摘 要

应用需求的复杂化推动着操作系统的任务调度与中断响应机制持续演进，然而现有机制在应对急剧增长的性能需求时仍面临诸多挑战。论文以软硬协同的方法为指导，以减小开销为目标展开研究，主要研究内容与贡献如下：

（1）为解决多种任务模型并存引起的调度复杂性增加，论文建立了基于执行流的任务模型。该模型在进程、线程和协程任务模型的基础上，剥离出进程具备的隔离特性，线程和协程具备的并发特性，将处于不同的地址空间、特权级和堆栈的任务统一起来，降低了调度的复杂性。

（2）为减小任务调度的开销，论文基于执行流的任务模型，设计了基于软硬协同的任务状态维护方法。该方法采用分层复合型任务标识来表达任务执行环境，利用硬件维护就绪队列、阻塞队列和当前执行任务，从而高效地实现负载均衡和任务状态修改。论文基于软硬协同的任务状态维护方法实现了硬件化的任务调度器，实验结果表明，使用硬件化的任务调度器的 tokio 运行时的调度开销降低了 30%。

（3）为了减小中断导致的负面影响，论文设计了硬件响应中断机制。该机制由软件预先在硬件中维护好阻塞任务与等待的中断事件的关系，当硬件收到中断信号后，直接将处于阻塞状态的任务唤醒，减小了中断信号打断 CPU 所导致的开销，以及软件定期轮询外部设备状态导致的开销。经过硬件中断响应机制改造后的网卡驱动使得 Redis 服务的吞吐量提升了 7%。论文还将该机制扩展到软件中断的处理上，实现了基于硬件的任务通知机制。该通知机制在硬件中建立了跨越不同地址空间、特权级的任务之间的直接通道，避免了大量的地址空间、特权级切换开销，将基于信号机制的任务通知的开销降低了 98.8%，将 ReL4 中的异步 IPC 开销降低了 45.7% 和 65.8%。

关键词：软硬协同；任务模型；任务调度；中断响应；任务通知

Abstract

The complexity of application requirements has promoted the continuous evolution of task scheduling and interrupt handling mechanisms of operating systems, but the existing mechanisms still face many challenges when dealing with the rapidly increasing performance requirements. Guided by the method of software-hardware collaboration, this paper carries out research with the goal of reducing the cost. The main research contents and contributions are as follows:

(1) In order to solve the increased scheduling complexity caused by the coexistence of multiple task models, this paper establishes a task model based on execution flow. Based on the task models of process, thread and coroutine, the execution-flow model strips out the isolation characteristic of process and the concurrency characteristic of thread and coroutine, and unifies the tasks in different address spaces, privilege levels and stacks, reducing the scheduling complexity.

(2) In order to reduce the overhead of task scheduling, this paper designs a software-hardware collaborative task state maintenance method based on the execution-flow model. This method uses hierarchical and complex task identification to express tasks' execution environment, and uses hardware maintenance ready queue, blocking queue and current execution task to achieve load balancing and task state modification efficiently. Based on this task state maintenance method, this paper implements a hardwareized task scheduler. Experimental results show that the scheduling overhead of tokio runtime using the hardwareized task scheduler is reduced by 30%.

(3) In order to reduce the negative impact caused by interrupt, this paper designs a hardware-based interrupt handling mechanism. In this mechanism, the software maintains the relationship between the blocked task and the waiting interrupt event in the hardware in advance. When the hardware receives the interrupt, it directly wakes up the task in the blocked state, which reduces the overhead of interrupting the CPU and the overhead caused by the software regularly polling the status of external devices. The netcard driver modified by the hardware-based interrupt handling mechanism improves the throughput of Redis service by 7%. This paper also extends this mechanism to software interrupt handling and implements a hardware-based task notification mechanism. The notification mechanism establishes a direct channel between tasks across different address spaces and

privilege levels in the hardware, avoiding a lot of address space and privilege switching overhead, reducing the overhead of task notifications based on the signal mechanism by 98.8%, and reducing the asynchronous IPC overhead in ReL4 by 45.7% and 65.8%.

Keywords: Software-Hardware Collaboration; Task Model; Task Scheduling; Interrupt Handling; Task Notification

第 1 章　引言

1.1　研究背景与意义

近年来，云应用程序（例如网络搜索、社交网络、电子商务、实时媒体等）在日常生活中发挥着越来越重要的作用。为了保证良好的人机交互体验，这些应用程序需要在几十毫秒的时间尺度上对用户的操作作出响应，而它们通常将单个请求分散到数据中心的数百台计算机上运行的数千个通信服务，其中耗时最长的服务成为影响端到端响应时间的主要因素，这要求每个参与的服务的尾部延迟在几百微秒的范围内；随着这些亿级用户应用的爆发式增长，数据中心面临的性能挑战已远超人机交互的低延迟需求，数据中心需要每分钟能够处理数百万请求，现代计算机系统正经历着前所未有的吞吐量压力考验。

嵌入式实时系统应用也呈现出相似的发展趋势，其应用场景逐渐泛化，从传统的工业控制领域（工业化流控、车辆制造等）向消费级场景（智能家居、健康监护、车辆智能座舱等）逐渐渗透，催生出与新型技术融合的需求，模糊了工业与消费电子领域对嵌入式实时系统的性能需求的传统界限。在消费端应用中，嵌入式实时系统开始集成轻量化人工智能模块，在要求系统保留实时响应能力的同时，还需要实现微型机器学习等新型功能；在车载智能座舱领域，嵌入式实时系统需要支持多媒体数据处理、车载网络通信、车辆控制等多种功能，既需要达到工业级安全标准的硬实时性要求，又需要满足娱乐性、互动性等消费级体验需求。

任务调度作为操作系统的核心机制，其作用不仅体现在对 CPU 时间的直接分配上，而且与中断机制紧密协作，负责管理任务执行和事件处理。任务调度通过使用合理的调度策略来优化 CPU 利用率和其他性能需求，而中断机制则确保了系统对异步事件的及时响应，两者相辅相成，共同保证了系统的稳定性和高效性。应用需求的提高和应用场景的复杂化对任务调度提出了更高的要求。云应用场景的高吞吐量与低延时需求要求任务调度需要具备高效的任务分配能力、动态调整能力和快速响应能力，能够在短时间内完成大量任务的分发，有效应对负载波动，确保任务及时响应，而且还需要考虑任务的局部性等特征；嵌入式场景下的实时性与非实时性混合需求则要求任务调度机制不仅需要保证实时性任务在规定时间内完成，具有确定的响应时间，而且还要求任务调度机制能够合理安排非实时性任务，在不影响实时性任务的前提下，提高系统的资源利用率，甚至还需要考虑任务对设备能耗的影响，延长设备使用寿命等需求。因此，针对操作系统中的任务调度和中断响应机制展开研究，是计算机系统研究的重要方向。

1.2　关键问题与挑战

当前已有研究从多个角度对任务调度进行了研究，主要集中在使用轻量级任务模型提高并发度、针对特定性能需求设计任务调度算法以及提供通用调度框架，在用户态自定义调度策略等方面；而针对中断机制展开的研究主要集中在使用轮询、中断合并等机制来减小中断开销，提高系统吞吐量；也存在着一些研究从软硬协同的角度出发，设计硬件加速器来节约任务切换开销，解决优先级反转等问题。尽管已有研究已经极大程度地推动了任务调度和中断响应的发展，但仍然存在以下问题和挑战。

问题一：多种任务模型并存导致调度复杂性增加。在使用轻量级的任务模型来提高并发度的研究方向上，系统中同时存在进程、内核线程、内核协程、用户态线程、用户态协程等多种任务类型，不同的任务类型使用不同的调度器，形成多重调度机制，增加了调度的复杂性。由于地址空间和特权级等保护机制，内核的调度器无法感知到用户态的任务，多个用户态任务绑定到单个内核线程上，导致高优先级的用户态任务可能阻塞在低优先级的内核线程上。

问题二：任务调度开销不可忽视。在针对性能需求定制调度算法的研究方向上，已有工作的关注点在于如何通过调度算法来安排任务的调度顺序，而很少关注到任务调度过程中由于上下文切换以及负载均衡等策略导致的开销。任务分配的时间片是固定的，一旦任务调度过程中的开销过大，就会导致任务的实际执行时间下降①，对系统的吞吐量、延时造成负面影响。

问题三：中断导致的负面影响。中断机制对于保证系统的实时性以及降低系统的响应延时起到了关键性作用，但由于其固有特性，它需要打断 CPU 正在执行的任务，切换到中断处理例程来处理中断事件，在完成处理后回到原来的任务上继续执行，这种切换会导致在任务执行过程中产生额外的上下文切换，以及其他的如缓存污染等间接开销，降低系统的吞吐量。因此，系统需要考虑到中断机制的负面影响，对吞吐量和延时这两项性能指标进行权衡。

1.3　论文主要贡献

论文针对第 1.2 节中提到的任务调度和中断响应机制中存在的挑战，以软硬协同的方法为指导，在软件和硬件层面上展开研究，降低任务调度和中断响应机制的开销，从而降低系统的响应延时、提高系统的吞吐量，以应对复杂多样化的应用

① 在任务切换过程中，调度代码以及上下文切换代码属于一段特殊的代码，它与任务实际的功能逻辑无关，但隶属于每个任务，当任务运行到切换代码时，调度以及切换代码即属于那个任务，执行这些代码需要占用任务的时间片。

场景需求。论文的主要贡献如下：

（1）针对问题一，论文建立了基于执行流[①]的任务模型，将进程、线程、协程统一到基于执行流的任务模型中，降低了调度的复杂性。论文在已有的进程、线程和协程任务模型的基础上，分析了它们在系统中各自承担的角色，剥离出进程具备的隔离特性，线程和协程具备的并发特性，建立了基于执行流的统一任务模型，从而使得处于不同地址空间、特权级的任务统一到一个维度上，能够在一个调度器中进行调度，降低了调度的复杂性。

（2）针对问题二，论文设计了基于软硬协同的任务状态维护方法，减小任务调度过程中的开销。论文基于执行流的任务模型，为任务设计了分层复合型任务标识，使得硬件可以通过任务标识来感知任务所处的地址空间、特权级以及优先级等信息。硬件在内部维护了就绪队列、阻塞队列以及正在运行的任务标识，从而可以感知任务所处的状态，通过在这些队列之间迁移任务标识来实现高效的任务状态变迁以及负载均衡机制。论文基于软硬协同的任务状态维护方法实现了硬件化的任务调度器，实验结果表明，使用硬件化的任务调度器的 tokio 运行时的调度开销降低了 30%。

（3）针对问题三，论文设计了硬件响应中断机制，消除了中断机制导致的负面影响。软件预先在硬件中维护好阻塞任务与等待的中断事件的关系，当硬件收到中断信号后，直接将处于阻塞状态的任务唤醒，减小了中断信号打断 CPU 所导致的开销，以及软件定期轮询外部设备状态导致的开销。经过硬件中断响应机制改造后的网卡驱动使得 Redis 服务的吞吐量提升了 7%。论文还将该机制扩展到软件中断的处理上，实现了基于硬件的任务通知机制。该通知机制在硬件中建立了跨越不同地址空间、特权级的任务之间的直接通道，避免了大量的地址空间、特权级切换开销，将基于信号机制的任务通知的开销降低了 98.8%，将 ReL4 中的异步 IPC 开销降低了 45.7% 和 65.8%。

1.4　论文的主要研究内容与组织结构

论文第 1 章为引言。作者首先结合云应用和嵌入式实时应用的发展趋势，阐述了论文的研究背景，强调了任务调度在操作系统中的重要性，简要描述了对任务调度和中断处理机制的研究现状以及其面临的挑战，并总结了论文的主要贡献。

