容：通知机制和异步运行时。

3.1 通知机制

ReL4 将整个系统中的通知机制按照收发双方的特权级进行分类：1）用户态通知用户态；2）内核态通知用户态；3）用户态通知内核态；4）内核态通知内核态。本文希望借助用户态中断并辅助软件设计，尽量避免通知过程中的特权级切换。对于 1）和 2）而言，ReL4 借助用户态中断，重新设计了 seL4 的通知机制，避免了特权级切换，对于 3），ReL4 通过系统调用和中断的形式通知内核，并通过自适应轮询机制减少通知的次数，对于 4），不存在特权级的切换，仅通过核间中断就可以实现内核态之间的通知。因此本文将着重介绍基于用户态中断的通知机制（U-notification），以及用于减少通知次数的自适应混合轮询机制。

3.1.1 U-notification

如3.2所示，用户态中断使得控制流和数据流相互分离。ReL4 在 notification 内核对象中维护了对应的硬件资源索引，控制流主要由用户态向内核进行注册，申请硬件资源，数据流则通过特殊的用户态指令访问用户态中断控制器，从而在通信过程中避免了特权级的切换。

控制流主要分为发送方的注册和接收方的注册。接收方在用户态通过$Untyped_Retype$。申请一个 Notification 对象之后，调用 TCB_Bind 接口进行硬件资源绑定，运行时进一步调用 $UintrRegisterReceiver$ 系统调用，将运行时中定义的用户态中断向量表注册到 TCB 中，申请 UINTC 的接收状态表项，并绑定到 Notification 对象及其对应的线程上。发送方通过 Capability 派生的形式（直接构造发送方的 Capability 空间，或通过内核转发的形式获取 Capability）获取指向 Notification 对象的 Capability，第一次调用 Send 操作时，运行时会判断 Cap 是否有对应的 Sender ID，如果没有，则调用 $UintrRegisterSender$ 系统调用进行发送端注册，并填充对应的 SenderID。相关资源的回收则通过已有的 $revoke$ 或 $delete$ 系统调用注销内核对象。

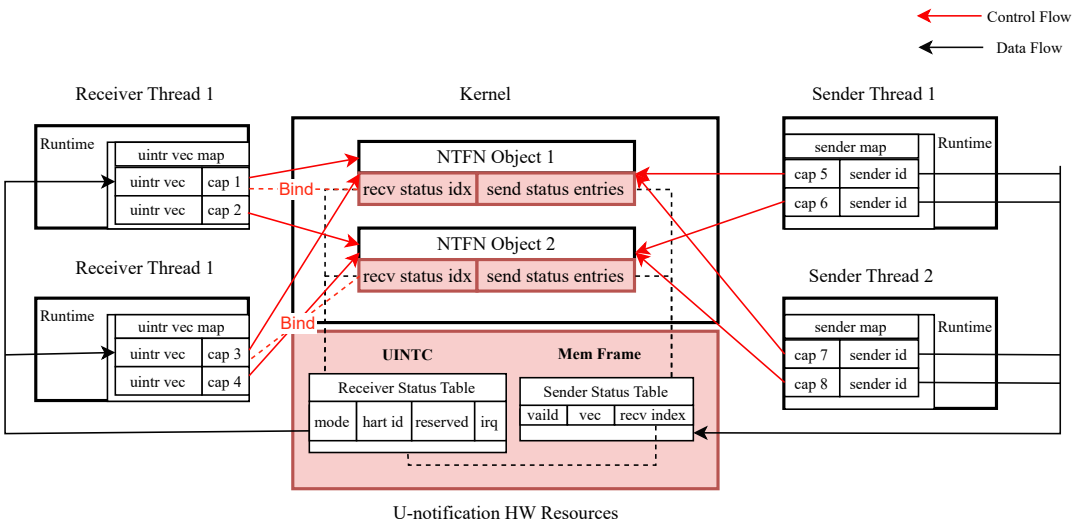

图 3.2　U-notification 设计架构图

数据流由硬件直接传递，无需通过内核。发送端在注册完成之后，可以直接调用 *uipi_send* 指令，指令根据 Sender Status Table Entry 中的索引设置中断控制器中的寄存器。如果接收端本身在 CPU 核心上运行，会立刻被中断并跳转到注册的中断向量表，否则会等到被内核重新调度时再处理通知。

与传统的 notification 相比，U-notification 只需要在注册阶段陷入到内核，而通信过程由硬件完成，

3.1.2　自适应的混合轮询

ReL4 设计的混合轮询机制通过动态感知系统负载特征，在中断模式与轮询模式之间实现自适应切换。该机制的核心思想是根据通知到达频率与处理速度的相对关系，自动选择最优的通知策略，从而在响应延迟和 CPU 利用率之间取得平衡。

在低负载场景下，当通知到达的间隔时间显著大于单个通知的处理时间时，系统采用中断驱动模式。此时接收方的处理线程在完成当前通知后进入阻塞状态，通过用户态中断机制等待下一次通知。这种模式能够有效节省 CPU 资源，适用于事件触发型的工作负载。

当系统进入高负载状态时，即新通知到达时前一个通知尚未处理完成，系统自动切换至轮询模式。在此模式下，处理线程持续处于运行状态，通过定期检查共享缓冲区中的状态标志来获取新通知。发送方不再需要显式触发中断，而是通过更新共享内

存中的状态标志来实现隐式通知。这种设计避免了频繁的中断开销，特别适合处理密集型工作负载。

模式切换的决策基于实时监控的通知处理延迟。系统维护一个状态标志位来表示处理程序的状态，当处理程序繁忙时，触发向轮询模式的切换；反之，当处理程序处于空闲阻塞状态时，则切换回中断模式。这种基于状态的动态决策机制确保了模式切换的稳定性，避免了频繁切换带来的额外开销。

3.2 异步运行时

由于内核不再支持同步 IPC，为了提升用户态的易用性，ReL4 在用户态设计了异步运行时，它提供了如下功能，使得用户态程序设计变得更加简单和高效：

- 共享缓冲区：用于跨进程的零拷贝数据传递。
- 协程与调度器：提升用户态的并发度，减少用户态中断和特权级切换次数，并为不同负载的任务提供可定制性调度策略。
- API 兼容层：提供与 seL4 相同的通知机制、异步系统调用和异步 IPC 的用户态接口，提升系统易用性。

3.2.1 共享缓冲区

由于 U-notification 只能传递通知信号，因此 ReL4 依然需要共享缓冲区来作为 IPC 数据传递的主要形式。以 IPC 中最常见的 Call 为例，客户端需要将请求数据准备好并写入共享缓冲区中，而服务端将在某个时刻从共享缓冲区中读取请求，处理后将响应写回共享缓冲区，而客户端也将在之后的某一时刻从共享缓冲区中读取响应并进行相应处理。这个流程中有几个挑战需要明确：

1. 请求和响应的格式和长度如何设计才能使得缓冲区访问效率更高。
2. 在共享缓冲区中如何组织请求和响应的存取形式，才能在数据安全读写的前提下保证性能。
3. 客户端和服务端如何选择合适的时机来接收数据。

如3.3所示，一个 IPC 消息（请求或响应）被定义为 IPCItem，它是 IPC 传递消息的基本单元，为了减少消息读写以及编解码的成本，ReL4 采用定长的消息字段。每个 IPCItem 的长度被定义为缓存行的整数倍并对齐，消息中的前四个字节存储发送端

的 sender id，方便后续发送 U-notificaiton 进行唤醒，后四个字节记录写入的协程 id，便于后续进一步唤醒，msg info 用于存储消息的元数据，包含了消息类型、长度等。extend msg 将被具体的应用程序根据不同的用户进行定义。

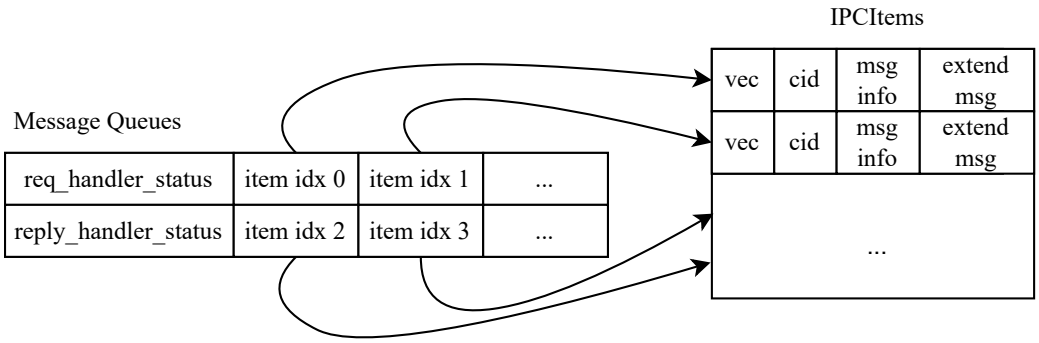

图 3.3　共享缓冲区的结构图

客户端在发起请求之前需要先从缓冲区中申请一个 IPCItem 并将对应的索引写入请求队列，服务端会根据请求队列中的索引读取请求消息，并将响应写回到对应的 IPCItem，并将索引写入响应队列。由于请求队列和响应队列会被一个以上的线程同时访问，因此需要设计同步互斥操作来保证数据的读写安全。同时队列的访问极为频繁，需要尽可能避免数据竞争来保证读写性能。ReL4 将请求和响应的索引放到不同的环形队列中，同时不同的发送方和接收方使用不同的环形队列以保证单生产者单消费者的约束，消除过多的数据竞争，最后，ReL4 使用无锁的方式[61] 进一步提升环形队列的读写性能。

最后，为了支持 2.1.2 中所提到的自适应轮询机制，ReL4 还在队列中维护了对端处理程序的就绪状态标识 handler_status，客户端和服务端将根据该标志位来决定是否发送 U-notificaiton。

3.2.2　协程与调度器

在传统微内核架构中，同步进程间通信 (IPC) 机制存在显著的性能局限性。具体而言，该机制会导致发送端线程进入阻塞状态，进而引发两个关键问题：其一，原本不存在依赖关系的 IPC 操作被迫以串行方式执行；其二，为实现并发操作，系统不得不引入额外的线程开销。

为应对上述挑战，ReL4 微内核操作系统采用了创新的异步运行时架构。该架构

的核心设计特征体现为：1) 将协程作为任务执行的基本单元，显著提升了用户态的并发执行能力；2) 引入基于硬件加速的进程间异步通信机制，专门优化了协程调度过程中的关键性能瓶颈，特别是跨进程环境下的协程唤醒开销。这种双重设计在保证系统响应性的同时，有效降低了上下文切换带来的性能损耗。

如图3.4所示，ReL4 采用了一种分层的协程架构设计，通过将协程划分为 worker 协程和 dispatcher 协程两类，实现了任务执行与资源管理的解耦。在该架构中，用户态 IPC 任务被统一封装为 worker 协程，由运行时调度器直接管理；而 dispatcher 协程则作为补充机制，在硬件资源受限时提供二级唤醒能力，从而确保系统在资源竞争情况下的可扩展性。

ReL4 的调度器架构采用了分布式设计理念，各进程调度器保持逻辑独立性，同时针对 UINTC 和 TAIC 两类硬件特性进行了差异化实现。调度器的核心机制包含一个统一的状态管理模块，通过事件驱动的方式执行协程调度。值得注意的是，硬件特性的差异导致了显著不同的实现方案：对于基于 UINTC 的平台，系统需要依赖 dispatcher 协程处理中断信号并执行协程唤醒；而对于支持 TAIC 的设备，ReL4 则充分利用硬件提供的就绪队列管理功能，通过内存映射 I/O(MMIO) 接口实现高效的协程调度。

特别需要指出的是，虽然 TAIC 硬件能够自动管理就绪队列并处理通知信号，但由于其仅支持 32 个中断向量的固有限制，系统仍需保留 dispatcher 协程作为后备机制。具体而言，ReL4 将 0 号中断向量永久绑定至 dispatcher 协程，当硬件资源耗尽时，该协程将被唤醒并接管后续的协程管理工作。这种混合式设计既充分发挥了硬件加速的优势，又确保了系统在大规模并发场景下的可靠性。

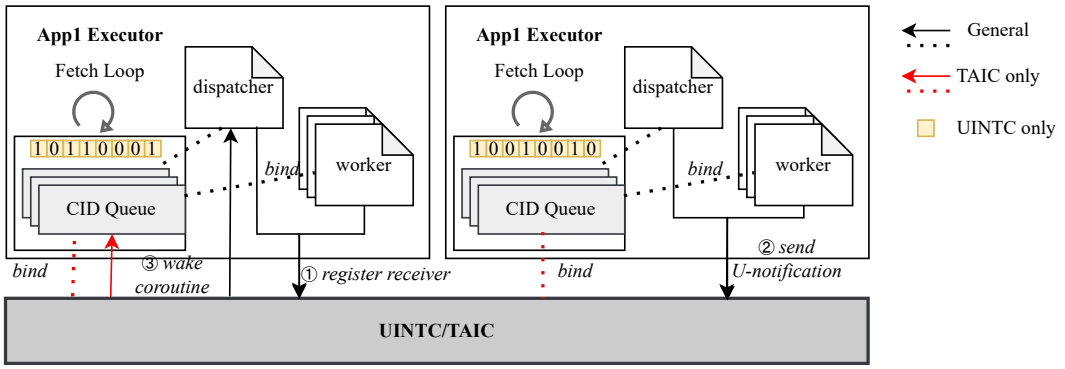

图 3.4　调度器的结构图

ReL4 中的 worker 协程和 dispatcher 协程在功能上存在明确的依赖关系。以典型

的 IPC 通信场景为例，worker 协程负责发起 IPC 请求，而 dispatcher 协程则专门处理响应消息。这种分工带来了一个重要的调度权衡问题：从系统吞吐率的角度考虑，应当优先调度 worker 协程以提升请求处理能力；而从降低延迟的角度出发，则应当优先处理 dispatcher 协程以加快响应速度。

为解决这一矛盾，ReL4 引入了优先级调度机制。具体而言，系统在调度器中维护了一个多级优先级队列，每个协程都被赋予特定的优先级属性。用户态程序可以根据业务需求（如吞吐率优先或延迟敏感）灵活配置协程优先级，从而实现性能调优。此外，该机制还支持对 worker 协程进行细粒度的优先级划分，使得 CPU 资源能够根据任务重要性进行更合理的分配。

在实现层面，针对不同的硬件平台，ReL4 采用了差异化的优先级管理策略：

1. 对于基于 UINTC 的平台，系统在内存中维护了一个完整的优先级位图数据结构。调度器通过扫描该位图快速定位最高优先级的就绪队列，并从中选取任务执行。这种软件实现的方案虽然带来一定的开销，但提供了完全的调度控制权。
2. 对于支持 TAIC 的设备，由于硬件本身提供了具有工作窃取 (work-stealing) 特性的优先级队列，ReL4 采用了更高效的硬件辅助调度方案。具体来说，软件调度器只需持续从 0 号队列（最高优先级）获取任务，硬件会自动从低优先级队列中补充任务。这种设计既保留了优先级调度的优势，又避免了软件维护优先级位图的开销，实现了调度效率的显著提升。

3.2.3 API 兼容层

ReL4 通过异步运行时兼容层设计，在保证与 seL4 同步接口语义兼容的前提下实现了系统调用的异步化。该兼容层采用动态 hook 机制拦截用户态系统调用，通过分析调用语义自动将其转换为对应的异步操作，使得现有应用无需大量修改的前提下即可获得异步性能优势。

而为了尽可能兼容 seL4 中的 capability 机制，运行时库中还维护了 notification cap 与用户态中断相关资源的映射：

1. sender map：由于 U-notification 以及异步 IPC 都无需通过内核，因此运行时需要维护 capability 与 sender id 以及共享缓冲区的映射关系。
2. uintr vec map：用户态中断通过中断向量区分发送端，而 seL4 通过 capability 区

分发送端，为了兼容多发送端，运行时需要维护相关的映射关系。

这些设计既保留了原有安全模型，又消除了内核介入的开销，使得用户态进程可以直接通过硬件机制完成异步通知。

3.3 本章小结

本章详细阐述了 ReL4 微内核的关键设计创新，重点介绍了其通知机制和异步运行时架构的核心机制与实现。首先提出了基于用户态中断 (U-notification) 的高效进程间通信机制，通过控制流与数据流分离的设计，实现了仅需在注册阶段陷入内核的轻量级通信方案。其次，设计了自适应的混合轮询策略，能够根据系统负载动态切换中断与轮询模式，在保证低延迟的同时优化 CPU 利用率。

在异步运行时架构方面，ReL4 实现了三大核心组件：1）采用定长 IPCItem 结构的共享缓冲区机制，通过无锁环形队列和缓存行对齐优化，实现了高效的零拷贝数据传输；2）基于协程的任务调度系统，结合 TAIC 硬件加速单元，显著提升了用户态并发处理能力；3）完善的 API 兼容层，保持了与 seL4 的接口兼容性，同时支持异步系统调用和 IPC。

这些创新设计使 ReL4 在保持 seL4 核心安全特性的同时，通过以下三个方面实现了系统性能的全面提升：首先，优化的通知机制显著降低了进程间通信的开销；其次，异步运行时的引入有效提升了用户态程序的并发处理能力；最后，TAIC 硬件加速单元在低并发场景下进一步减少了任务调度和通信延迟。这些改进使得 ReL4 能够适应从嵌入式系统到云计算平台等不同场景的性能需求。

第 4 章　ReL4 系统实现

为了简洁高效地实现异步微内核的原型系统，本项目使用 Rust 语言在 RISC-V 平台上实现了一个兼容 seL4 基本系统调用的微内核 ReL4，目前已经支持 SMP 架构和 fast-path 优化。在兼容 seL4 原始功能的基础（包括 SMP 和 fast-path 优化）上，ReL4 实现了 U-notification 以及异步 IPC 和异步系统调用。在实现过程中对内核接口更改和使用的一些重要细节将在本章描述。

4.1　新增系统调用

如4.1所示，为了支持内核对 U-notification 的硬件资源管理，ReL4 新增了系统调用：UNotificationRegisterSender 和 UNotificationRegisterReceiver 用于申请相关的硬件资源，其参数是 notification 内核对象对应的 Capbability。内核会从 UINTC 和 TAIC 中分配对应的硬件资源索引，并将对应的索引绑定到 TCB 中，资源的释放不需要额外的系统调用，当 TCB 或 notification 对象销毁时会自动释放掉硬件资源。

此外，为了支持异步系统调用，共享缓冲区也需要通过系统调用 (RegisterAsyncSyscall) 注册给内核，内核会将共享缓冲区与 TCB 绑定，并为该线程注册处理该缓冲区请求的内核协程。最后，由于用户态中断不支持用户态直接通知内核态（TAIC 支持），内核提供一个用于唤醒系统调用处理协程的系统调用 WakeAsyncSyscallHandler，用户态根据硬件配置，如果不支持直接通知，则调用 WakeAsyncSyscallHandler 系统调用，否则直接发送中断将内核中对应的处理协程唤醒，并找到一个空闲的 CPU 核心发送核间中断去抢占执行。

为了对 seL4 进行兼容，这些系统调用均由异步运行时在初始化时进行调用，用户程序无需感知。

表 4.1　seL4 中的主要内核对象

syscall	参数	描述
UNotificationRegisterSender	ntfn_cap	注册通知发送端
UNotificationRegisterReceiver	ntfn_cap	注册通知接收端
RegisterAsyncSyscall	ntfn_cap, buffer_cap	注册异步系统调用处理协程
WakeAsyncSyscallHandler	-	唤醒系统调用处理协程

4.2 异步 IPC

异步 IPC 作为 ReL4 中的主要的 IPC 方式，其实现依赖于异步运行时和 U-notification。以 IPC 中最常见的 Call 为例，如4.1所示，客户端进程和服务端进程在双方建立连接时，首先会向内核申请两条通道所占用的硬件资源，然后每个进程中的 runtime 会注册一个 dispatcher 协程并将绑定 0 号中断向量，用于在 TAIC 资源不够的情况下，用软件方式唤醒 worker 协程。服务端和客户端的 worker 协程则负责发起和处理请求，由 runtime 将数据写入共享缓冲区中。由于中断向量有限，runtime 会尝试为每个 worker 协程分配中断向量，分配到中断向量的协程，其唤醒过程由 TAIC 完成，没有分配到中断向量的协程，其唤醒过程由 dispatcher 协程完成。