论文第 2 章为相关技术研究，作者在第 1 章的简要描述基础上对任务模型、任务调度、中断机制和 I/O 模型等相关技术进行了详细描述，总结了软硬协同发展过

① 执行流是指在 CPU 上执行的用于完成某项具体功能的指令序列，它既包含了程序代码在运行过程中的指令执行顺序和逻辑路径的动态描述（如顺序执行、条件分支、循环跳转等），也包含了对 CPU 上的硬件寄存器状态（控制寄存器和其他通用寄存器等）的描述。

程中的一些关键技术演化过程，为论文后续的研究工作提供了理论基础。

论文第 3 章基于前文的研究内容，展开了基于软硬协同的任务调度和中断响应机制设计。作者提出基于执行流的任务模型，以此模型为基础，设计分层复合型任务标识和基于软硬协同的任务状态维护方法，由硬件实现任务调度、中断处理以及任务通知等功能，并且定义了一套接口用于软硬件之间的交互。

论文第 4 章将第 3 章所述的设计方案与具体的硬件平台以及操作系统内核结合，落实到具体的硬件实现，以及与软件之间的适配，搭建了一个基于软硬协同的任务调度和中断响应机制的系统原型。

论文第 5 章对前两章所提出的设计方案与系统原型进行了验证，通过微基准测试以及综合性能测试，验证了设计方案的有效性。

论文第 6 章是总结与展望，作者在总结全文的基础上，指出了后续可能的研究方向。

第 2 章　相关技术研究

本章围绕着与任务调度和中断响应的相关技术展开探讨。首先从任务模型切入，揭示了计算机系统如何通过软硬协同设计推动任务模型的持续演进——从早期的单任务串行执行逐步发展为支持多任务抢占式调度的复杂架构；然后从不同应用场景对延时、吞吐量、实时性等性能指标的需求展开对任务调度和中断机制这两个研究方向的讨论，最后介绍了 I/O 模型的发展趋势。

2.1　任务模型

操作系统任务模型的演化历程深刻反映了计算范式的变革与软硬协同设计的迭代进步，其发展轨迹始终与底层硬件架构革新及并发编程理论突破紧密耦合。

2.1.1　批处理作业模型

早期计算机只能一次运行一个程序，每个用户带着程序和数据来到计算机前，装载程序，运行并调试，在完成后取走输出的结果，在这段时间内，用户对机器拥有唯一控制权。随着计算机速度的提升，人工装载和卸载占用的时间逐渐增加，计算机资源利用率逐渐降低。为了提高计算机资源利用率，批处理作业模型应运而生。它使用监控程序来处理一系列或批处理的程序。监控程序自动加载批处理程序中的第一个任务，在任务结束后，监控程序重新获得计算机的控制权并加载运行下一个任务，直到批处理完成，减少人工操作时间。

批处理作业模型的核心思想在于将任务或数据集合处理，通过减少任务切换开销和资源闲置时间来优化系统的整体效率。这种思想不仅应用于早期操作系统，也深刻影响了现代计算系统设计，例如在大数据场景中通过批量处理提升吞吐量，甚至在某些场景下通过聚合操作降低宏观层面的延迟。批处理作业模型处理了一个执行流从开始到结束的整个过程，并实现了到下一个执行流的平滑过渡，通过引入监控程序，将原本需要人工直接与计算机硬件之间的协作转换为监控程序与计算机硬件之间的自动协作。这种转换不仅减少了人工干预，提高了作业处理的自动化水平，还通过优化任务调度和资源分配，显著地提升了计算机硬件的利用率和系统整体效率。

2.1.2　进程模型

随着计算机速度的进一步提升，当任务在等待 I/O 操作时，CPU 仍然处于闲置状态，导致整体效率受到影响，尽管可以通过多道程序设计（Multiprogramming）提高 CPU 的利用率，但多个程序之间没有形成保护边界，存在资源竞争和安全隐患。这些问题促使了进程模型的提出。Saltzer 于 1966 年完善了进程作为虚拟处理器的形式化定义，Dijkstra 在 THE 系统中首次提出“协同顺序进程”概念。

进程模型引入了状态机模型，其基础的三状态模型包括就绪态（已获资源但等待 CPU）、运行态（占用 CPU 执行指令）和阻塞态（因等待 I/O 等事件暂停执行），扩展的五状态模型增加了创建态（分配初始资源）和终止态（资源回收），而七状态模型进一步引入了挂起状态（如就绪挂起和阻塞挂起），用于处理内存不足时进程被换出到磁盘的情况。状态转换由事件触发，例如处于运行态的任务因为时间片耗尽转为就绪态，或主动请求资源进入阻塞态，待事件完成后再恢复为就绪态。进程模型使用 PCB 来描述进程的状态以及对系统资源的占用情况（包括程序计数器、通用寄存器、内存映像、文件描述符、进程 ID 等信息）。操作系统将 PCB 作为任务的唯一标识以及任务调度的基本单位，通过管理 PCB 来实现进程的创建、调度、终止等操作。同时，进程模型借助硬件 MMU 提供的地址空间隔离机制，保证了不同进程的独立性和安全性。

进程模型体现了计算机的软硬协同的发展趋势，它通过硬件的地址空间隔离机制以及在软件上使用状态机模型和 PCB，实现了对计算机硬件资源以及执行流的抽象与封装。通过硬件提供的时钟中断机制，操作系统可以实现进程的时间片轮转等调度算法，处理了多个执行流交替执行过程中的抽象，进一步提高了 CPU 的利用率。例如，UNIX V7 采用多级反馈队列（MLFQ）算法，结合固定时间片轮转与优先级抢占机制，缩短了任务平均周转时间。

2.1.3　线程模型

由于进程既是资源分配的单位（拥有文件描述符、内存映像等），又是调度实体，其创建、销毁以及切换的开销较大，并且硬件提供的地址空间隔离机制，导致 IPC 需要依赖一些复杂机制（例如管道、套接字或共享内存）。这些复杂机制涉及特权级切换、地址空间切换等操作，导致通信开销显著增大，例如，共享内存通信需要缓存一致性协议来保证数据同步，而消息传递模型则需要多次复制数据。这些问题限制了细粒度并发，促使了多线程模型的发展。

线程模型将执行单元与资源所有权解耦，在 TCB 中维护了状态信息（包括程序计数器、通用寄存器等），实现了对执行流的细粒度封装，而进程仍然为系统资

源分配的单位，同进程的线程之间共享进程的资源（如内存和文件句柄等），从而降低通信的开销。线程模型根据其实现方式可以分为内核级线程（1：1 模型，每个用户态线程对应一个内核态线程）、用户级线程（N：1，多个用户态线程对应一个内核态线程）以及两者的混合模型（M：N，将 M 个用户态线程映射到 N 个内核态线程）。内核级线程由操作系统内核直接管理，线程的创建、调度、销毁等操作都由内核完成，因此线程切换的开销较大，但能够充分利用多核处理器的并行性。用户级线程则由用户空间的线程库管理，线程的创建、调度、销毁等操作都由用户空间的线程库完成，因此线程切换的开销较小，但无法利用多核处理器的并行性，单个线程阻塞会导致其他线程无法执行。混合模型则结合了两者的优势，既降低了开销又支持多核并行。除了在软件上的线程模型，硬件上的多核处理器技术（如 SMP 和 AMP）① 以及超线程技术 ② 进一步推动了多线程模型的发展，这些创新共同促成调度粒度从进程级的毫秒量级向线程级的微秒精度迈进。

线程模型减少了不必要的地址空间切换，优化了执行流切换的开销，从而实现了隔离性与并发性的折中。在这一模型中，用户级线程、内核级线程和超线程分别代表了用户库、操作系统和硬件提供的多层次线程支持，它们共同构成了一个软硬协同的线程管理体系，为现代应用程序的并发性和性能需求提供了有力支持。

2.1.4　协程模型

多线程模型作为并发编程的“事实标准”，但由于线程之间需要额外的同步互斥机制（如互斥锁、信号量）来管理共享的进程资源，避免数据竞争，且每个线程需要固定大小的栈来保存执行流的上下文，因此在大规模的并发编程中，多线程模型的不足逐渐体现出来。协程（coroutine）的概念早已经被提出，Marlin 在他的博士论文中总结了协程的核心特征：

- 协程的本地数据在连续调用之间不变；
- 当协程失去控制权时，其暂停执行；在协程重新获取控制权后，协程会从上一次暂停的地方恢复执行；

但是这个通用的定义却没有解决协程结构的相关问题，影响了编程语言中对于协程的支持方式，其中一些实现对协程的表达能力存在误解，此外，一等延续（first-class continuations③）的引入和多线程作为并发编程的“事实标准”，导致协程没有得到广泛使用。直到 Moura 等人证明了完全协程（full coroutines）具有和一次性

① 多个物理处理器之间共享内存子系统以及总线结构等。
② 单个物理核心模拟多个逻辑核心，实现指令级并行与资源复用。
③ continuation 是计算机程序控制状态的抽象表示，它实现了程序控制状态，即 continuation 是一个表示程序执行中给定点的计算过程的数据结构；所创建的数据结构可以被编程语言访问，而不是隐藏在运行时环境中。

续延（one-shot continuations）和一次性限定续延（one-shot delimited continuations）同等的表达能力，协程才逐渐复兴。

协程通过 CPS ①或者状态机保存执行现场，这种无栈式设计（Stackless，论文后续中关于协程的描述均指无栈协程）相较于线程模型，占用的内存资源更少，提高了内存资源的利用率，处理了执行流对程序栈的依赖关系，将执行流上下文切换开销降至纳秒级，并且其与 IO 多路复用、事件驱动等机制非常契合，协程的这些特性展现出在高并发场景下的潜力，现代的运行时甚至能够每 GB 内存承载百万级轻量协程。现代编程语言（如 C++、Go、Rust、Python、Kotlin 等）中的协程基础设施以及操作系统提供的协程运行时从软件上的不同层面提供了不同程度的协程支持。

小结：任务模型的演化历程体现了计算机软硬协同设计的不断进步，这些模型从软件上提供了对硬件资源的不同程度的抽象与封装。进程模型对硬件提供的地址空间和特权级等机制进行了封装，使得任务之间相互隔离；线程模型对隔离性和并发性进行了折中，降低了任务切换的开销，增强了并发能力；而协程模型则通过无栈式设计和状态机模型，具备了更小的内存占用以及更低的任务切换开销，进一步提高了并发性。这些任务模型具备不同的特性和优势，论文将这些模型的特性进行剥离与整合，提出了基于执行流的任务模型。

2.2　任务调度

任务调度机制直接决定了系统的性能表现，目前已经存在着大量的研究工作从多个角度展开研究。

2.2.1　轻量级任务模型

近年来，以协程为任务单元的研究开始引起学术界和工业界的关注，DepFast 在分布式仲裁系统中使用协程；Capriccio 使用相互协作的用户态线程来实现可扩展的大规模 web server；Demikernel 利用了 Rust 无栈协程上下文切换低成本与适合基于状态机的异步事件处理机制的特点，能够在十几个时钟周期内完成任务切换，向应用程序提供异步 I/O；基于 Rust 协程实现的为嵌入式设备上运行的异步驱动生成框架 Embassy 在中断耗时和中断延迟方面远胜于用 C 语言实现的 FreeRTOS 。协程这种轻量级的任务模型，其上下文切换开销小，易于与异步事件处理结合，从操作系统内核以及用户库等不同层面提供对协程的支持，可以实现对硬件缓存的

① 是一种通过显式传递“后续操作”来控制程序执行流程的编程范式。其核心思想是将函数原本隐式返回值的逻辑，替换为将结果传递给一个显式的回调函数（称为 Continuation）。