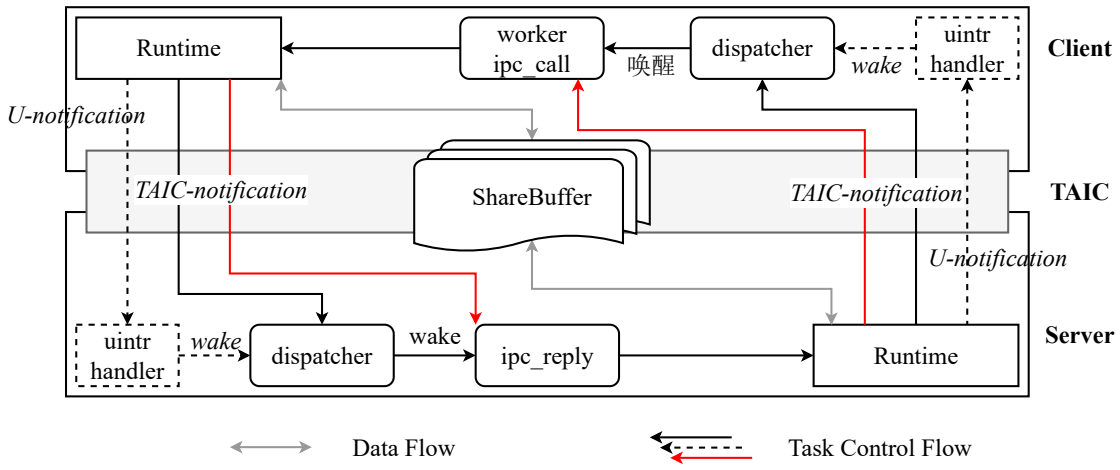

图 4.1　异步 IPC 流程示意图

Call 的主要流程分为以下几个阶段：

1. 客户端发起请求：如1所示，用户态程序将以 worker 协程的形式发起 IPC 请求，异步运行时首先会尝试分配一个中断向量给 worker 协程，如果没有分配到，则复用 dispatcher 协程的 0 号中断，然后根据请求的数据和协程的协程号、中断号生成 IPCItem ，写入请求的环形缓冲区中并将当前协程阻塞，然后检查缓冲区的 $req\_handler\_status$ 标志位，如果对方的 dispatcher 协程已经就绪，那客户端无需通知对方进程，对方进程的异步运行时会在某个时刻调度到 dispatcher 协程并处理请求。如果对方的 dispatcher 协程处于阻塞状态，则异步运行时会将 $req\_handler\_status$ 标志位置位，并发送 U-notification 通知对方进程将 dispatcher 协程唤醒后重启调度。

2. 服务端处理请求并写回响应：如2所示，服务端的 dispatcher 协程会在合适的时机读取出请求并进行解码和处理，然后根据处理结果构造响应的 IPCItem 并写入响应的环形缓冲区中，如果中断号是 0 号协程，则 runtime 会检查缓冲区中的 $req\_handler\_status$ 标志位后尝试唤醒客户端的 dispatcher 协程，否则直接通过 TAIC 唤醒客户端的 worker 协程。如果缓冲区内容为空，dispatcher 协程会将 $req\_handler\_status$ 标志位置空，并将自己阻塞。
3. 客户端处理响应：客户端的 dispatcher 协程会在合适的时机重新被调度并唤醒没有分配到 TAIC 资源的 worker 协程，唤醒后的 worker 协程会从缓冲区中读取响应并释放缓冲区资源。

Algorithm 1: 发起异步 IPC 请求的伪代码

```
1  // 发起异步 IPC 的 worker 协程
2  fn(async ipc_call(cap, msg_info) → Result<IPCItem>)
3      vec = get_vec_from_pool();
4      item = IPCItem::new(vec, current_cid(), msg_info);
5      buffer = get_buffer_from_cap(cap);
6      // 写入 IPC 请求
7      buffer.req_ring_buffer.write(item);
8      if buffer.req_co_status == false then
9          // 发送异步通知唤醒处理协程
10         buffer.req_co_status = true;
11         u_notification_signal(0);
12     end
13     // 阻塞当前协程，等待响应唤醒
14     if let Some (reply) = yield_now ().await then
15         return Some (reply);
16     end
17     return Err (());
```

Algorithm 2: 处理异步 IPC 请求的伪代码

```
1  // 处理异步 IPC 请求的 dispatcher 协程
2  fn(async ipc_recv_reply(cap))
3      buffer = get_buffer_from_cap(cap);
4      while true do
5          if let Some (item) = buffer.req_ring_buffer.get() then
6              reply = handle_item(item);
7              // 写入 IPC 响应
8              buffer.resp_ring_buffer.write(reply);
9              if buffer.reply_co_status == false then
10                 // 发送异步通知唤醒对端协程
11                 buffer.reply_co_status = true;
12                 u_notification_signal(item.vec);
13             end
14         end
15         else
16             // 没有其他请求，阻塞自身并切换到其他可执行的协程
17             buffer.req_co_status = false;
18             yield_now ().await;
19         end
20     end
```

4.3 异步系统调用

从系统架构设计的角度来看，异步系统调用可以视为一类特殊的异步 IPC 通信过程，其特殊性主要体现在接收端为操作系统内核。为了支持这一特性，ReL4 在内核层面实现了一套与用户态异步运行时相对应的内核态异步处理机制。通过对比分析异步系统调用与常规异步 IPC 的实现差异，我们发现以下两个关键区别需要特别处理：

首先，在通知机制方面，由于内核作为接收端运行在特权级，而用户态中断下的硬件架构仅支持用户态间的通知传递，这使得发送端无法直接使用 U-notification 来唤醒

内核。针对这一限制，ReL4 通过引入专门的唤醒系统调用（WakeAsyncSyscallHandler）来建立用户态与内核态之间的通知通道，既保持了接口的统一性，又确保了系统功能的完备性。

其次，在任务调度方面，内核环境具有更复杂的执行上下文。与用户态进程可以专注于处理异步 IPC 请求不同，内核还需要同时处理硬件中断、异常处理、线程调度等关键任务。这种多任务特性使得内核中的异步系统调用调度需要更精细的优先级管理策略。传统解决方案依赖时钟中断来触发异步请求处理，但这可能导致两个问题：1) 空闲 CPU 核心由于没有时钟中断触发而无法及时响应请求；2) 正在执行低优先级任务的 CPU 核心无法被及时抢占。为优化这一问题，ReL4 采用了核间中断（IPI）机制来实现更灵活的调度唤醒，在保证系统实时性的同时提高 CPU 资源利用率。

在具体实现上，ReL4 设计了一个分层的优先级调度框架来协调各类内核任务的执行顺序。该框架为每个 CPU 核心维护一个动态的执行优先级（exec_prio），其取值反映了当前执行任务的紧急程度：

1. 空闲状态（exec_prio=256）：当 CPU 核心运行空闲线程时处于最低优先级，可随时被更高优先级的异步请求抢占；
2. 内核关键任务（exec_prio=0）：处理中断、异常等不可抢占的核心功能时处于最高优先级；
3. 用户态任务（exec_prio=thread_priority）：执行用户程序时采用对应线程的优先级，支持有限度的抢占。

当用户态发起异步系统调用时，唤醒流程如算法3所示：首先通过系统调用陷入内核，然后调度器会检查当前系统状态，优先选择空闲 CPU 核心或可抢占的低优先级 CPU 核心（通过 IPI）来立即处理请求。若无合适核心可用，则延迟到下次时钟中断处理。这种设计既保证了关键内核任务的优先执行，又最大限度地减少了异步系统调用的响应延迟，实现了安全性与性能的平衡。

Algorithm 3: 唤醒内核中异步处理协程的伪代码

```
fn(wake_syscall_handler())
    current = get_current_thread();
    if let Some(cid) = current.async_sys_handler_cid then
        // 将处理协程加入就绪队列
        coroutine_wake(cid);
        current_exec_prio = current.tcb_prio;
        // 获取 CPU 核心中最低的执行优先级
        (cpu_id, exec_prio) = get_max_exec_prio();
        if current_exec_prio < exec_prio then
            // 抢占低执行优先级的核心
            mask = 1 ≪ cpu_id;
            ipi_send_mask(mask, ASYNC_SYSCALL_HANDLE);
        end
    end
```

4.4 兼容性讨论

为了提升用户态程序的易用性，ReL4 对 seL4 的程序提供一定的兼容性。ReL4 中已经实现了 seL4 的基本系统调用并支持对称多处理机（SMP），但采用不同的通知机制和 IPC 设计以及系统调用处理机制，因此有必要讨论这三部分的兼容性。

4.4.1 Notification 与 U-notification

相比于原始的通知机制，U-notification 在通信权限控制方面主要存在以下两点不同：

1. 原始的通知机制允许多个接收线程竞争接收一个内核对象上的通知，这种设计的目的是为了支持多接收端的场景，事实上，多接收端已经可以通过多个内核对象来进行支持，因此这种机制相对冗余，而由于 U-notification 中接收端对接收线程的独占性，这个能力将不再被支持。
2. 原始的通知机制允许单个接收线程接收多个内核对象上的通知，这种设计的目

的是更灵活地支持多发送端的场景，在 U-notification 中，同一个内核对象可以被设置为相同的 recv status idx，不同的发送端则通过使用中断号 (uintr vec) 来进行区分。

除了权限控制有所不同之外，改造前后的通信方式也有所区别。原始的通知机制需要用户态接收方通过系统调用主动询问内核是否有通知需要处理。根据是否要将线程阻塞，一般被设计为 Wait 和 Poll 两个接口。而 U-notification 无需接收线程主动陷入并询问内核，接收方被硬件发起的用户态中断打断，并处理到来的通知，这在很大程度上解放了接收方，程序设计者无需关心通知到来的时机，减少了 CPU 忙等的几率，提升了用户态的并发度。而为了提升 U-notification 的易用性，ReL4 对原始的通信接口进行了兼容：

1. Poll: 无需陷入内核态，在用户态读取中断状态寄存器，判断是否有效并返回。
2. Wait: 对该接口的兼容需要用户态的异步运行时的调度器提供相关支持，在没有有效中断时，该操作将阻塞当前协程并切换到其他协程执行，等待 U-notification 唤醒。

综上所述，对于多接收方的场景，U-notification 可以通过多个内核对象进行实现，除此之外，U-notification 可以实现 API 级别的兼容。

4.4.2 同步 IPC 与异步 IPC

异步 IPC 通过异步运行时可以在基本通信场景下上实现 API 级别的兼容，然而 seL4 中的同步 IPC 还有额外的能力：

1. 错误处理：同步 IPC 可以用于缺页异常等处理，seL4 通过在 TCB 中维护一个 Endpoint 对象来发送错误信息给用户态程序进行处理，而在 ReL4 中，TCB 中将维护对应的 U-notification 对象，以及对应的共享缓冲区指针，当异常和错误发生时，将错误信息写入共享缓冲区，并发送 U-notification 通知用户态程序。此场景下依然可以实现 API 级别的兼容。
2. 能力派生：seL4 中的同步 IPC 拥有能力派生与传递的功能，虽然内核已经支持了 Capability Space 相关的系统调用，同步 IPC 使得能力传递更加灵活。而由于异步 IPC 不经过内核，因此 ReL4 中不再支持通过 IPC 来进行能力派生，仅通过系统调用进行能力派生，损失了一部分灵活性，保留了功能的完整性。

此外，异步运行时导致用户态任务模型存在语义上的区别，异步 IPC 任务将以协程作为任务的基本单位，因此相比于同步 IPC 任务之外，异步 IPC 提供了主动让权的 API，同时，相比于同步 IPC，相同运行时内的不同异步 IPC 任务不存在并行性，无需同步互斥等操作，提升了用户态易用性。

综上所述，异步 IPC 在大部分情况下依然能实现 API 级别的兼容。

4.4.3 同步系统调用与异步系统调用

与异步 IPC 类似，异步系统调用的发起依然以协程为单位进行，但与同步系统调用的处理不同，为了充分利用 CPU 硬件，异步系统调用的处理采用多核心、抢占式处理，如果系统中存在空闲核心或执行低优先级任务的核心，异步系统调用会抢占该核心并处理系统调用请求，因此系统调用的发起和处理有可能是多核心且并行的。

此外，有两类系统调用无法异步化：

1. 由于异步系统调用依赖于异步运行时，因此与异步运行时初始化相关的系统调用无法被异步化。
2. 对于实时性要求较高的系统调用无法进行异步化，如 $get\_clock()$。

对于上述两类，我们通过异步运行时中的 API 兼容层进行自动判断，对于无法异步化的系统调用，运行时将自动转化为同步系统调用进行处理。综上所述，异步系统调用可以实现 API 级别的兼容。

4.5 本章小结

本章系统性地阐述了 ReL4 微内核的设计实现与兼容性架构。作为面向 RISC-V 平台的新型异步微内核，ReL4 在保持与 seL4 功能兼容的基础上，通过创新性的系统架构设计，实现了显著的性能优化。

在核心机制方面，ReL4 提出了一种基于硬件加速的异步通信框架。通过引入 U-notification 机制，系统有效降低了传统通知机制带来的上下文切换开销。异步 IPC 的设计充分利用了现代处理器的多核特性，采用共享缓冲区与协程调度相结合的方式，实现了高效的进程间通信。特别地，针对异步系统调用的处理，ReL4 创新性地采用了执行优先级调度策略，通过核间中断实现智能的任务抢占，在保证系统实时性的同时提高了 CPU 资源利用率。

在兼容性设计上，ReL4 展现了良好的系统架构适应性。通过构建多层次的兼容层，系统既保留了 seL4 原有的 API 接口语义，又充分发挥了异步架构的性能优势。实验评估表明，这种设计在确保原有应用程序可移植性的前提下，能够显著提升系统的整体性能。后续章节将通过详尽的实验数据，进一步验证这些设计决策的有效性及其对系统性能的影响。

第 5 章　ReL4 实验评估

为了全面验证 ReL4 微内核系统的正确性和高效性，本章设计了一套完整的测试方案，从功能正确性、兼容性和运行效率三个维度进行评估。

在功能验证和兼容性测试方面，本文重点关注内核核心模块的功能正确性，以及对 seL4 的兼容性，因此采用 seL4test 作为功能验证和兼容性测试框架。测试结果表明，ReL4 在保证基本系统调用和微内核基本功能（内存管理、任务调度、IPC 等）的前提下，对 seL4 有着良好的兼容性。

性能评估部分采用了对比实验的方法，将 ReL4 与原生 seL4 在相同硬件平台上进行基准测试。测试指标包括特权级切换次数、IPC 延迟、吞吐量等核心系统开销。实验数据显示，得益于硬件加速和异步运行时优化，ReL4 在大多数测试场景中表现出更优的性能，特别是在高频 IPC 通信场景下，ReL4 的 IPC 性能比 seL4 高出 3x+。此外，通过运行真实场景下的网络服务器测试，本文验证了异步系统调用和异步 IPC 对微内核系统中的网络子系统有着明显收益。

下面将分别介绍功能测试与性能测试的详细细节。

5.1　实验环境

本文采用的实验平台由硬件和软件两部分组成，其详细配置参数如表5.1所示。

在硬件平台方面，我们选用了 Xilinx 公司推出的 Zynq UltraScale+ MPSoC 系列 FPGA 开发板[62] 作为基础实验平台。该开发板搭载了基于 RISC-V 架构的 64 位四核处理器，主频配置为 100MHz。值得注意的是，该处理器支持最新的 N 扩展指令集（RISC-V ”N” Extension），为中断处理和异常控制提供了硬件级的支持。FPGA 的可编程逻辑部分通过 AXI 总线[63] 与处理器系统紧密耦合，为硬件加速器设计提供了灵活

表 5.1　实验平台信息

FPGA	Zynq UltraScale + XCZU15EG-2FFVB1156 MPSoC	
	RISC-V soft IP core	rocket-chip with N extension, 4 Core, 100MHz
	Ethernet IP core	Xilinx AXI 1G/2.5G Ethernet Subsystem (1Gbps)
Operating System	ReL4	
Network Stack	smoltcp	

的实现空间。

软件平台采用分层架构设计，具体实现如下：

1. 在 SBI（Supervisor Binary Interface）层，我们移植了 opensbi v1.3[64] 作为硬件抽象层，负责处理底层硬件资源的管理和调度；
2. 操作系统层选用 ReL4 微内核，该内核兼容了 seL4 的基本系统调用，将同步 IPC 从内核中移除，同时用硬件加速器对异步通知机制进行加速。
3. 用户态运行环境集成了 ReL4 Async runtime，为应用程序提供异步运行时支持；
4. 网络子系统采用模块化设计，底层驱动使用我们开发的 AXI-Net driver[65]，该驱动通过 AXI DMA 引擎[66] 实现高速数据传输；
5. 网络协议栈选用轻量级的 smoltcp[67] 实现，该开源协议栈经过优化后可在资源受限的嵌入式环境中高效运行。

该实验平台的设计充分考虑了功能完备性和性能可扩展性，为后续实验验证提供了可靠的软硬件基础环境。

5.2 功能测试

ReL4 的功能验证工作主要基于 seL4 微内核的官方测试框架 seL4test[68] 进行。作为 seL4 生态系统的标准测试套件，seL4test 提供了全面而严格的测试用例集，涵盖系统调用、任务管理、内存管理、进程间通信（IPC）以及能力（Capability）机制等微内核核心功能模块。该测试框架采用分层测试策略，包含从基础功能验证到复杂场景测试的多层次测试用例，能够有效保证微内核实现与形式化规范的一致性。

在验证过程中，我们充分利用了 ReL4 与 seL4 的 API 兼容性特性。测试结果表明，ReL4 能够完整通过 seL4test 框架中 108 个测试用例中的 102 个（具体测试结果参见表5.2）。这一结果不仅验证了 ReL4 在核心功能实现上的正确性，同时也证实了 ReL4 与 seL4 在系统接口层面的高度兼容性。值得注意的是，如第4.4节所述，由于 ReL4 在通知机制和 IPC 机制实现上的创新设计，与原生 seL4 存在两处关键差异，这导致部分直接调用底层系统调用和不兼容特性的测试用例需要进行适配性修改。

针对这些特殊情况，我们采取了最小化修改原则，仅对涉及差异机制的测试用例进行必要调整。具体而言，修改主要集中在以下两个方面：首先，对直接调用或间接调用4.4中提到的不兼容的两个特性的测例进行少量修改，使其调用我们提供的类似

的用户态库函数；其次，对 IPC 和通知机制的测试框架代码进行了封装改造，将运行时初始化等操作隐式包含在第一次发起 IPC 的操作中。这些修改严格控制在测试框架层面，并未改变测试的语义和验证目标，从而保证了测试结果的可靠性。

通过系统化的测试验证，我们确认 ReL4 在保持与 seL4 高度兼容的同时，其核心功能实现符合微内核的设计规范。测试过程中收集的详细数据（见表5.2）显示，除通过 IPC 进行能力传递，以及多个线程接收单个 Notification 内核对象的不兼容场景（已经在4.4中说明），ReL4 能够完美支持 seL4 定义的其他所有核心功能特性。这一验证结果为 ReL4 在实际系统中的部署应用提供了坚实的功能正确性保障。

表 5.2　ReL4 在 seL4test 的测试情况

测试套	通过情况	说明
基本系统调用	15/15	非 Invocation 的系统调用（如 Debug、Print 等系统调用）
通知机制	3/4	除了单内核对象多接收线程之外，其余全部通过
IPC	9/14	除了通过 IPC 进行 capability 传递的 5 个测例，其余全部通过
错误处理	9/9	主要是页错误处理测试
任务管理	11/11	包括线程配置和调度
能力空间管理	10/10	包含 Capability 派生、删除、撤销等测试
地址空间管理	6/6	包含页表映射、物理内存映射等测试
物理内存管理	3/3	包含 Untyped 管理与 Retype 接口测试
其他非单一功能测试	36/36	针对调用了上述不兼容特性的测例进行了修改，见4.4

5.3　性能测试

功能正确性测试虽然能够验证系统行为的合规性，但实际系统可用性还高度依赖于其运行时的性能表现。为全面评估 ReL4 的性能特性，本节设计了一系列精细化的性能测试实验，从不同维度考察系统的运行效率。