更好利用，减小调度中的任务切换的开销，提高系统的并发度。

2.2.2　调度算法优化

任务调度的一个研究方向是设计和优化调度算法来满足特定的性能需求。Linux 内核中的 $O(1)$ 调度器设计突破了传统 $O(n)$ 调度器的性能瓶颈，通过双队列轮转机制实现了时间复杂度优化，降低了调度决策的时间复杂度。而 Linux 内核使用的 CFS 调度器为了保证公平性，赋予任务虚拟运行时间（*vruntime*）的概念，保证了在任意调度周期内，所有任务 *vruntime* 的累积增量相等，从而实现公平调度。而葛文博等人提出的众核嵌入式实时调度策略，引入了任务关键度概念，建立了任务最坏响应时间和优先级分配的数学模型，有效地降低了传统调度方式在众核环境下的调度开销。Wierman 等人证明，没有单一的调度策略可以最小化所有可能工作负载的尾部延迟，在任务的尾部延迟较小时，FCFS（first come first served）策略的尾部延迟是渐进最优的；在任务的尾部延迟较大时，则 PS（processor sharing）策略的尾部延迟是渐近最优的。这些策略体现出明显的二分性，即在轻尾工作负载下表现良好的策略在重尾负载下表现较差，反之亦然。Prekas 等人在开展 ZygOS 的研究工作时，对单队列和多队列进行了分析，证明了无论是 FCFS 还是 PS 调度策略，单队列的尾部延迟均优于多队列。Concord 通过构建涵盖抢占式调度、线程同步与通信、任务分发等环节的开销模型，对系统尾部延时的生成路径进行解耦分析，最终在保障吞吐量的同时将尾部延时优化至微秒级。IX、Arachne、Shenango、Caladan、Fred 等工作则围绕着任务调度的负载均衡策略展开研究，充分利用了 CPU，能够有效应对动态负载波动。

2.2.3　通用调度框架

优化调度算法能够有效地提高系统吞吐量和降低尾部延迟，但是在实际应用中，调度策略的选择往往受到应用程序的特性、硬件环境、负载情况等多方面因素的影响，在云服务这种多租户的环境中，针对某个应用来实现定制化的调度策略是不现实的。因此，设计一个通用的调度框架，使得用户可以根据自己的需求定制调度策略，是一个值得研究的方向。ghOSt 设计了一套内核与用户态进程之间传递调度信息的 API，提供了通用的调度策略并托管给用户进程，允许用户在用户态代理中表达复杂策略，从而实现了用户态调度策略的定制化。Syrup 利用 eBPF 机制将用户态提供的网络包调度策略在内核中执行，并借助 ghOSt 框架在用户态定制线程的调度策略，在用户态实现了对整个网络协议栈的定制化，实现了不同层级的网络协议栈信息共享，提高了调度决策的准确性。Skyloft 在内核中增加了内核模块，在其中维护一个多个应用程序共享的运行队列，根据用户定义的策略

进行调度决策，并借助 x86 架构下的用户态中断技术，利用用户态的时钟中断实现了用户态的抢占式调度，提供了一个通用且高效的用户态调度框架。这些工作基于内核态与用户态的协作，通过内核态提供的接口，用户可以在用户态实现自己的调度策略，从而实现了用户态调度策略的定制化。

小结：除了以上三个方向的研究，还存在一些研究使用其他的优化手段，例如保留额外的处理器核心或者使用专用调度核、减少抢占开销、缩小任务时间片等手段提高系统的性能。目前，任务调度方面的已有研究取得了相当不错的成果，但这些工作很少涉及任务调度本身的开销，例如负载均衡机制导致的同步互斥开销，以及调度过程中关中断的操作影响实时性需求等问题。

2.3　中断机制

中断机制作为一种提高计算机工作效率的技术，最初是用于解决处理器忙等外部设备导致的效率低下问题，在硬件与软件之间构建起高效的协作桥梁。当外部设备完成 I/O 请求时，设备向 CPU 发送中断信号，CPU 被迫暂停当前执行的任务，转而优先处理突发 I/O 事件。中断机制打破了传统顺序执行的局限性，使得计算机即能够快速响应键盘输入、网络数据传输等紧急任务，又能通过精密的时钟中断定期重新分配计算资源，从而营造出多任务流畅运行的假象。

然而，中断机制为系统提供实时响应能力以及时分复用机制的同时，也引入了一些开销。每次中断触发时，CPU 必须暂停当前执行的指令流，保存通用寄存器、程序计数器等现场状态，当中断处理完成后再恢复现场，造成直接开销；中断处理程序可能覆盖原任务的指令/数据缓存内容，在原程序恢复执行时，需要重新加载，造成间接开销。现代高速 I/O 设备每秒可以完成数百万个 I/O 请求，若针对每个 I/O 请求产生一次中断则会导致中断风暴，产生不可接受的上下文切换开销。目前，针对中断机制，已经展开了大量的研究工作。

2.3.1　轮询机制

为了避免中断频繁干扰处理器，一些研究和技术选择采用轮询机制。例如，IX 和 DPDK（以及 SPDK ）直接将设备暴露给应用程序，并在用户空间实现轮询，绕过了内核和中断机制。Intel 的 DDIO 和 ARM 的 ACP 允许网络设备直接将传入数据写入处理器缓存，将内存映射的 I/O 查询转换为缓存命中，显著提高了轮询效率。陈旭辉等人提出了一种根据外部事件发生频率分配优先级的方法，并基于这些优先级进行轮询，有效地提高了对外部事件的响应速度。管海兵等人提出的 sEBP（基于事件的智能轮询模型）通过收集各种系统事件优化了网卡的轮询机制。为了

减少轮询的高 CPU 开销，Linux 内核建议使用混合轮询方法，避免轮询整个 I/O 等待时间，I/O 线程会被阻塞一段时间，然后无论 I/O 是否已完成都会被唤醒进行轮询。其关键思想是，没有必要从 I/O 过程的开始就启动轮询，因为 I/O 操作需要一些时间来完成。但是，轮询机制要求 CPU 必须定期主动查询设备状态，即使这些查询没有结果，这将会浪费 CPU 时间，尤其是在设备不太活跃的情况下，并且轮询机制的响应时间受轮询间隔的限制，如果间隔很长，系统的响应可能会延迟，从而影响效率。

除了单独使用中断或轮询机制外，一些研究已尝试将两者结合起来，以保留它们各自的优势。Langendoen 等人结合了轮询、中断和线程管理，利用线程调度器感知任务等待外部事件的能力，当 CPU 空闲时，它采用轮询方式接收网卡数据，并在有可运行线程时切换到中断方式。AlQahtani 等人采用 EDP（使能—禁用中断和轮询）机制，根据系统负载灵活地在轮询和中断策略之间切换。然而，这些混合方法需要依赖启发式的策略，预设的启发式规则难以适应动态变化的负载。

2.3.2　硬件快速上下文切换

大量的研究致力于优化中断上下文切换的开销。例如，RMTP（响应式多线程处理器）芯片已经在硬件中实现了资源分配、上下文切换和中断唤醒机制。其专用的上下文切换指令能够在四个时钟周期内完成一次上下文切换。此外，RMTP 的硬件中断唤醒机制可以在单个时钟周期内切换到响应任务以处理中断。多流水线寄存器架构（Multi Pipeline Register Architecture，MPRA）处理器展示了在几个时钟周期内完成上下文切换的能力，优化了中断处理。然而，这些基于特定硬件平台的优化方法缺乏通用性。

2.3.3　中断合并

尽管前述研究已经优化了单个中断上下文切换的开销，但并未解决由于中断数量过多导致的中断过载问题。因此，许多研究开始探索中断合并技术。中断合并技术减少了 CPU 需要处理的中断数量，以避免中断过载。例如，Salah 等人评估了中断合并技术与传统中断处理的对比。Ahmad 等人提出了一个虚拟中断合并方案，用于在虚拟机监控器中实现虚拟 SCSI 硬件控制器。董耀祖等人进行了高效的中断合并和虚拟接收端扩展，充分利用了多核处理器的优势，用于网络 I/O 虚拟化。市场上许多网卡和存储设备都内置了中断合并功能，但这些功能通常基于静态配置，缺乏动态调整能力。Amy Tai 等人在静态配置的基础上引入了适应性中断合并，使设备能够根据请求的延迟敏感性自适应地合并中断。然而，中断合并机制与混合中断与轮询机制存在着相同的问题。

2.3.4　高级中断控制器

一些研究尝试在中断控制器中提供超越传统中断控制器的高级功能。RISC-V 架构下的 AIA、ARM 架构下的 GICv3/v4 以及 x86 架构下的 APIC 中断控制器都提供了对 MSI 机制的支持，设备写入特定的地址即可发出中断，支持将中断直接注入虚拟机，减小虚拟化场景下的中断转发开销。此外，一些中断控制器支持向用户态发送中断，硬件直接传递中断信号到用户空间，并允许用户态程序注册自定义的中断处理逻辑，从而消除传统的由内核处理中断机制导致的上下文切换开销（特权级切换开销，在开启 KPTI ①后，还包括地址空间切换开销）。目前已经存在多项工作围绕着用户态中断技术展开，Skyloft、uProcess 使用了 Intel 提出的 UINTR 扩展来实现用户态的抢占式调度。此外，RISC-V 指令集的 N 扩展也实现了对用户态中断技术的支持，Sandro 等人已经将其用于嵌入式系统，用于构建可信执行环境。

2.3.5　软硬协同

一些研究还在探索如何通过软硬协同来提升系统的性能。例如，G.R. Gao 等人提出的轮询 Watchdog 硬件扩展，只有在显式轮询无法及时处理到来的消息时才会触发中断，从而减少了不必要的中断。在 Gomes 等人的研究工作中，中断控制器可以识别当前运行线程的优先级，并确保低优先级中断不会抢占高优先级线程的执行，有效地避免了由中断引起的优先级反转问题。Erwin 等人使用的队列内容寻址存储器（CAM）技术提供了一种高效组织和管理中断的新方法。Fabian Scheler 等人使用外围控制处理器（PCP）协处理器可以在中断发生时直接更新内存中的任务状态，只有当高优先级中断任务准备就绪时才通知 CPU 进行重新调度，然而，这种方法也有一定的局限性，因为 CPU 和 PCP 同时访问内存可能会导致较高的同步互斥开销，有时会增加中断处理的延迟。这些方法的共同之处在于它们都利用了内核的任务调度器，通过增加与任务控制相关的属性（如任务优先级、状态和时间尺度）来增强它。然而，由于内核和硬件都访问内存中的任务控制块，因此需要额外的同步互斥开销。

小结：这些研究从软件或硬件上尝试优化中断机制对 CPU 的影响，即使对中断控制器增加了新功能，但没有改变中断机制的本质。在中断产生时，CPU 需要从当前运行的任务切换到中断处理程序，但不会修改当前正在运行的任务标识（`current_task`）。因此，中断处理例程占用了当前任务的时间片。即使 Linux

① KPTI 是 Linux 内核的一项功能，它将用户空间和内核空间页表完全分离来强化内核安全性。

内核将中断处理划分为上半段[①]和下半段[②]以减轻中断处理对当前任务的影响，但仍然会打断当前任务，在当前任务的开销中增加中断导致的开销。

2.4　I/O 模型

I/O 模型描述了应用程序与外部设备之间通信的过程，应用程序通过系统调用来执行 I/O 操作，内核来执行具体的 I/O 操作。I/O 模型设计面临的核心问题是任务需要等待 I/O 完成，CPU 与外部设备速度不匹配，不同的模型通过不同的方式协调这一矛盾：（1）同步与异步：同步模型要求应用程序主动轮询或等待结果，而异步模型则由内核主动通知应用程序；（2）阻塞与非阻塞：阻塞模型会挂起线程直到操作完成，而非阻塞模型则允许线程立即返回并执行其他任务。

阻塞式 I/O 因其实现简单且资源消耗低更加适用于简单低负载应用（如嵌入式设备），而实时系统或响应式界面更适合非阻塞 I/O 以避免阻塞延迟；由于同步阻塞 I/O 模型效率低下，应用程序中其他处于就绪状态的任务会因为执行阻塞操作的任务而一并等待 CPU，而异步非阻塞 I/O 模型由于其高效的 I/O 处理方式，充分利用了任务等待 I/O 的时间提高了系统的并发度，因此适用于高并发的场景。PM-AIO 针对持久化内存文件系统中使用伪异步（同步）I/O 路径处理异步 I/O 请求导致性能低下的问题，使用内核线程实现真正的异步 I/O 路径，提高了持久化文件系统的性能；Nenov 等人使用基于异步非阻塞的 I/O 模型，解决了大量并发连接时容易出现的资源耗尽和性能下降的 C10K 问题，构建出了高性能的 Web 服务器；Leonard 等人利用异步 I/O 和协程来克服 SSD 的高延迟和低带宽问题，通过隐藏 I/O 延迟实现最大化 I/O 并行性，提高了基于 SSD 的数据库系统的读带宽，降低了经济成本。

小结：I/O 模型从同步阻塞向异步非阻塞 I/O 模型发展，提高了系统的并发度。但这些模型对任务调度存在不同程度的影响：（1）在同步阻塞 I/O 模型中，用户态的某个任务通过系统调用切换到内核态任务，若内核态任务阻塞，其他用户态任务无法继续运行；（2）用户态任务将 I/O 设置为非阻塞模式，内核态任务通过返回 `EWOULDBLOCK` 通知其他用户态任务继续执行，并定期主动轮询 I/O 进展，增加了额外的轮询开销；（3）在 I/O 多路复用模型中，内核态任务需要额外的机制（文件描述符集合的管理、事件循环等）来维护用户态任务与 I/O 的状态信息，在 I/O 就绪后需要通过分发机制来通知具体的用户态任务；（4）基于信号或基于线程的回调函数构建的异步 I/O 模型会增加特权级切换的开销。

① 上半段为一些必须紧急处理的事项，例如将设备的中断寄存器清空，完成必要的数据拷贝操作等。
② 下半段作为一个单独的任务，例如网络协议处理等。

2.5　本章小结

本章介绍了与任务调度、中断响应机制以及 I/O 模型的相关研究，这些研究的发展和应用作为计算机系统软硬协同发展的结果，为论文的研究工作提供了理论基础和技术支持。然而，这些研究仍存在着一定的不足之处，任务调度机制仍存在优化空间，任务调度机制和中断机制之间的矛盾需要得到缓解。论文将在后续的章节对这些问题展开深入研究。

第 3 章　基于软硬协同的任务调度和中断响应机制设计

本章将介绍基于软硬协同的任务调度和中断响应机制的设计原则，以及组成该机制的基于执行流的任务模型、基于软硬协同的任务状态维护方法、硬件处理中断机制和软硬件交互接口的设计。

3.1　设计原则

系统中进程、线程和协程多种任务并存，处于不同层次的调度器针对各个类型的任务控制块进行管理。然而，由于特权级机制，操作系统的调度器以内核态的任务作为调度实体，而不关注用户态任务的边界，将用户态的若干任务视为一个整体捆绑在一个内核态任务上。因此，内核态任务包括了用户程序在完成某项功能的生命周期内的所有执行流的变化，不仅包括在用户态运行的由应用程序定义的执行流，还包括了运行于内核态中用于提供系统服务、中断处理以及异常处理的执行流。这种任务调度实体模糊了执行流之间的边界的本质，导致了任务调度机制与中断机制的矛盾，以及在 I/O 模型中出现的相关问题（详见第 2.3 节和第 2.4 节的小结部分）。为了应对这些问题，准确地描述操作系统中所维护的执行流在整个生命周期中的状态以及状态变迁，基于软硬协同的任务调度和中断响应机制设计应该遵循以下原则：

（1）严格划分执行流之间的边界，把执行流抽象成任务，建立基于执行流的任务模型。执行流在 CPU 上执行时的硬件寄存器状态（如地址空间、特权级、堆栈指针等）以及占用的资源（如 I/O 设备、内存页和 CPU 时间片等），这些元素共同构成了任务的完整执行环境，任务一定处于某个特定的执行环境中。在这种界限严格的定义下，中断处理例程、异常处理例程和实现普通功能的函数都是各自独立的任务。

（2）将调度实体的粒度对齐到基于执行流的任务模型上。一旦执行环境发生变化，例如地址空间、特权级、堆栈或占用资源变化，执行流也将发生变化，这意味着发生了任务切换。因此，由于中断、异常和系统调用导致的从用户态跳转至内核态的行为都将被视为任务切换。调度器需要能够感知任务的执行环境变化，即感知任务，从而将调度实体的粒度对齐到基于执行流的任务上，合理地安排任务的执行顺序。

（3）合理划分软件与硬件的边界和协作关系，简化任务之间的交互关系。应用程序由若干个在用户态的普通任务与内核的中断处理任务、异常处理任务和系统

调用任务共同组成，这些任务之间相互传递信息，并基于通知机制或其他同步互斥机制相互协作，共同完成应用程序需要的功能。

本章将围绕着上述三项原则展开设计：（1）第 3.2 节将按照原则一展开基于执行流的任务模型设计，使得任务之间的边界更加清晰，并对系统中存在的任务切换进行分类汇总，将系统的整体开销划分为任务本身的开销、任务切换的开销和任务之间交互的开销，从而帮助分析系统中的性能瓶颈，找到优化的方向。（2）第 3.3 节将根据原则二展开对硬件化的任务调度器的设计，使得调度器需要能够感知到任意任务。目前已经有研究围绕着让操作系统内核的调度器能够感知到用户态任务的优先级、状态信息，论文将在此基础上更进一步，让硬件感知到任务的状态信息，从而实现硬件化的任务调度器，减小软件实现的任务调度器的开销。（3）第 3.4 节根据原则三展开对硬件响应中断机制的设计，使用硬件来完成中断处理这个特殊任务，简化中断这种任务交互机制，并在此基础上，提供任务之间的基于硬件的通知机制，减小任务之间的通知机制的开销。最后，软件还需要一套与硬件的交互接口（第 3.5 节），来使用硬件提供的功能，这些机制共同构成完整的软硬协同的任务调度和中断响应机制，从而使得系统的性能得到提升。

3.2　基于执行流的任务模型设计

3.2.1　任务建模

基于执行流的任务模型建立在已有的进程、线程、协程概念的基础上，进程的概念用于区分不同的地址空间，以及作为资源分配的实体，而不再对线程与协程进行严格的区分，两者都是对执行流的表示，其区别在于线程将执行流的函数调用关系等上下文信息保存在栈中，而协程则将执行流的上下文转换成有限状态机保存在堆上。因此，基于执行流的任务模型根据执行流所处的执行环境（包括地址空间、特权级、堆栈和占用资源等），使用 $T(P_i, L_j, S_k)$ 来表示某个具体的任务实例，其中：

- P：表示执行流所处的地址空间以及占用的资源，即进程。P_i 可以是内核，也可以是某个进程；
- L：表示执行流运行的特权级，L_j 可以是用户态、内核态或其他特权级；
- S：表示执行流的函数调用状态，S_k 可以是线程或协程的函数调用状态。当执行流暂停时，线程的函数调用状态被保存在栈上，而协程的函数调用状态被保存在堆中。无论线程还是协程，它们在运行中都需要运行栈来记录函数调用状态，区别在于多个协程可以复用同一个运行栈。为了提高对内存的利用率，执行流通常以协程的形式存在。

3.2.2　任务切换

基于执行流的任务模型使得任务之间的边界更加清晰，应用程序的逻辑由 N 个用户态任务与 M 个内核态任务共同组成，任务之间的切换可以使用三元组（$[prev] \rightarrow [next] : \{condition\}$）进行描述，其中 *prev*、*next* 分别表示切换前后的任务，*condition* 表示切换的条件，任务切换可以从多种维度进行划分。根据触发方式可以任务切换划分为以下两类：

（1）主动切换：这种情况包括了任务在执行过程中由于等待资源或申请系统服务（发起系统调用）而进行的切换，以及任务执行结束后进行的切换。任务在进行这类切换时，会主动保存当前的函数调用状态到堆中，因此切换前的任务 *prev* 为协程，而切换后的任务 *next* 则可能为线程或者协程。

（2）被动切换：这种情况包括了任务在执行过程中，CPU 收到中断或者任务执行产生异常而导致的切换。这类切换可能发生在任务执行的任意时刻，因此任务的函数调用状态直接被保存在运行栈上，切换前的任务 *prev* 为线程（当 *prev* 任务下一次被调度时，它将从栈上恢复函数调用状态，并重新转化为协程），对应的中断或异常处理任务 *next* 为协程。由于产生了中断或异常，*next* 任务将从阻塞状态转变为就绪状态。

发生切换时，前后两个任务的函数调用状态 S_k 必定发生变化，而地址空间和特权级则不一定变化。因此，根据切换时的执行环境变化，任务切换又可以划分为以下四类：

（1）相同地址空间内不跨越特权级的切换：这种切换包括了在同一进程内，在用户态或者内核态中的任务主动让权引起的任务切换。任务主动让权，只保存必要的函数调用状态，因此开销较小。此外，收到由同特权级下进行处理的中断（包括了用户态中断以及运行在内核态时产生中断）或异常信号导致的被动任务切换也属于此类切换。任务被动进行切换，任务正在运行的所有函数调用状态被保存在运行栈上，此运行栈被占用但被中断或异常处理任务直接复用，因此，这种任务切换的开销比任务主动让权引起的切换开销略大。中断或异常处理任务相关的函数调用状态不保存在栈上，因此中断或异常处理任务为协程。无论内核是否使能了 KPTI 机制，当内核态任务执行过程中出现了中断或异常时，此时的地址空间中均包含了内核的地址映射，无需切换地址空间和特权级。

$$T(P_i, L_j, S_k) \rightarrow T(P_i, L_j, S_l) : \{\text{任务主动让权} \mid \text{同特权级处理的中断或异常}\}$$

（2）相同地址空间内跨越特权级的切换：在没有使能 KPTI 机制的情况下，进程地址空间中包含了内核的地址映射，当用户态任务要获取内核提供的系统服务时，用户态任务会主动执行系统调用将自己的函数调用状态保存在堆上，并通过

相应的指令将系统调用的参数发送给内核中的系统调用处理任务，当内核态的系统调用处理任务结束后，将结果返回给用户态任务。这属于用户态的协程与内核态的协程之间的相互切换。而当用户态任务正在执行时，收到了需要内核处理的中断信号或者出现了异常时，用户态的任务无法主动让权，被动地将当前的函数调用状态保存在运行栈上，并切换到内核的中断或者异常处理任务，当内核的任务处理完成后，返回到用户态，被打断的任务将从运行栈上恢复函数调用状态并继续执行，这属于用户态的线程与内核态的协程之间的相互切换。这种切换的开销包括特权级切换的开销以及堆栈切换的开销。

$$T(P_i, L_j, S_k) \rightarrow T(P_i, L_l, S_n)：\{\text{禁用 KPTI 时的系统调用 | 中断 | 异常}\}$$