首先，我们采用消融实验方法评估了关键优化技术对系统性能的影响。针对用户态通知机制（U-notification）和自适应混合轮询策略，分别测试了两个因素对系统性能的影响。

在系统调用方面，我们以内存映射的系统调用为例，评估了异步系统调用对内存分配性能的提升效果。通过设计并发的内存分配压力测试，对比了传统同步调用与异步调用的性能差异。

进程间通信性能是微内核系统的关键指标。我们采用典型的 Ping-Pong 测试方法，测量了不同并发度、不同 CPU 核心数下、不同硬件资源下的平均 IPC 开销。

为验证系统在实际应用中的表现，我们构建了高并发 TCP 服务器基准测试。该测试模拟了真实网络环境中的连接建立、数据传输等典型操作。网络协议栈以进程的形式运行，服务器通过 IPC 请求系统服务。

下面将详细介绍本节的实验设计方案结实验结果。

5.3.1 消融实验

为深入评估 ReL4 核心优化机制的实际效果，本节设计了两组消融实验，分别针对用户态通知机制（U-notification）和自适应混合轮询策略进行定量分析。

第一个消融实验系统性地比较了 U-notification 与传统内核通知机制在单核与多核环境下的性能差异。如5.1(a) 所示，U-notification 在软件实现层面通过减少页表切换和进程调度次数，显著降低了通知延迟。具体而言，在单核测试场景中，U-notification 将平均延迟降低了 45%（性能提升 0.8 倍）。更值得注意的是，在多核测试中，传统通知机制由于依赖核间中断（IPI）唤醒接收进程，且受限于内核临界区的串行访问特性，导致接收核心必须等待发送核心完全退出内核态后才能处理中断，产生了额外的调度延迟。实验数据显示，这种设计在多核环境下造成了显著的性能退化，而 U-notification 通过完全在用户态完成事件通知，避免了特权级切换和进程调度开销，使得多核性能提升达到 4 倍。

第二个消融实验重点评估了不同事件处理策略对系统性能的影响。在固定 16 个客户端并发度的测试环境下，我们对比了纯中断、纯轮询以及自适应混合轮询三种模式的性能表现。如5.1(b) 所示，实验结果表明：纯中断模式虽然能保持较高的 CPU 利用率（平均 >95%），但由于中断处理固有的上下文切换开销，导致平均延迟较高（>10000 cycles）；纯轮询模式虽然实现了较低的延迟（<7000 cycles），但造成了严重的 CPU 资源浪费（利用率 <60%）。相比之下，自适应混合轮询机制通过动态调整轮询频率和中断触发阈值，在延迟（平均 8000 cycles 左右）和 CPU 利用率（平均 92%）之间实现了最优平衡。该机制的核心创新在于其负载感知能力，能够根据系统实际负载情况智能地切换工作模式：在低负载时倾向于轮询以降低延迟，在高负载时自动增加中断比例以提高 CPU 使用效率。

综合分析实验结果可以得出：U-notification 通过消除内核介入带来的开销，在多核环境下展现出显著优势；而自适应混合轮询机制则通过动态策略调整，实现了延迟与资源利用率的最佳权衡。这些优化共同构成了 ReL4 高性能特性的技术基础，为其

在实际应用场景中的优异表现提供了机制保障。

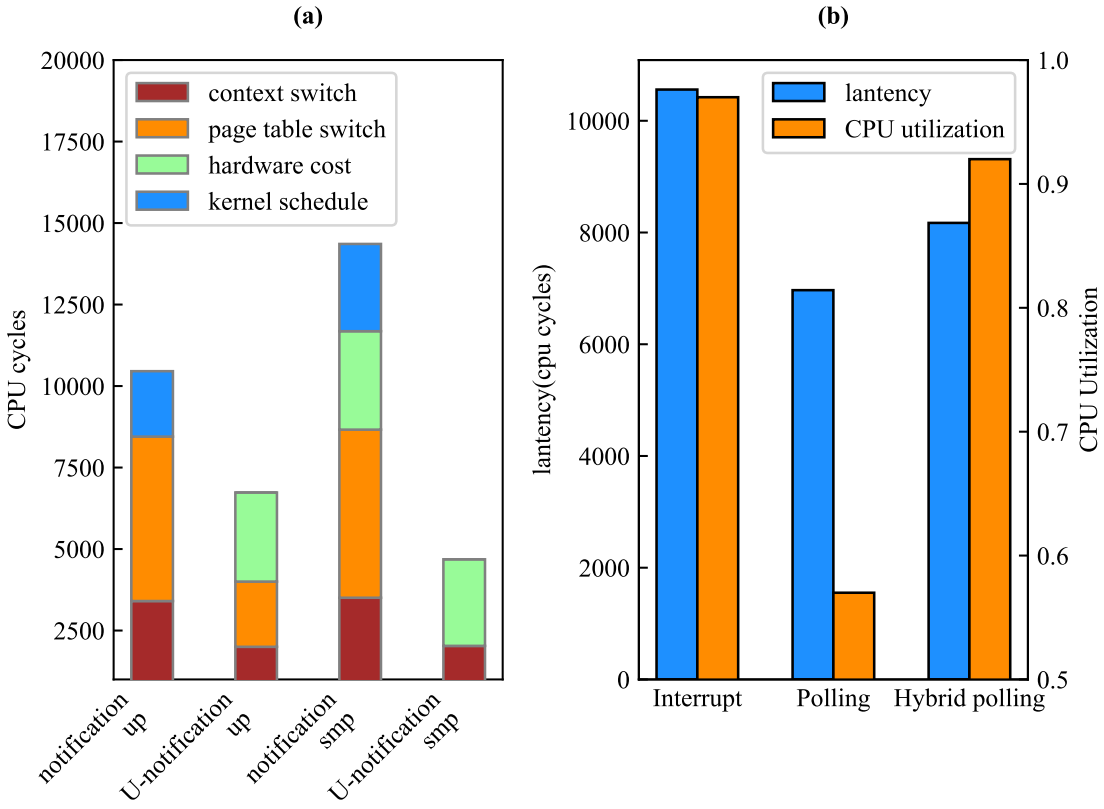

图 5.1　U-notification 与自适应混合轮询的消融实验

5.3.2　内存分配服务器

为深入理解异步系统调用对操作系统性能的影响，本节设计了一个基于用户态内存管理服务的对比实验。实验构建了一个典型的生产者-消费者模型，其中多个工作线程通过消息队列向核心服务线程发送内存操作请求。该设计模拟了现代系统软件中常见的内存管理场景，为评估系统调用性能提供了有说服力的测试场景。

在实验方法上，本节采用了严格的对照设计。通过保持硬件环境、工作负载等条件一致，仅改变系统调用方式（同步/异步）这一独立变量，确保了实验结果的有效性和可比性。测试过程中，我们不仅关注传统的平均 CPU 周期指标，还特别考察了内核交互频率这一关键参数，以全面评估不同系统调用方式对整体系统性能的影响。

实验结果如5.2所示，从系统调用的整体性能趋势来看，基于用户态中断的异步实现的内存分配器展现出显著的可扩展性优势。随着并发度的提高，其性能呈现近似

线性的增长趋势，这一现象在并发度达到 8 时趋于稳定。深入分析表明，这种性能提升主要源于一个关键因素：高并发条件下批处理效应显著降低了内核陷入频率，测试数据显示在 8 并发时内核交互次数较同步情况减少了 90% 以上，特权级切换次数的减少直接降低了上下文保存与恢复的开销，同时还增强了代码局部性。

在同步与异步实现的对比分析中，观察到一个重要的临界点现象。当并发度低于 4 时，同步系统调用反而展现出更好的性能表现。这一看似矛盾的现象可以通过细致的开销分解得到解释：异步机制虽然减少了特权级切换，但引入了额外的运行时管理开销（平均每个请求增加约 2000 个 CPU 周期）。量化分析显示，在并发度为 2 时，异步方案的总开销比同步方案高出约 35%。然而，当并发度超过 4 后，异步方案的优势开始显现。此时，批量处理带来的规模效应使得单次内核陷入可以处理多达 4 个左右请求，特权级切换的开销占比下降，从而实现了性能的反超。而基于 TAIC 的异步系统调用在低并发度下能够显著减少运行时的切换开销，在并发度为 1 的场景下，相比于用户态中断的异步系统调用，TAIC 将性能提升了 74%，在单核情况下，由于需要陷入内核执行系统调用，因此相比于同步仍有所差距，而在多核情况下，基于 TAIC 的系统调用无需陷入内核，避免了特权级的切换，进而比同步系统调用的性能还要高出 30%，这一优势随着并发度的增加逐渐扩大到 70%。

从执行架构的角度考察，多核环境下的性能特征呈现出新的特点。当内核与客户端分布在不同的 CPU 核心上时，系统调用的请求处理与内核执行可以实现真正的并行化。测试数据显示，这种配置下的吞吐量比单核情况提升了约 16%。然而，值得注意的是，在用户态中断的硬件下，由于内核独占一个计算核心，其负载压力相对降低，这导致了一个意外的现象：在多核配置下，客户端陷入内核的频率反而比单核情况高出约 25%。这一现象揭示了内核负载与客户端行为之间复杂的相互作用关系，为后续的系统优化提供了新的研究方向。

5.3.3 同步 IPC vs. 异步 IPC

为深入理解异步 IPC 的性能特性，本节设计了一套乒乓测试方案，通过控制服务端与客户端进程的交互模式，排除了其他系统组件的干扰，专门考察同步与异步 IPC 的路径差异。如5.3所示，实验结果揭示了若干重要的性能特征。

同步 IPC 的性能表现呈现出明显的场景依赖性。在多核环境下，由于 fast-path 检查必然失败，所有 IPC 请求都需要通过核间中断进行传递，导致性能显著下降。测试