（3）跨越地址空间但不跨越特权级的切换：这种切换通常发生在内核态中，因为需要在高特权级状态下才可以对 CPU 的页表寄存器进行修改。内核态执行的前一个任务与被调度的下一个任务属于不同的进程，因此切换不需要跨越特权级但需要跨越地址空间。此外，一些特殊的硬件提供了用户态的隔离机制，例如 uProcess 使用了 Intel 的 MPK 硬件，在用户态构建出不同的保护域，实现了不跨越特权级而跨越不同保护域的用户态任务切换，实现了原本需要进行两次跨特权级切换的组合的相同效果，大幅度节约了切换的开销。但这些保护域事实上仍处于同一个地址空间中，因此，类似于 uProcess 实现的用户态切换不属于此类任务切换，仍属于同地址空间不跨越特权级的切换。

$$T(P_i, L_j, S_k) \rightarrow T(P_l, L_j, S_n)：\{\text{内核态任务主动让权}\}$$

（4）跨越地址空间且跨越特权级的切换：在未使能 KPTI 机制时，因为跨越地址空间，需要先从相同地址空间中的低特权级态切换到高特权级态，才可以切换地址空间，因此这种情况属于上述的第二类切换与第三类切换的组合。当使能 KPTI 机制时，用户态的任务必须切换到内核地址空间的系统调用处理任务，才可以获取内核提供的系统服务。同理，用户态任务运行过程中收到需要内核处理的中断和产生异常时，也需要经过这种切换。这种切换既包括了特权级切换也包括了地址空间切换，因此开销较大。

$$T(P_i, L_j, S_k) \rightarrow T(P_l, L_m, S_n)：\{\text{使能 KPTI 时的系统调用 | 中断 | 异常}\}$$

3.2.3　任务交互

在基于执行流的任务模型下，用户态的执行流与内核态的执行流分别属于不同的任务，两个不同的任务作为发送方和接收方通过通知机制进行交互协作。在这种视角下，系统调用、中断和异常处理机制可以被解释为：

- 发起系统调用：用户态任务通过陷入指令主动通知内核态的系统调用处理任务；
- 系统调用返回：系统调用处理任务通过指令通知用户态任务；
- 产生中断：某个任务将自己阻塞在某个外部事件的等待队列中，在某一时刻通过中断信号通知对应的中断处理任务；
- 中断返回：中断处理任务通过指令通知被中断信号打断的任务；
- 产生异常：硬件错误通过异常信号通知对应的异常处理任务；
- 异常返回：异常处理任务通过指令通知被异常打断的任务。

表 3.1　任务切换汇总

执行环境	触发方式	切换条件	任务变化	是否存在通知
P[1], L[2] 均相同	主动	用户态/内核态任务主动让权	协程 → 协程/线程	
		同 L 中断处理后的任务抢占[3]		✓
		同 L 中断/异常处理后返回[4]	协程 → 线程	✓
	被动	产生同 L 处理的中断/异常	线程 → 协程	✓
P 相同 L 不同	主动	发起系统调用 系统调用返回	协程 → 协程	✓
		高 L 中断处理后的任务抢占	协程 → 协程/线程	✓
		高 L 中断/异常处理后返回	协程 → 线程	✓
	被动	产生高 L 处理的中断/异常	线程 → 协程	✓
P 不同 L 相同	主动	内核态任务主动让权	协程 → 协程/线程	
P, L 均不同 使能 KPTI	主动	发起系统调用 系统调用返回	协程 → 协程	✓
		高 L 中断处理后的任务抢占	协程 → 协程/线程	✓
		高 L 中断/异常处理后返回	协程 → 线程	✓
	被动	产生高 L 处理的中断/异常	线程 → 协程	✓

[1] P 指地址空间。
[2] L 指特权级。
[3] “中断处理后的任务抢占”指，中断处理任务将某个处于阻塞状态的任务唤醒为就绪状态，被唤醒任务的优先级高，中断处理任务主动切换至被唤醒的任务（可能为协程或线程），而不是返回至原本被打断的任务。
[4] “中断/异常处理返回”指从中断处理任务切换回被中断信号打断的任务。

由于中断处理而导致的任务抢占则可被解释为，中断处理任务根据中断信号修改了某个任务的状态，由于被唤醒任务的优先级较高，中断处理任务主动让出 CPU，通知该任务直接运行。进程、线程、协程模型中已有的同步互斥等其他任务

交互机制在基于执行流的任务模型中仍然适用，这里不展开深入描述。

表 3.1 根据任务切换的执行环境、触发方式、切换条件、切换前后两个任务的类型以及任务切换是否存在通知，对任务切换进行了汇总。表格中的分类进一步指出：**当产生通知（发生系统调用、中断、异常）时，接收通知的任务必定处于阻塞的状态，需要修改其状态，且必定会导致任务切换；反之，任务切换不一定表示存在通知。**这一结论将指导本章后续的设计（3.4 节）以及在软件适配中的异步改造过程（4.4 节）。

3.3　硬件化的任务调度器设计

任务调度器需要根据任务的状态以及调度策略来决定任务所处的队列，并维持队列的偏序关系。通常，任务的状态被保存在任务控制块中的某个字段中，任务调度器对任务状态字段所在的位置进行读写操作实现对任务状态的管理，并根据任务的状态将任务插入到不同的任务队列中。这一系列行为要求任务调度器获取任务控制块的位置、状态字段在任务控制块的偏移、任务的优先级以及任务队列的位置等信息。为了实现硬件化的任务调度器，将任务调度从软件卸载到硬件中，硬件需要依赖以下两项机制来感知任务的状态、优先级等信息，与软件协作，共同维护任务状态以及状态之间的变迁：（1）分层复合型任务标识；（2）软硬协同的任务状态维护方法。

3.3.1　分层复合型任务标识

在 Linux 等系统中，任务 ID 仅仅为数字编号，用于唯一标识任务，软件需要通过任务 ID 获取到任务控制块，进而获取任务的状态以及优先级等调度属性，实现任务管理。但这种 ID 无法表达出任务的地址空间、特权级和调度属性等信息。

论文加强了任务标识的作用，将任务的数字编号、优先级以及其他调度属性（例如 CPU 亲和性掩码）等信息组合成复合型任务标识 $T(W_i)$。软件向硬件传递这个标识，硬件即可感知任务的详细信息，从而能够作出调度决策。更进一步，论文将该任务标识与任务所属的操作系统的标识 $O(W_i)$[①]和进程的标识 $P(W_j)$[②]组合，形成了分层复合型任务标识 $(O(W_i),\ P(W_j),\ T(W_k))$，如图所示。硬件根据分层复合型任务标识中的操作系统标识和进程标识来感知任务所属的地址空间[③]和特权

① 论文使用操作系统内核在初始化阶段时的默认任务的标识来表示操作系统标识。在没有虚拟化的场景下，计算机系统中只存在一个操作系统，操作系统标识没有作用；而在虚拟化场景中，计算机系统中存在多个操作系统，因此这个标识可以用于区分不同的操作系统。

② 与操作系统标识类似，论文采用在初始化进程时，创建的进程的默认任务的标识来表示进程标识。

③ 标识中有限的字段长度不能完全表示出无限数量的地址空间。因此，这里的“感知”不意味着硬件可以直接感知页表信息，而是通过切换前后两个任务的标识变化来判断是否存在地址空间切换。

级[①]等执行环境信息，并根据任务标识中的优先级等调度属性信息作出更加准确的调度决策。

3.3.2　软硬协同的任务状态维护方法

实现硬件化的任务调度器还需要对传统的任务状态维护方法进行扩展，使得硬件能够维护任务的状态以及状态变迁，论文设计了基于软硬协同的任务状态维护方法，如图所示。

任务状态仍然为创建、就绪、运行、阻塞和退出这五种状态，其中灰色框线中的就绪态、运行态与阻塞态以及它们之间的状态转移由硬件进行。硬件在内部为每个 CPU 维护当前正在运行的任务标识 $(O(W_i),\ P(W_j),\ T(W_k))$，从而感知运行态的任务；硬件在内部维护就绪任务队列 RQ 和阻塞任务队列 BQ，根据任务标识 $(O(W_i),\ P(W_j),\ T(W_k))$ 中提供的信息将任务标识添加至就绪队列或阻塞队列中，感知处于就绪态和阻塞态的任务，并维护这些队列的偏序关系，实现调度器的功能。硬件通过将任务标识在这些队列中迁移实现对任务状态变迁的管理。在硬件中维护处于就绪态、阻塞态和运行态的任务信息可以得到以下好处：

（1）将调度实体的粒度与基于执行流的任务模型对齐。就绪队列中可以存放属于不同地址空间、不同特权级（不同操作系统、进程）下的任务标识，当处于内核态或者用户态的运行时从硬件化的任务调度器中取出一个任务时，这个调度决策是在硬件调度器感知到所有任务的信息的基础上进行的，因此，调度实体的粒度与基于执行流的任务模型是对齐的。

（2）高效地进行负载均衡和任务状态变迁。在硬件中维护就绪队列，可以直接在硬件层面实现负载均衡机制，减小由软件实现负载均衡导致的同步互斥开销。此外，软件运行时还可以利用一些硬件机制（例如总线事务）来实现对任务队列的互斥访问。硬件通过维护任务队列而感知任务状态，可快速的进行状态变迁，减小了软件修改任务状态过程中的由于读取任务控制块中的状态字段以及操作任务队列等行为带来的开销。

（3）支持抢占式调度。硬件感知处于运行态的任务以及处于就绪态的任务，可以直接判断就绪任务的优先级与当前任务的优先级，从而实现抢占式调度。

硬件无法感知灰色框线外的其他任务状态，这些其他任务状态以及它们之间的变迁由软件来维护，其他任务状态与就绪态、运行态、阻塞态之间的状态变迁也由软件通过读写硬件的相应端口来发起。任务的状态变迁如下：

- 创建 → 就绪：软件写硬件端口将新创建任务的标识添加至就绪队列中，例如在软件执行 `spawn` 等创建任务相关的函数时，将发生这种状态变迁。

① 特权级数量有限，可以通过操作系统标识和进程标识中未使用的字段直接表示。

- 创建 → 阻塞：软件写硬件端口将新创建任务的标识添加至阻塞队列中，这种状态变迁通常发生在初始化中断子系统，注册中断处理任务时。
- 就绪 → 运行：在使用 schedule 函数进行任务调度时，软件读硬件端口，获取到就绪队列中最高优先级的任务标识，将其标注为正在运行。
- 运行 → 就绪：软件写硬件端口将正在运行的任务标识添加至就绪队列中，例如任务主动让权执行 yield 函数。
- 运行 → 阻塞：软件写硬件端口将正在运行的任务标识添加至阻塞队列中，例如任务因为等待某个事件而执行 task_block 函数。
- 运行 → 退出：软件将正在运行的任务标识放入软件中的退出队列中，这通常发生在任务运行结束执行 exit 函数时。
- 阻塞 → 就绪：硬件调度器根据特定的条件（例如收到中断信号）将阻塞队列中的任务标识添加至就绪队列中。
- 阻塞 → 退出：软件向硬件端口写任务标识，硬件调度器将任务标识从阻塞队列中移除，软件将任务标识放入软件中的退出队列中。例如，当需要替换中断处理任务时，将会出现这种状态变迁。

3.4　硬件处理中断机制设计

3.4.1　中断处理机制

在基于执行流的任务模型下，中断处理任务的功能是根据收到的中断信号，在阻塞队列中找到等待中断信号的任务，修改其状态并放入就绪队列中，完成唤醒操作。中断处理的过程从表象上看是中断处理任务与硬件设备之间的交互，但其本质仍然是中断处理任务与其他任务之间的通知。例如，当系统调用任务调用 sleep() 函数时，它告知中断处理任务，某个用户态任务需要在一段时间后被唤醒，中断处理任务在接收到时钟中断后，唤醒这个用户态任务。这一套任务之间通过通知相互协作的流程构成了内核中提供的 sleep() 系统服务，同理其他的系统服务也可以通过这种通知机制来进行解释。中断处理任务需要获取中断信号、任务的状态、优先级等信息，以及任务队列的位置。因此，在实现了硬件化的任务调度器后，中断处理任务可以被卸载到硬件中。