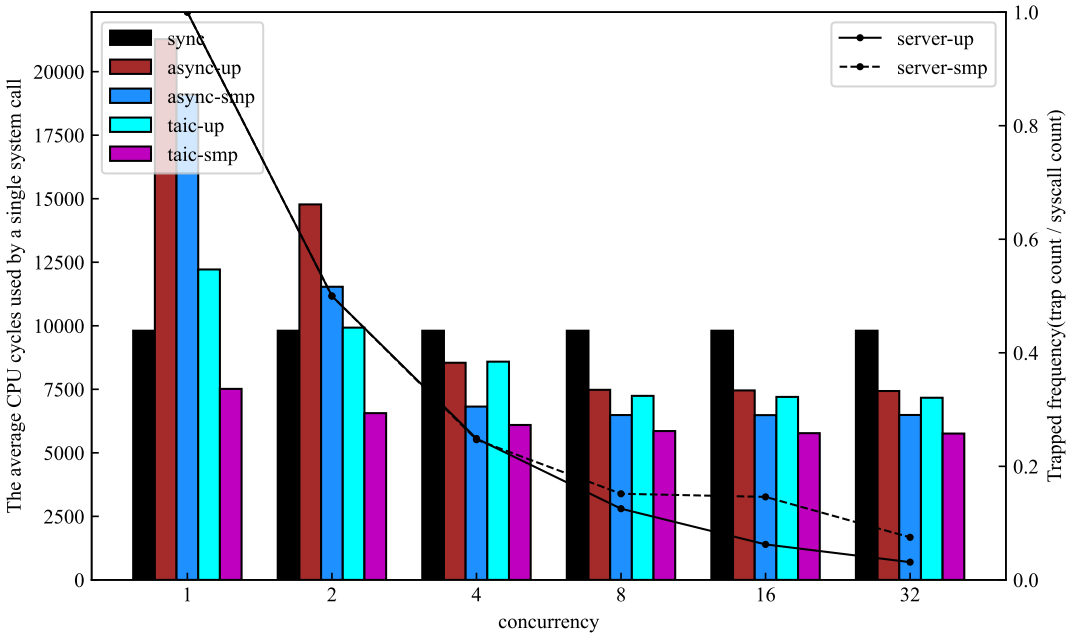

图 5.2　内存分配服务器性能测试实验

数据显示，多核同步 IPC 的延迟比单核情况增加了近 3 倍。而在单核场景下，当满足 fast-path 条件时（即线程优先级匹配且消息长度短），同步 IPC 可以绕过复杂的消息解码和调度流程，性能提升幅度达到 167%。然而，实际应用中的统计表明，由于严格的检查条件限制，fast-path 优化的适用场景相当有限，仅在单核且 C/S 请求模式下的短消息传递才能生效。

异步 IPC 则展现出截然不同的性能特征。随着并发量的增加，其性能呈现明显的改善趋势。这一现象可以从两个层面进行解释：首先，自适应 U-notification 机制会根据负载情况动态调整通知频率，在高并发时减少不必要的中断；其次，批量处理效应使得固定开销得以分摊。值得注意的是，多核环境下的性能优势呈现出有趣的动态变化：在低并发时（1-4 个并发请求），多核配置比单核快 52%，这得益于真正的并行处理；但随着并发度增加（超过 16 个请求），优势逐渐缩小至 17%。深入分析表明，这是由于专用服务端核心的负载不足，导致中断频率过高，反而限制了整体吞吐量。

同步与异步 IPC 的性能对比揭示了一个关键的系统设计权衡。在低并发场景（并发度 =1）下，异步 IPC 由于额外的用户态中断（2 次）和调度器开销，其性能比无 fast-path 的同步 IPC 低 31%，比带 fast-path 的同步 IPC 更低达 249%。然而，随着并发度增加，异步 IPC 的优势快速显现：当并发度达到 8 时，其性能已超过带 fast-path

的同步 IPC；在 32 并发时，性能优势达到 76%。这种动态特性说明，异步 IPC 特别适合现代高并发应用场景，而同步 IPC 则在特定低并发场景下可能更具优势。

而对比 TAIC 加速之后的异步 IPC，我们可以明显看到在低并发场景（并发度 =1）下，异步 IPC 性能提高了 84%，不仅超越了普通同步 IPC，而且接近 fast-path 优化版本。这一提升主要源于 TAIC 的两个关键创新：首先，硬件自动化处理中断信号消除了软件中断处理的开销，改善了程序局部性并减少了上下文保存开销；其次，硬件自动唤醒协程避免了避免了运行时调度器的频繁操作。随着并发度提高，虽然中断优化的收益被均摊，但 TAIC 的唤醒机制优化仍能带来 48% 的性能提升，这主要得益于其减少了运行时调度开销。

综合测试结果表明，异步 IPC 在多核环境下始终展现出良好的性能表现。TAIC 加速技术进一步拓展了其优势范围，使其在低并发场景也具备竞争力。虽然在单核低并发场景下异步 IPC 存在一定劣势，但随着并发度的增加，其性能优势迅速显现，在当今普遍的多核高并发计算环境下，结合 TAIC 等硬件加速技术的异步 IPC 架构具有显著的应用价值。

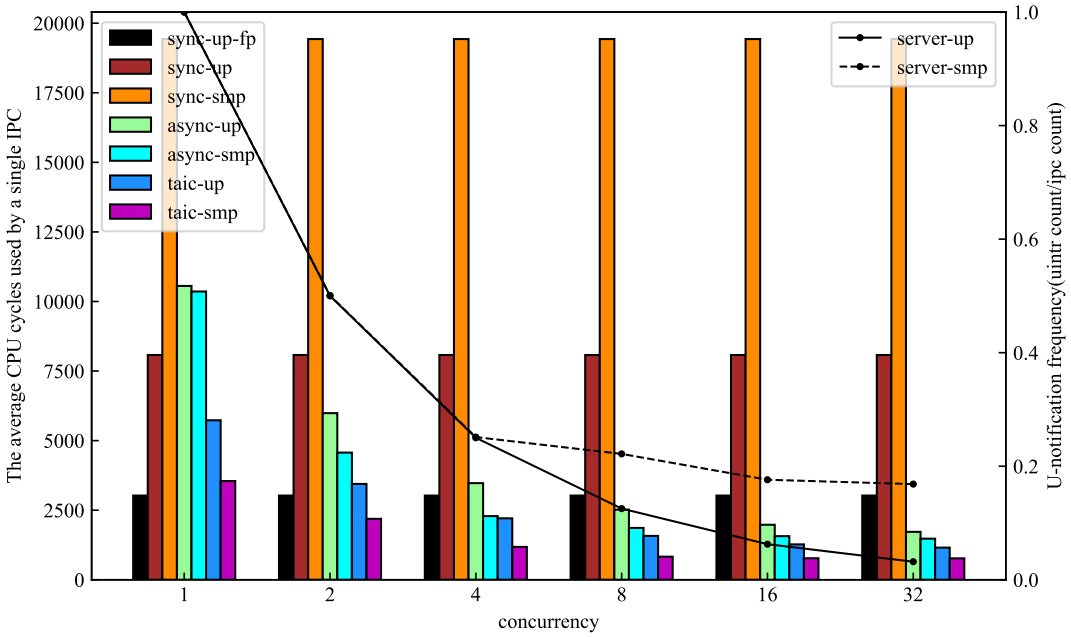

图 5.3　IPC 对比测试实验

5.3.4 TCP 服务器

为验证异步 IPC 在实际应用中的性能优势，本节设计并实现了一个完整的 TCP 服务器基准测试系统。该测试架构采用微内核中典型的模块化设计，充分模拟了真实网络应用场景中的数据处理流程。

如5.4所示，测试系统的客户端部署在标准 PC 上，通过以太网与运行在 FPGA 开发板上的 ReL4 系统建立网络连接。客户端采用多线程架构，每个线程维护独立的 TCP 连接，持续发送 64 字节的小数据包并等待服务器响应，统计响应延迟和吞吐量，用于评估系统在高并发小包处理场景下的性能表现。

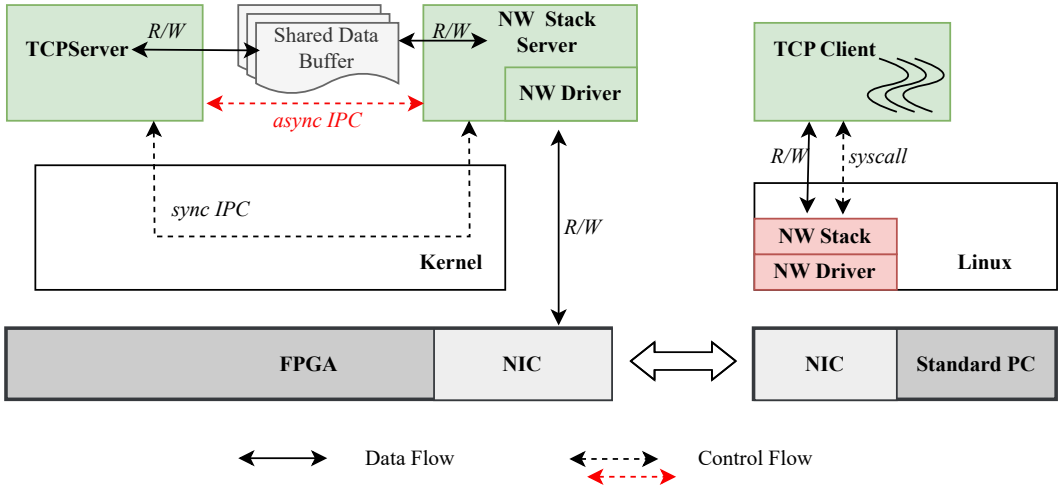

图 5.4 TCP 测试场景示意图

在 ReL4 系统内部，网络处理流程采用模块化设计。网络协议栈服务器（NW Stack Server）作为核心数据平面组件，直接与网卡驱动交互，负责底层数据包的收发和协议处理。该服务器集成了开源的 smoltcp 协议栈实现，能够高效维护大量连接状态信息。通过精心设计的共享缓冲区机制，NW Stack Server 与上层 TCP Server 之间实现了零拷贝数据传输，最大限度地减少了内存复制开销。

TCP Server 作为应用层服务，通过 IPC 机制与 NW Stack Server 进行通信。这种架构设计使得网络协议处理和业务逻辑处理能够并行执行，充分发挥多核处理器的计算能力。性能指标采集方面，客户端会精确测量每个请求的端到端延迟，并统计单位时间内的消息吞吐量，为系统评估提供全面的性能数据。