使用硬件来实现中断处理任务，其本质是构建一套硬件电路，在收到中断信号之后，找到硬件中维护的该中断对应的阻塞队列，找出阻塞在这个中断上的任务的标识，并将任务标识放入就绪队列中，完成对中断对应的事件处理。这个事件处理的过程由软件发起，软件和硬件共同协作完成，其流程如下：

1. 软件向硬件端口写任务标识，硬件将任务标识放入与中断相关的阻塞队列，

等待中断信号产生。这个过程由软件发起，预先将任务标识与中断进行绑定，维护了任务与等待的中断之间的关系。

2. 当硬件收到中断信号后，根据中断信号，找到对应的阻塞队列并从中找到等待该中断的任务标识，根据任务标识中的优先级信息，将任务标识从阻塞队列中移除，放入就绪队列中的合适位置。如果唤醒的任务的优先级比当前正在 CPU 上运行的任务的优先级高，那么硬件还应该向 CPU 发送中断（CPU 在收到这个中断信号后，需要进行任务切换，执行被唤醒的任务），从而完成硬件处理中断的过程。

通过硬件快速处理中断，唤醒阻塞的任务，CPU 将不被一些中断信号打断，从而减少由于中断导致的上下文切换开销。若被唤醒的任务优先级较高，需要进行任务切换，在这种情况下，仍然可以减少由于中断对应的事件所引起的调度开销，硬件中断处理机制与传统机制的对比如图所示。

3.4.2　基于硬件中断处理的任务通知机制

在硬件处理中断的基础上，硬件可以进一步向软件提供任务通知机制，从而使得某个任务可以直接向其他处于不同地址空间、不同特权级的任务发送通知，并配合其他的机制（例如共享内存等机制）实现快速的 IPC。任务通知涉及的两个任务分别为发送方与接收方，它们属于不同的地址空间或不同特权级下的操作系统内核或进程，在上述的硬件支持的中断处理机制的基础上，将中断信号的发送方从硬件设备扩展为软件上的发送方任务。硬件中维护与双方通知相关的接收与发送能力表，并配置额外的能力检查机制来避免恶意通知。发送方和接收方通过写硬件端口来注册相关的能力以及发送通知（内核态任务可直接注册能力，用户态任务在注册时需要内核参与，但注册成功后可直接发送通知）。以两个用户态的收发任务为例，其完整的任务通知流程如下：

1. 用户态接收方任务发起系统调用，内核的系统调用处理任务分配一个通道号，并向硬件端口中写入用户态任务的操作系统标识、进程标识、任务标识和通道号，硬件将这些信息保存至接收能力表中，从而在硬件中保留一条通道。系统调用处理任务向硬件注册能力后切换至用户态接收方任务，完成注册接收方的流程；
2. 用户态发送方任务注册的流程如接收方任务注册的流程类似，通过系统调用由内核来完成，但内核的系统调用处理任务向硬件注册能力后，将通道对应的端口映射到地址空间中，并返回一个描述符，完成注册发送方的流程。当发送方和接收方均注册完成后，硬件才会建立一条完整的通道；
3. 发送方任务向描述符中写入通道号，即可尝试发起通知。硬件首先检查发送

能力表中是否存在通道号对应的发送方操作系统标识以及进程标识。如果存在，则硬件根据通道号找到接收能力表中对应的接收方的操作系统标识、进程标识和任务标识；如果不存在，则发起通知失败；

4. 如果接收能力表中存在通道号对应的接收方的操作系统标识、进程标识以及任务标识，则硬件将接收方的任务标识放入就绪队列中的适当位置。如果不存在，则这次通知失败；
5. 如果通知的接收方任务的优先级比正在 CPU 上运行的任务的优先级高，那么硬件还应该向 CPU 发送中断，CPU 在收到这个中断信号后，使用硬件的任务出队接口（在用户态运行时初始化时已经添加了地址映射）来取出接收方任务标识，执行接收方任务。

基于硬件中断处理的任务通知机制，跨域不同地址空间和特权级的任务之间可以实现快速交互，减少了切换地址空间以及特权级的开销，图展示了使用信号机制与使用硬件进行任务通知的对比。

3.5　软硬件交互接口设计

I/O 端口不仅是软件访问硬件的媒介，也是硬件响应软件操作并返回结果的媒介。软件通过向 I/O 端口发送读写命令，将数据与控制信号传递给硬件，最终转化为对硬件中的寄存器或状态的操作，从而驱动硬件完成特定的功能。在基于软硬协同的任务调度和中断响应机制中，软硬件相互协作，共同完成任务调度、中断处理以及任务通知等功能，软硬件之间的交互接口如表 3.2 所示。软件中的运行时通过这些接口来使用硬件提供的功能：

- 运行时通过 `task_enqueue` 接口将任务标识放入硬件的就绪队列中，通过 `task_dequeue` 接口从硬件的就绪队列中取出下一个需要运行的任务。
- 运行时通过 `register` 接口来维护任务与事件之间的关系，将某个任务注册为外部中断事件的接收方或者与任务通知的接收方或发送方，并通过 `cancel` 接口取消任务与事件之间的关系；
- 任务通知过程中的发送方使用 `send` 接口向接收方发起通知。

软件运行时使用任务调度相关的接口来减小调度过程中的同步互斥开销。此外，当某个任务需要等待设备 I/O 时，这个任务将自己注册为 I/O 事件的接收方，当事件发生时，硬件将这个任务唤醒，放入就绪队列中，从而减小了原本在中断处理任务中唤醒这个等待任务的开销，可以构建出高效的异步设备驱动。软件可以使用基于硬件中断处理的任务通知机制来唤醒接收方任务，减小任务通知的开销，但是基于表 3.1 得出的结论（一旦发生任务通知，接收方任务一定不处于运行

表 3.2　软硬件交互接口

功能	接口	说明
任务调度	task_enqueue	任务入队
	task_dequeue	任务出队
事件通知	register	注册与事件相关的能力
	cancel	取消与事件相关的能力
	send	发送方发起通知

状态，这意味着发生任务切换），这种方式无法节约掉任务切换的开销。在此基础上，如果让原本需要通知来唤醒的接收方并行地运行在其他 CPU 上，那么可以充分利用多核 CPU 来提高系统调用、IPC 的性能，构建出高效的异步系统调用、IPC 机制。

3.6　本章小结

本章基于任务模型遇到的挑战，提出了基于软硬协同的任务调度和中断响应机制的三项设计原则，并且围绕着这些原则展开了具体的设计。基于执行流的任务模型将多种任务模型统一起来，对任务切换的类型和过程进行了抽象；基于软硬协同的任务状态维护方法实现了高效的任务状态变迁和负载均衡机制，减小了任务调度的开销；硬件处理中断机制减小了中断机制对 CPU 的影响，缓解了任务调度机制与中断机制的矛盾，基于硬件中断处理机制而设计的任务通知机制提供了任务之间的快速通知通道，避免了大量的地址空间、特权级切换开销；软硬件交互接口使得软件可以方便地使用硬件提供的任务调度和中断处理功能，从而实现异步驱动、异步系统调用和异步 IPC 机制。这些设计共同构成了基于软硬协同的任务调度和中断响应机制，从而达到提高系统性能的目标。

第 4 章　基于软硬协同的任务调度和中断响应机制实现

本章根据第 3 章所述的设计方案，在 FPGA 平台上实现了具备任务调度、中断处理和任务通知的控制器（TAIC，Task Aware Interrupt Controller），进行了相关的软件适配，实现了基于软硬协同的任务调度和中断响应机制的系统原型。

4.1　系统整体结构

图给出了基于软硬协同的任务调度和中断响应系统的整体架构。系统由运行在 CPU 上的软件、TAIC 和外部设备三部分组成。其中，TAIC 由就绪队列、发送能力表、接收能力表和中断处理模块组成，就绪队列根据任务标识中与调度相关的属性维护任务标识的顺序，且直接支持负载均衡功能；发送能力表中记录了可以发送任务通知的发送方标识；而接收能力表中则记录了可以接收中断事件和任务通知的接收方任务标识。外部设备的中断信号连接到 TAIC 的中断处理模块中，软件通过总线与 TAIC 交互，通过软硬件交互接口使用 TAIC 的调度功能以及注册与中断事件和任务通知相关的能力，中断处理模块在收到中断或者任务通知信号时，会根据中断号或者通道号来检查对应的发送方能力表或者接收能力表，从而完成中断处理和任务通知。CPU 上的软件通过在所有地址空间和特权级中共享的一段跳板代码实现前后跨地址空间、特权级的任务切换。

4.2　任务调度

TAIC 内部维护了与 CPU 数量相等的任务队列，每个任务队列与 CPU 进行绑定（局部队列）。局部队列中存放任务标识，不同的局部队列之间能够窃取任务标识，这些局部任务队列共同组成了就绪队列。就绪队列直接在硬件层面提供了软件中的任务队列需要的出队、入队、优先级排序和负载均衡等功能，满足软件的任务调度需求。运行在某个 CPU 上的软件通过交互接口来操作与该 CPU 绑定的局部队列，实现对该 CPU 时间的分配，并且保证 CPU 的缓存友好性。每个局部队列由一块脉冲阵列组成，实现优先级排序，如图所示。

脉冲阵列用于存放任务标识 ($O(W_i)$, $P(W_j)$, $T(W_k)$)，其中 $T(W_k)$ 包含了任务的优先级等与调度相关的信息。脉冲阵列具备以下属性：

- 在每个时钟周期，脉冲阵列会根据每个任务的优先级来对任务标识进行排序，以此来实现优先级排序；

- 当删除某个位置的任务标识时，该位置后的所有任务标识自动前移；
- 当在某个位置插入任务标识时，先将该位置以及后续的所有任务标识后移，然后再将任务标识插入到该位置。

基于脉冲阵列的属性，只需要使用一个 *head* 指针和一个 *tail* 指针，表示局部队列的头部和尾部在脉冲阵列中的位置，用于实现出队和入队操作，此外还维护了一个 *count* 变量，表示局部队列中的任务标识的数量。局部队列支持的操作为：

- 出队：取出脉冲阵列中 *head* 处的任务标识，脉冲阵列后续元素自动前移，将 *tail* 指针 -1，将 *count* -1；
- 入队：由于脉冲阵列排序需要若干个时钟周期，因此脉冲阵列插入元素 *new_item* 时，先将 *new_item* 的优先级信息与 *head* 处的任务标识 *head_item* 的优先级信息进行比较，如果 *new_item* 的优先级更高，则直接将 *new_item* 插入到 *head* 处，将 *head_item* 取出，入队操作转化为插入 *head_item*；如果 *new_item* 的优先级不高于 *head_item*，则仍然为插入 *new_item*。后续的操作为：将任务标识插入到脉冲阵列中 *tail* 处，脉冲阵列根据插入的任务标识的优先级信息在每个时钟周期进行排序，将 *tail* 指针 +1，将 *count* +1。通过先与 *head* 处的任务标识比较，可以保证下一次出队时，不会因为脉冲阵列的排序延迟导致优先级较高的任务标识无法及时出队；

TAIC 中的多个局部队列之间实现了负载均衡机制，在硬件层面支持了任务在 CPU 之间的迁移。当软件读当前 CPU 绑定的局部队列的端口，从中取出下一个任务标识时，若这个局部队列为空时，TAIC 会从其他不为空的局部队列中窃取任务标识，当多个局部队列均不为空时，会优先从在 TAIC 中局部队列编号最小的局部队列中窃取任务标识。窃取的任务标识仍然通过软件操作的局部队列端口返回。只有当所有局部队列为空时，才会返回空值。当局部队列满时，软件继续向局部队列中插入任务标识，TAIC 会将任务标识插入其他未满的局部队列中，只有当所有局部队列都满时，才会插入失败。软件也可以主动操作其他队列，在软件层面来实现任务在 CPU 之间迁移，但不允许窃取批量任务，这种主动负载均衡的方式也可以利用总线事务来避免使用互斥锁导致的开销。硬件局部队列之间的负载均衡机制对软件透明，因此可以减少在软件层面实现的负载均衡策略导致的同步互斥开销。

由于硬件资源有限，阻塞队列只用于保存因为等待外部设备以及等待任务通知而进入阻塞状态的任务的标识，对应的硬件实现为接收能力表（将在 4.3 中进行描述）。软件可以使用表 3.2 中定义的 `register` 接口实现任务调度中的 `task_block` 接口将某个任务阻塞。当中断处理模块唤醒了接收能力表中的阻塞

任务时，TAIC 会将任务标识插入到任务数量最少的局部队列中，进行负载均衡。

4.3　中断处理与任务通知

TAIC 快速处理中断机制依赖于其维护的接收能力表，它记录了由于等待外部设备中断事件而进入阻塞状态的任务的标识。由于应用程序通常只有一个任务用于处理某个外部中断，因此，接收能力表针对每个外部中断信号只准备了一个表项。对于任务通知机制，接收能力表中还记录了哪些接收方可以接收通知。除了接收能力表外，TAIC 还维护了发送能力表用于记录哪些发送方可以发送通知。接收能力表与发送能力表的结构以及 TAIC 的中断处理和任务通知流程如图所示。

接收能力表中的前几项用于记录与中断处理相关的能力。在使用时，软件通过 `register` 接口向 TAIC 中写中断号 irq_i、接收中断的任务的操作系统标识 os_j、进程标识 $proc_k$ 以及任务标识 $task_l$，TAIC 将任务标识写入到中断号 irq_i 对应的表项中，注册成功即使能了硬件处理外部中断机制。当外部设备产生中断信号时，TAIC 的中断处理模块会根据中断号来检查对应的表项，若其中存在任务标识，将任务标识从表项中取出，并放入就绪队列中的合适位置，以此来唤醒因为等待外部设备中断而进入阻塞状态的任务。由于每个中断号只对应一个表项，因此只能唤醒一个任务，在同一时刻只能有一个进程或内核在使用同一个外部设备。

接收能力表中靠后的表项用于记录与任务通知相关的能力，记录了接收方任务的操作系统标识 $recv\_os_i$、进程标识 $recv\_proc_j$ 和任务标识 $task_k$。发送能力表则只记录与任务通知相关的接收方的操作系统标识 $send\_os_i$、进程标识 $send\_proc_j$。发送能力表项和接收能力表项由通道号 $channel_i$ 连接，构成了（*sender*，*receiver*）二元组，这表示了一条可以跨越不同地址空间、不同特权级的任务通知的通道。发送方任务成功发送通知后，TAIC 将接收能力表中记录的任务标识插入到就绪队列中，唤醒任务。

无论唤醒的是等待中断事件的任务还是等待任务通知的任务，如果唤醒任务的优先级比插入的局部队列所绑定的 CPU 上正在运行的任务的优先级高，则 TAIC 会向 CPU 发送中断信号，CPU 在收到中断信号后，直接进行任务切换，减少唤醒任务的响应延时。由于 TAIC 向 CPU 发送中断信号时，CPU 上正在运行的任务与被唤醒的任务可能处于不同的地址空间和特权级，因此，TAIC 需要能够向处于不同特权级下的 CPU 发送中断信号，论文在 rocket-chip 软核上实现了这种功能：

（1）发送用户态中断：当 CPU 正在运行用户态任务，且被唤醒的任务与正在运行的任务处于同一个地址空间且都运行在用户态时，TAIC 需要直接发送用户态中断，让 CPU 能够直接在用户态进行响应，完成任务切换，减少特权级切换的开销。

论文在 rocket-chip 软核上实现了与用户态中断相关的控制寄存器（*uepc*、*ustatus*、*ucause*、*utvec* 等）以及 *uret* 指令。TAIC 在内部维护了一个 *usip* 寄存器并与 CPU 的 *mip* 寄存器的 *USIP* 位相连，实现了 RISC-V 的 N 扩展。当需要发送用户态中断时，TAIC 只需将 *ssip* 寄存器置为 1，*USIP* 位将被拉高，CPU 跳转至用户态的中断向量寄存器 *utvec* 指向的位置，保存被打断任务的寄存器现场，从 TAIC 中取出被唤醒的任务标识并执行任务。

（2）发送内核态中断：根据 CPU 上正在运行的任务和被唤醒任务的地址空间和特权级的切换的组合（对应到第 3.2 节提到的四类切换，这里还需要考虑具体的用户态/内核态特权级变化），除去上述的同地址空间同用户态特权级时发送用户态中断的情况，其余情况均需要发送内核态中断。TAIC 内部维护了一个 *ssip* 寄存器并将其与 CPU 的 *mip* 寄存器的 *SSIP* 位相连。当需要发送内核态中断时，TAIC 只需将 *ssip* 寄存器置为 1 即可。

4.4　软件适配

为了充分利用 TAIC 提供的任务调度、中断处理和任务通知能力，图展示了软件层次中需要进行相应适配的模块。底层的硬件由 TAIC 和外部设备组成，支撑了上层的操作系统内核，内核之上还会运行应用程序。内核中的任务调度、任务通信、设备管理等模块直接利用 TAIC 进行适配，并在此基础上对系统调用进行适配，用户库则在 TAIC 以及适配后的系统调用模块之上进行适配。这些模块之间不构成强烈的依赖关系，系统可以根据需要进行选择性适配来提高系统的性能。

TAIC 可以与现有的进程、线程或协程任务模型进行结合，线程与协程模型建立在进程提供的地址空间隔离机制上，将 TAIC 的端口映射到各自的地址空间中，对应的伪代码如所示。地址映射添加成功后，将返回一个关于 TAIC 端口的句柄，软件即可操作该句柄使用 TAIC 用于任务调度、中断处理和任务通知。

4.4.1　任务调度适配

由于 TAIC 将调度实体的粒度对齐到了基于执行流的任务，它的局部队列中可能存放了属于不同地址空间、不同特权级的任务，一次调度前后的两个任务可能属于相同地址空间、相同特权级中或者完全属于两个不同的地址空间和特权级。现有的系统中由于调度器能够感知的实体为内核任务，因此，当属于不同地址空间的用户态任务需要切换时，需要先从一个地址空间的用户态任务切换到内核态任务，在内核中进行一次切换，进入到另一个地址空间的内核态任务，再从内核态任务切换到用户态任务。为了减小切换过程的开销，充分利用 TAIC 的调度功能，软件需

要实现一段在所有地址空间中都共享的跳板代码 trampoline，内核与进程的运行时将使用该跳板，用于完成前后跨地址空间、特权级的任务切换。trampoline 的伪代码如所示。软件可以通过多种方式进入其中：

1. 初始化运行时后，通过函数调用进入 trampoline 中，此时 curren_task 为空，设置当前运行的任务标识后，调用 run_task 函数执行任务；
2. 任务（协程）在执行到可让出点后，函数返回到第 16 行，进入新一轮循环，选出下一个任务继续执行；
3. 任务在执行过程中，被中断或异常信号打断时，软件先将被打断任务的上下文暂存在正在运行的栈上，再进入到 trampoline 中。

系统调用属于上述的第 2 种情况，当用户态任务需要进行系统调用时，用户态任务将参数存放在某个位置，此时这个任务无法继续执行，主动返回到 trampoline 函数的第 16 行。在新一轮循环中，取出的下一个任务为内核中用于处理系统调用的任务，切换到内核的地址空间和特权级中，再通过 run_task 函数运行系统调用处理任务。

由于中断导致的任务抢占，其任务切换的过程属于上述第 3 种情况。在进入这个函数之前，被打断任务的上下文将会被保存在栈上，这个栈不能被后续任务复用。run_task 函数根据下一个任务的类型（协程/线程）从堆或者栈上恢复上下文。

4.4.2　任务通知适配

当两个用户态的任务使用 TAIC 进行任务通知时，接收方和发送方需要分别通过系统调用注册任务通知的能力，即可直接进行任务通知。若需要进行任务抢占，还需要使能用户态中断机制，伪代码展示了初始化用户态中断机制的过程，伪代码展示了 CPU 收到中断信号后的任务抢占过程。伪代码展示了收发双方任务通过 TAIC 进行一次任务通知的示例，首先通过 fork 系统调用创建一个子进程作为发送方。父进程作为接收方，当注册了接收能力后，通过 task_block 主动阻塞自己，等待 TAIC 唤醒。发送方在注册了发送能力后，先使用 sleep 系统调用等待一段时间，避免在接收方注册之前先发送通知导致通知失败。发送方通过 send 接口发送任务通知，TAIC 唤醒接收方任务，在其他任务主动让权或者通过用户态中断抢占进入到 trampoline 函数时，TAIC 取出接收方任务标识，从第 13 行恢复执行。

由于中断处理相关的适配与任务通知伪代码中的接收方相关的适配类似，这里不再进行赘述。

4.5　本章小结

本章给出了基于软硬协同的任务调度和中断响应机制的系统的整体架构，描述了 TAIC 内部的具体硬件实现、用户态中断扩展以及对应的软件适配工作。硬件实现的就绪队列、发送能力表和接收能力表等数据结构提供了对任务调度和中断处理能力的支持。软件中与任务切换相关的跳板代码以及与中断处理、任务通知相关的适配代码充分利用了硬件提供的能力，将调度实体的粒度对齐到了基于执行流的任务，减少任务切换、负载均衡以及任务通知的开销。

第 5 章　性能评估

为了验证基于软硬协同的中断响应与任务调度设计的有效性，评估设计中的各个模块对系统性能的影响，本章在 FPGA 平台上围绕以下三个核心维度展开客观评价：

- 任务调度：对基于 TAIC 提供的任务调度功能与传统的软件调度进行多维度对比测试，评估 TAIC 在任务调度开销等方面的影响；
- 中断处理：评估基于 TAIC 提供的中断处理机制相较于传统中断处理机制对中断延时的影响；
- 任务通知：评估基于 TAIC 提供的任务通知能力与传统任务通知机制之间的性能差异。

5.1　测试环境

性能评估实验在 ALINX AXU15EG 开发板上展开，该开发板以 Xilinx Zynq UltraScale+ XCZU15EG MPSoC 为核心板，并配备了 DDR4、双千兆以太网接口等外部设备。核心板分为两部分，分别为处理器子系统和可编程逻辑（FPGA）。其中处理器子系统集成了四核 ARM® Cortex-A53 处理器、双核 Cortex-R5 实时处理器，可以直接对开发板上的资源进行控制。FPGA 部分实现了带有 RISC-V N 扩展的 rocket-chip 软核。rocket-chip 软核使用 TileLink 协议与 TAIC 进行交互，并且将 TileLink 协议转换成 AXI 协议来访问 DDR4 内存资源。因此，软核访问 TAIC 与内存的开销相当。

表 5.1　测试硬件环境

IP 核	**配置**
RISC-V 软核	rocket chip，N 扩展
	100MHz 时钟
	TAIC 中断控制器
以太网	Xilinx AXI 1G/2.5G 以太网子系统（1Gbps）
	Xilinx AXI DMA

测试的硬件环境为 FPGA 平台，具体配置如表 5.1。软件运行于 FPGA 中的 RISC-V 软核上，在微基准测试中，首先测试了软硬件交互接口开销，后续使用了

Linux 6.6.0 版本的基于 RISC-V 指令集的内核进行部分模块的单独测试。而综合测试分别基于 Arceos 单体内核和 ReL4 微内核展开，测试 TAIC 的不同功能对应用程序性能的影响。