需要注意的是，由于同步 IPC 在架构上存在固有局限，无法支持连接的多路复用，因此在同步配置下，每个 TCP 连接都需要独立的服务线程进行处理。这种设计虽然

保证了功能正确性，但导致了显著的资源开销和性能下降。相比之下，异步 IPC 架构能够实现真正的连接多路复用，单个服务线程即可高效处理大量并发连接，展现出明显的性能优势。

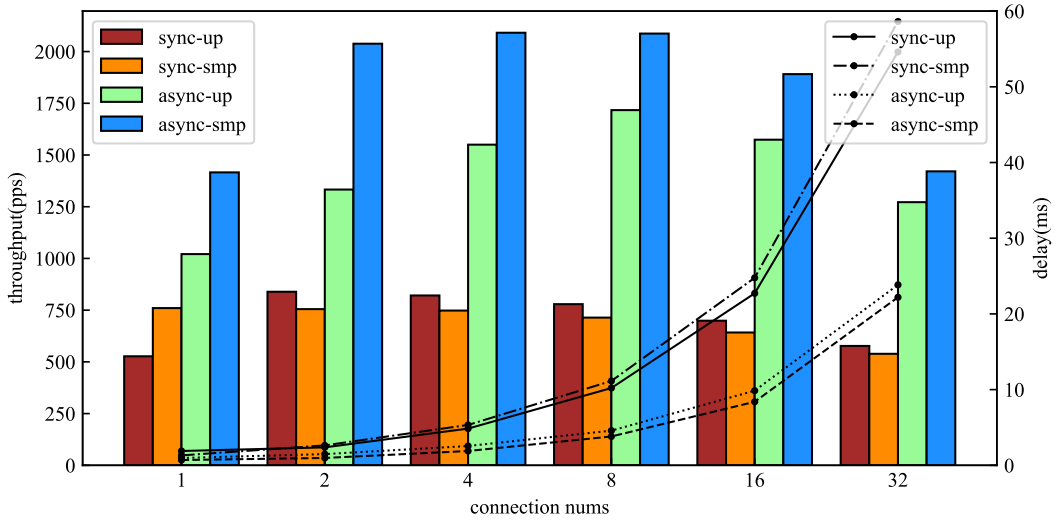

图 5.5　TCP 服务器性能测试实验

如5.5所示，系统吞吐量与延迟随并发度的变化呈现出典型的非线性关系。在初始阶段（1-8 个并发连接），吞吐量随并发度增加而增长，这反映了系统资源尚未达到饱和状态。当并发度超过临界值（约 8 个连接）后，系统进入过载状态，吞吐量开始下降。这一现象可以通过中断处理模型来解释：随着并发连接数增加，网卡中断频率呈线性上升，导致 CPU 将大量时间消耗在中断上下文切换上。同时，共享资源（如协议栈缓冲区）的竞争加剧，进一步降低了系统效率。与吞吐量的抛物线特征不同，端到端延迟在整个测试范围内保持单调递增，这符合排队论的基本原理，即系统负载增加必然导致请求等待时间延长。

同步与异步架构的性能对比展示了微内核异步 IPC 的设计优势。即使在低并发场景（1-4 个连接）下，异步 IPC 架构已展现出显著优势，其吞吐量比同步实现高出 85-120%，平均延迟降低约 40%。这种优势源于异步架构的两个关键特性：首先，它避免了频繁的内核态切换，使得网络中断能够被及时响应；其次，其事件驱动模型消除了传统多线程架构中的上下文切换开销。随着并发度增加，同步实现的多线程架构暴露出严重的扩展性问题。在 4 个连接时，性能差距达到峰值（192%）。值得注意的是，即使在最大测试负载下，异步架构仍保持 120% 的性能优势，证明了其在高压环

境下的稳定性。

多核扩展性分析揭示了更深层次的架构差异。异步 IPC 实现能够有效利用多核资源，在双核配置下最好情况能实现接 52% 的加速比，同时将平均延迟降低约 52%。这种良好的扩展性得益于异步 IPC 几乎无需内核参与，避免了内核锁的竞争，实现了最大化并行。相比之下，同步 IPC 在多核环境中的表现反常地劣于单核情况（性能下降约 7%）。通过性能剖析发现，这种退化主要来自两个方面：核间中断引入的额外开销，以及核间中断导致陷入内核而产生的 TLB shootdown 性能惩罚。特别是在小数据包处理场景下，同步实现的核心局部性显著恶化，进一步放大了多核环境下的性能劣势。

上面的测试结果表明：异步 IPC 在实际应用场景中有利于多路复用的实现，可以有效减少特权级切换的开销，同时提升系统的多核扩展性，进而提升系统的整体性能。

5.4 本章小结

本章通过系统的实验评估，从功能正确性、兼容性和运行效率三个维度对 ReL4 微内核系统进行了全面验证。实验结果表明，ReL4 在保持与 seL4 高度兼容的同时，通过创新的异步架构设计和硬件加速机制，显著提升了微内核系统的整体性能。

在功能验证方面，基于 seL4test 标准测试框架的实验数据表明，ReL4 能够完整支持微内核的核心功能模块，包括任务管理、内存管理和能力机制等。特别值得注意的是，除与异步架构设计直接相关的少数测试项外，系统展现出与 seL4 良好的兼容性，这为现有 seL4 生态向 ReL4 的平滑迁移提供了重要保障。

性能评估部分通过多组对照实验揭示了 ReL4 架构的性能优势。消融实验结果表明，用户态通知机制（U-notification）相比传统内核通知方式，在多核环境下可降低 45% 的延迟；而自适应混合轮询策略则实现了延迟与 CPU 利用率的最佳平衡。系统调用性能测试显示，异步机制在高并发场景（>32 并发度）下展现出显著的批处理效应，使内核交互频率降低 90% 以上。IPC 性能对比实验进一步证实，异步 IPC 相比于同步 IPC，在高并发场景下有着显著优势，TAIC 硬件加速技术使异步 IPC 在低并发场景下的性能提升达 84%，接近同步 IPC 的性能，同时保持了在高并发场景下的优势。

实际应用场景的测试结果更具说服力。TCP 服务器基准测试表明，即使在低并发条件下，异步架构的吞吐量仍比同步实现高出 85-120%，且随着并发度增加，性能优势持续扩大。多核环境下的测试数据尤其值得关注：异步架构在双核配置下实现了接

近线性的加速比，而同步实现反而因核间通信开销出现性能退化，这一对比充分证明了异步架构在多核环境下的扩展性优势。

综合实验数据可以得出，ReL4 通过体系结构创新，在保证功能正确性的前提下，有效解决了传统微内核系统在性能方面的固有瓶颈。异步 IPC 架构与硬件加速技术的结合，使系统在保持微内核安全特性的同时，获得了接近宏内核的性能表现，为高安全需求场景下的系统设计提供了新的技术路径。

结论

本文针对微内核系统中进程间通信（IPC）的性能瓶颈问题，提出并实现了基于用户态中断的高性能异步微内核 ReL4。通过系统性地重构传统微内核架构，本研究取得了以下创新性成果：

首先，设计并实现了异步微内核 ReL4，在保持与 seL4 API 基本兼容的前提下，提出了一种纯异步的 IPC 机制，将同步 IPC 完全移出内核，仅保留异步通知机制，显著简化了内核设计。这一架构创使得系统中的内核参与度显著降低，为性能提升奠定了基础。

其次，基于 UINTC 硬件支持，在 ReL4 中创新性地设计了异步通知机制（U-notification）。该机制通过用户态中断技术有效减少了特权级切换开销，实验数据显示，与传统内核通知机制相比，U-notification 在多核环境下可降低 45% 的延迟，同时避免了核间通信带来的性能退化。

第三，基于上述异步通知机制，在 ReL4 中系统性地实现了异步 IPC 和异步系统调用架构。通过精心设计的用户态异步运行时系统，不仅提升了编程易用性，还实现了显著的性能优势。同时基于 TAIC 支持，异步调度的部分工作卸载到硬件，进一步提升低并发性能。

本研究提出的异步架构特别适用于高并发、上下文无关的通信场景。在低并发条件下，通过 TAIC 加速器和用户态中断技术的协同优化，有效弥补了异步运行时引入的额外开销。虽然测试数据显示在极低并发（<4 个请求）场景下性能仍略逊于同步实现，但当并发度提升至 8 以上时，异步架构即展现出明显的性能优势。这一特性使其能够很好地适应现代计算环境的高并发需求。

未来工作可考虑从以下方面继续优化：首先，通过硬件加速实现异步运行时中的其他关键操作，如协程调度等，以进一步消除运行时开销；其次，探索更智能的自适应策略，使系统能够根据负载特征动态调整运行模式，在各种负载条件下均能保持卓越性能。这些优化将进一步提升异步微内核架构的实用价值和应用范围。

参考文献

[1] Rana M R, Baul S. A survey on microkernel based operating systems and their essential key components[J]. Available at SSRN 4467406, 2023.

[2] Abhilash T. Microkernel vs monolithic kernel design trade - offs[J]. International Journal For Multidisciplinary Research, 2024, 6(2): 1 – 5.

[3] Heiser G. The role of virtualization in embedded systems[C]. IIES '08: Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems. New York, NY, USA: Association for Computing Machinery, 2008: 11–16.

[4] Herder J N, Bos H, Gras B, et al. Construction of a highly dependable operating system[C]. 2006 Sixth European Dependable Computing Conference. [S.l.]: IEEE, 2006: 3–12.

[5] Liedtke J. Toward real microkernels[J]. Communications of the ACM, 1996, 39(9): 70–77.

[6] Liedtke J. Improving ipc by kernel design[C]. SOSP '93: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 1993: 175–188.

[7] Jochen L. On μ-kernel construction[J]. Proc 15th SOSP, 1994, 1994.

[8] Klein G, Elphinstone K, Heiser G, et al. sel4: Formal verification of an os kernel[C]. Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. New York, NY, USA: Association for Computing Machinery, 2009: 207–220.

[9] Caruso J. 1 million iops demonstrated[M]. Framingham, MA, USA: Network World, 2021.

[10] Lipp M, Schwarz M, Gruss D, et al. Meltdown: Reading kernel memory from user space[J]. Communications of the ACM, 2020, 63(6): 46–56.

[11] Kocher P, Horn J, Fogh A, et al. Spectre attacks: Exploiting speculative execution[J]. Communications of the ACM, 2020, 63(7): 93–101.

[12] kernel development community T. Page table isolation (pti)[M]. Portland, OR, USA: The Linux Kernel Organization, 2021.

[13] Blackham B, Shi Y, Heiser G. Improving interrupt response time in a verifiable protected microkernel [C]. Proceedings of the 7th ACM european conference on Computer Systems. New York, NY, USA: Association for Computing Machinery, 2012: 323–336.

[14] Heiser G, Elphinstone K. L4 microkernels: The lessons from 20 years of research and deployment [J]. ACM Transactions on Computer Systems (TOCS), 2016, 34(1): 1–29.

[15] Peng X, Xiao K, Li Y, et al. Fast interprocess communication algorithm in microkernel[J]. International Journal of Performability Engineering, 2020, 16(2): 185.

[16] Arm architecture reference manual: Asid management in armv7[M]. [S.l.]: ARM Limited, 2009.

[17] Cox G, Bhattacharjee A. Efficient address translation for architectures with multiple page sizes[J]. ACM SIGPLAN Notices, 2017, 52(4): 435–448.

[18] Kuang J, Waddington D G, Tian C. Towards a scalable microkernel personality for multicore processors[C]. Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings 19. Berlin, Heidelberg: Springer-Verlag, 2013: 620–632.

[19] Li C, Ding C, Shen K. Quantifying the cost of context switch[C]. Proceedings of the 2007 workshop on Experimental computer science. New York, NY, USA: Association for Computing Machinery, 2007: 2–es.

[20] Götzfried J, Eckert M, Schinzel S, et al. Cache attacks on intel sgx[C]. Proceedings of the 10th European Workshop on Systems Security. New York, NY, USA: Association for Computing Machinery, 2017: 1–6.

[21] Mi Z, Li D, Yang Z, et al. Skybridge: Fast and secure inter-process communication for microkernels [C]. Proceedings of the Fourteenth EuroSys Conference 2019. New York, NY, USA: Association for Computing Machinery, 2019: 1–15.

[22] Li W, Xia Y, Chen H, et al. Reducing world switches in virtualized environment with flexible cross-world calls[J]. ACM SIGARCH Computer Architecture News, 2015, 43(3S): 375–387.

[23] Du D, Hua Z, Xia Y, et al. Xpc: architectural support for secure and efficient cross process call[C]. Proceedings of the 46th International Symposium on Computer Architecture. New York, NY, USA: Association for Computing Machinery, 2019: 671–684.

[24] Kuo H C, Williams D, Koller R, et al. A linux in unikernel clothing[C]. Proceedings of the Fifteenth European Conference on Computer Systems. New York, NY, USA: Association for Computing Machinery, 2020: 1–15.

[25] Olivier P, Chiba D, Lankes S, et al. A binary-compatible unikernel[C]. Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. New York, NY, USA: Association for Computing Machinery, 2019: 59–73.

[26] Yu K, Zhang C, Zhao Y. Web service appliance based on unikernel[C]. 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). 2017: 280–282.

[27] Zhou Z, Bi Y, Wan J, et al. Userspace bypass: Accelerating syscall-intensive applications[C]. 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). Boston, MA: USENIX Association, 2023: 33–49.

[28] Jeong E Y, Woo S, Jamshed M, et al. mtcp: a highly scalable user-level tcp stack for multicore systems[C]. NSDI'14: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. USA: USENIX Association, 2014: 489–502.

[29] Yang Z, Harris J R, Walker B, et al. Spdk: A development kit to build high performance storage applications[C]. 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). 2017: 154–161.

[30] Soares L, Stumm M. Flexsc: flexible system call scheduling with exception-less system calls[C]. OSDI'10: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. USA: USENIX Association, 2010: 33–46.

[31] Nassif N, Munch A O, Molnar C L, et al. Sapphire rapids: The next-generation intel xeon scalable processor[C]. 2022 IEEE International Solid-State Circuits Conference (ISSCC): volume 65. 2022: 44–46.

[32] International R V. The risc-v instruction set manual, volume ii: Privileged architecture, document version 1.12[M]. Zurich, Switzerland: RISC-V International, 2020.

[33] Levy A, Andersen M P, Campbell B, et al. Ownership is theft: Experiences building an embedded os in rust[C]. Proceedings of the 8th Workshop on Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2015: 21–26.

[34] Balasubramanian A, Baranowski M S, Burtsev A, et al. System programming in rust: Beyond safety [C]. Proceedings of the 16th workshop on hot topics in operating systems. New York, NY, USA: Association for Computing Machinery, 2017: 156–161.

[35] Heiser G, Leslie B. The okl4 microvisor: Convergence point of microkernels and hypervisors[C]. Proceedings of the first ACM asia-pacific workshop on Workshop on systems. New York, NY, USA: Association for Computing Machinery, 2010: 19–24.

[36] Smejkal T, Lackorzynski A, Engel B, et al. Transactional ipc in fiasco. oc[J]. OSPERT 2015, 2015, 19.

[37] Mehta S. x86 User Interrupts support[J]. LWN.net, 2021.

[38] Pinto S, Garlati C. User mode interrupts[M]. [S.l.: s.n.].

[39] Sutter H, Larus J. Software and the concurrency revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry.[J]. Queue, 2005, 3(7): 54–62.

[40] Adya A, Howell J, Theimer M, et al. Cooperative task management without manual stack management.[C]. USENIX Annual Technical Conference, General Track. USA: USENIX Association, 2002: 289–302.

[41] Deligiannis P, Donaldson A F, Ketema J, et al. Asynchronous programming, analysis and testing with state machines[C]. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2015: 154–164.

[42] Bovet D P, Cesati M. Understanding the linux kernel[M]. [S.l.]: O'Reilly Media, 2005.

[43] Belson B, Xiang W, Holdsworth J, et al. C++ 20 coroutines on microcontrollers—what we learned [J]. IEEE Embedded Systems Letters, 2020, 13(1): 9–12.

[44] Kröning M W. Concurrency techniques and hardware abstraction layer concepts for embedded systems in rust[D]. [S.l.]: Universitätsbibliothek der RWTH Aachen, 2023.

[45] Matsakis N, Turon A. Async/await in rust: A full proposal[J]. Rust RFC 2394, 2018.

[46] Samson C F. Asynchronous programming in rust: Learn asynchronous programming by building working examples of futures, green threads, and runtimes[M]. [S.l.]: Packt Publishing Ltd, 2024.

[47] Edwards J. Coherent reaction[C]. OOPSLA '09: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications. New York, NY, USA: Association for Computing Machinery, 2009: 925–932.

[48] Tjäder H. Rtic - a zero-cost abstraction for memory-safe concurrency[M]. Ithaca, NY, USA: arXiv, 2021.

[49] Klabnik S. Rust for rustaceans[M]. [S.l.]: No Starch Press, 2020.

[50] Moslehian A S. An experimental integration of io_uring and tokio: An asynchronous runtime for rust[R]. Mashhad, Iran: Ferdowsi University of Mashhad, 2022.

[51] Developers T R P. Rust standard library futures module[M]. San Francisco, CA, USA: The Rust Foundation, 2021.

[52] Waterman A, Asanović K. The risc-v instruction set manual: volume 1[M]. Berkeley, CA, USA: UCB EECS-2016-1, 2016.

[53] taic repo. taic[EB/OL]. GitHub, 2023. https://github.com/taic-repo/taic.

[54] Song W J. Hardware accelerator systems for embedded systems[M]. Advances in Computers: volume 122. [S.l.]: Elsevier, 2021: 23–49.

[55] Doran M A, Kandalaft N. Embedded virtualization on risc-v with sel4[C]. 2023 IEEE 14th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON). [S.l.]: IEEE, 2023: 0736–0740.

[56] Dude M. sel4 design principles[EB/OL]. 2020. https://microkerneldude.org/2020/03/11/sel4-design-principles/.

[57] Klein G, Andronick J, Elphinstone K, et al. Comprehensive formal verification of an os microkernel [J]. ACM Transactions on Computer Systems (TOCS), 2014, 32(1): 1–70.

[58] De Matos E, Lunardi W T, Ukkonen J, et al. An sel4-based trusted execution environment on risc-v [C]. 2024 International Wireless Communications and Mobile Computing (IWCMC). [S.l.]: IEEE, 2024: 712–717.

[59] Lyons A, McLeod K, Almatary H, et al. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time[C]. Proceedings of the Thirteenth EuroSys Conference. New York, NY, USA: Association for Computing Machinery, 2018: 1–16.

[60] Peters S, Danis A, Elphinstone K, et al. For a microkernel, a big lock is fine[C]. Proceedings of the 6th Asia-Pacific Workshop on Systems. New York, NY, USA: Association for Computing Machinery, 2015: 1–7.

[61] Barnes G. A method for implementing lock-free shared-data structures[C]. Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures. New York, NY, USA: Association for Computing Machinery, 1993: 261–270.

[62] Zynq ultrascale+ mpsoc technical reference manual[M]. San Jose, CA, USA: Xilinx Inc., 2023.

[63] Amba axi and ace protocol specification[M]. Cambridge, UK: ARM Limited, 2023.

[64] RISC-V Software Source. RISC-V Open Source Supervisor Binary Interface (OpenSBI)[M]. Berkeley, CA, USA: RISC-V International, 2023.

[65] Axima Power. AXINET | Axima-power.com[M]. Hamburg, Germany: Axima Power, 2023.

[66] AMD. AXI DMA Controller[M]. Santa Clara, California, USA: AMD, 2023.

[67] smoltcp-rs Development Team. smoltcp: a smol tcp/ip stack[M]. San Francisco, California, USA: GitHub, Inc., 2023.

[68] seL4 Project Contributors. seL4/sel4test: Test suite for seL4[M]. San Francisco, California, USA: GitHub, Inc., 2023.

攻读学位期间发表论文与研究成果清单

（二）发表的学术论文

[1] 廖东海，陆慧梅，陈伟豪，赵方亮，向勇. ReL4：高性能异步微内核设计与实现 [J]. 小型微型计算机系统，2025. (录用，IF=0.975)

[2] hao F, Liao D, Wu J, et al. COPS: A coroutine-based priority scheduling framework perceived by the operating system[C]. 024 International Conference on Ubiquitous Computing and Communications(IUCC). Chengdu, China: IEEE, 2024. (EI 收录)

致谢