5.2　微基准测试

微基准测试主要用于从微观层面评估 TAIC 的性能表现。CPU 与 TAIC 之间的交互在几个时钟周期内完成。其中，与任务调度相关的入队/出队操作在 2～4 个时钟周期内完成，与任务通知相关的注册（注销）发送方/接收方以及发送通知的操作在 8～10 个时钟周期内完成。TAIC 接收到中断信号把处于阻塞队列中的任务标识放入到就绪队列中的过程（中断处理），需要 6～8 个时钟周期。除软硬件交互接口测试外，其余测试在 Linux 6.6.0 系统环境下完成，具体通过 tokio-bench 和 ipc-bench 两个专业工具展开。

5.2.1　任务调度

tokio 作为 Rust 生态中的高性能异步运行时框架，提供了从任务调度到 I/O 操作的全套异步工具链，用于构建高并发、低延时的网络服务。tokio 提供了 rt_multi_threaded 基准测试，用于评估多线程环境下的任务调度性能。tokio 的多线程调度器默认使用了任务窃取算法来动态平衡 CPU 上的任务负载，具体的规则如下：

1. 针对每个工作线程，单独维护一个有界的局部队列；
2. 所有的工作线程共用一个无界的全局队列；
3. 当局部队列达到容量上限时，任务将会被放入全局队列中（全局队列还用于存放 I/O 事件完成后被唤醒的任务）；
4. 当局部队列没有任务时，工作线程会随机地从其他的工作线程的局部队列中窃取一半的任务；
5. 当工作线程连续 60 次（tokio 的默认配置）从局部队列中取出任务后，将会从全局队列中取出一个任务。

论文对 tokio 默认的多线程调度器进行了修改，将其任务窃取算法中使用的任务队列替换为 TAIC 提供的硬件任务队列，来调度 tokio 管理的协程，没有对内核任务调度进行修改。每个工作线程使用 TAIC 提供的硬件局部队列。硬件中没有实现全局队列，被唤醒的任务将直接被放入到对应的局部队列中，而局部队列有限，因此，修改后的 tokio 任务调度器无法满足规则 3 和 5，为了保证公平，测试过程中保证了不会出现局部队列溢出的情况。此外，由于硬件提供了局部队列之间的

负载均衡功能，当局部队列中没有任务时，会自动从其他的局部队列中取出任务，因此，规则 4 也不再需要，但保留了计算随机数的开销来保证测试的公平性。使用上述修改后的 tokio 调度器来评估 TAIC 在任务调度方面的性能表现，最终的结果如图所示，证明了 TAIC 的硬件任务窃取功能在任务调度方面的性能优势。

任务窃取：chained_spawn 测试的逻辑是递归的生成任务，任意时刻，至多存在两个任务，当一个工作线程运行任务 ***A*** 生成一个新的任务 ***B*** 时，其他的空闲的工作线程会窃取新生成的任务 ***B*** 来执行，循环这个过程，直到结束。因此两个测试的性能差异主要取决于任务窃取的性能。TAIC 在任务窃取时直接从局部队列中可以窃取出其他的局部队列中的任务，而 tokio 原本的软件任务窃取需要搜索开销以及局部队列的同步互斥等额外开销，从图可以看出，TAIC 使得任务窃取的开销下降了 30.68%。同理，spawn_local 测试生成更多的任务，并将任务放入当前工作线程的局部队列中，尽管 tokio 原本的任务窃取会直接从其他局部队列中窃取一半的任务，但 TAIC 提供的硬件任务窃取功能几乎是零开销的，因此，TAIC 在 spawn_local 测试（图）中仍然将任务窃取的开销下降了 34.88%。spawn_remote_idle 测试（图）与 spawn_local 测试类似，不同之处在于它将生成的任务放到了远端队列（远端队列用于任务窃取，等价于局部队列）中，两者的性能差距仍然来源于任务窃取的开销（32.41%）。

在 yield 测试（图）中，任务通过 tokio 提供的 `yield_now` 函数主动让权，被暂时存放到软件实现的 defer 队列中。当 tokio 调度器在无法从局部队列中取出任务时，唤醒 defer 队列中的任务，将其放入局部队列中，但测试时没有其他任务，因此任务会马上被放入到局部队列中；在 ping_pong 测试（图）中，两个任务互相等待对方的消息并发送响应，接收方任务阻塞在通道 *A* 上等待发送方发送消息，当发送方发送消息唤醒接收方任务后，发送方任务阻塞在另一个通道 *B* 上等待接收方发送消息，接收方收到通道 *A* 的消息后向发送方发送消息，形成回路；在 spawn_remote_busy 测试（图）中，任务主动让权一次，当再次运行时，会使用 `sched_yield` 系统调用让当前的工作线程进入内核态让权，等工作线程被内核调度后，任务才会继续运行。因此，使用 TAIC 节省的任务窃取开销被分摊到每次任务切换以及等待的过程中，随着任务的等待时间逐渐增加，使用 TAIC 的优化效果逐渐降低，这三个测试的开销分别降低了 24.22%、15.37% 和 9.69%。其中 spawn_remote_busy 测试过程中，因为 Linux 内核产生随机数缓慢，导致了使用 TAIC 时部分测试的平均延时过高。

5.2.2　任务通知

ipc-bench 是一个用于测试 Linux 上的包括信号、管道和 eventfd 等 IPC 机制的开源项目。ipc-bench 针对每种 IPC 机制使用了 ping-pong 通信模型[①]来测试其性能，性能指标包括通信的总时间、平均每次通信的时间、吞吐量等。

论文参考 ipc-bench 项目中对信号、管道和 eventfd 等 IPC 机制的测试方法，实现了使用 TAIC 的任务通知机制进行通信的测试程序。测试程序排除了向 TAIC 注册收发能力以及初始化用户态中断机制的时间，测试了通信的总时间。

在 signal 测试中，发送通知需要使用 `kill` 函数发送自定义信号，而接收通知则需要使用 `sigwait` 系统调用使得线程阻塞在内核态。因此在使用 signal 机制进行通知时，接收双方需要在内核态和用户态之间切换，存在特权级切换的开销。而 pipe 测试则是使用 signal 机制来进行通知，通过写 pipe 文件描述符传输数据；而在 eventfd 测试中，发送通知是通过写入文件描述符来实现的，接收通知则需要接收方不断地尝试从文件描述符中读出信息，存在多次特权级切换。在与 TAIC 相关的单向通信测试中，当不使用中断时，发送方发起通知后，TAIC 直接唤醒阻塞的任务，而接收方线程则不断尝试从 TAIC 中取出被唤醒的任务的标识，一旦取出任务标识，则表示通信过程结束；当使用中断时，接收方收到中断信号后，进入用户态的中断的处理逻辑中，从 TAIC 中取出被唤醒的任务才算收到通知，但需要从中断返回才结束通信。与 TAIC 相关的双向通信测试与单向通信测试类似，不同之处在于，接收方收到发送方的通知后，向发送方发起通知，直到发送方收到通知才结束通信的流程。

表 5.2　IPC 性能对比

测试	吞吐量 msg/s	平均延时 μs	相对延迟比
taic-uni	199215.0	5.02	1
taic-bi	163158.4	6.13	1.221
taic-uni-uint	137127.0	7.29	1.453
taic-bi-uint	108734.7	9.20	1.832
signal	1334.7	749.23	149.258
pipe	649.6	1539.41	306.673
eventfd	374.9	2667.38	531.382

测试过程中，消息大小为 1 bit，发送消息次数为 1000 次。各项 IPC 机制的性

① 在初始化之后，接收方等待发送方的消息，在发送方发送消息后，接收方收到消息并向发送方发送响应，发送方收到响应后结束通信的流程。

能对比如图和表 5.2 所示，实验结果表明 TAIC 具备以下特性。

（1）优化特权级开销。无论是单向通信还是双向通信，使用 TAIC 的任务通知机制进行通信的性能远远优于其他的 IPC 机制，在性能上存在数量级的提升。signal 测试中，发送信号需要进出内核态一次，这个时间开销为 240 μs。接收方在等待信号时，线程阻塞在内核态，在收到信号后，会进入到用户态的信号处理函数中，这存在一次进出内核开销，后续在信号处理函数中使用 `sig_return` 系统调用回到内核态，再返回到用户态继续执行，这也存在一次进出内核开销，因此使用信号机制的开销约为 720（240 * 3）μs。基于 TAIC 的任务通知机制，除了节省特权级切换开销外，还节省了内核中的与信号相关的系统调用处理路径开销以及内核任务调度开销等。根据 taic-bi-uint 测试与 signal 测试的结果对比，TAIC 将任务通知的开销降低了 98.8%。

（2）提供微秒级 IPC。使用 TAIC 实现的四种测试的开销均为微秒级。其中，用户态中断的开销为 144（72 * 2）[①]个时钟周期，再加上一些额外的读写变量的开销，因此 taic-uni-uint 测试的开销比 taic-uni 测试的开销增加了约 2 μs（100 MHz 时钟频率下）。因此，当使用 TAIC 以及用户态中断时，用户态的任务之间可以实现微秒级的 IPC 以及任务抢占。在不使用用户态中断的情况下，若接收方进程正在执行其他的任务，TAIC 内部的中断处理和中断的开销会被其他任务的执行开销掩盖掉，若其他任务的执行时间也处于微秒级（轻尾负载时），即使不需要中断机制，基于 TAIC 的任务通知同样可以达到微秒级的时间尺度，但在重尾负载时，仍然需要发送中断来避免队头阻塞（就绪队列中的任务无法被及时处理），减小延迟。

5.3　综合测试

综合测试分别在 Arceos 单体内核环境和 ReL4 微内核环境下展开，分别评估了 TAIC 的中断处理和任务通知能力对现有软件的影响。

5.3.1　Arceos 网卡驱动优化

Arceos 是一个用 Rust 语言编写的模块化操作系统，允许开发者通过条件编译按需选择功能模块，将应用程序与内核编译为单一镜像，减少由于系统调用的开销，从而实现高度定制化的轻量级系统。论文针对 Arceos 系统的网络驱动进行了适配，由于 Arceos 缺少对中断的支持，因此需要定期切换到网卡驱动的线程来收发数据包，而不是通过中断来唤醒线程。论文利用了 TAIC 的外部中断处理能力，

① 在对 TAIC 提供的任务通知机制测试前，论文首先测试了 CPU 分别在用户态和内核态下的中断延迟（包括了软件保存被打断任务上下文以及进行中断分发的开销），两者所需要的时钟周期分别为 66 和 92。使用 TAIC 时，中断延迟会在此基础上增加 6 ～8 个时钟周期的内部处理开销，造成的影响可以忽略。

对 Arceos 的网卡驱动进行了优化，将网卡的中断打开（被 PLIC 屏蔽），而 TAIC 会接收到网卡的中断信号，并将对应的线程唤醒。通过这种方式，CPU 不需要定期切换到网卡驱动的线程来检查是否收到数据包，减少了查询的开销。

表 5.3　YCSB 工作负载描述

负载类型	描述
YSCB-A	50% get 操作 + 50% update 操作
YSCB-B	95% get 操作 + 5% update 操作
YSCB-C	100% get 操作
YSCB-D	获取最近插入的数据
YSCB-F	50% get 操作 + 50% read-modify-write 操作

在上述优化的基础上，论文定制了一个基于 Arceos 系统的 Redis 服务，使用了 YCSB（Yahoo! Cloud Serving Benchmark）测试工具评估了 TAIC 的中断处理能力，如表 5.3 所示。对于除 YCSB-D 之外的所有工作负载，测试首先运行 YCSB-LOAD 用 10K PUT 请求预热 Redis，然后运行相应的 10K 请求工作负载。对于 YCSB-D，在 YCSB-LOAD 之后，测试为最近写入的键发出 10K GET 请求。所有测试均使用 128 个线程。结果如图所示，包括吞吐量和 99 百分位数延迟。此外，论文还分析了在不同吞吐量水平下 YCSB-A 的尾部延迟，如图所示。

基于 YCSB-A 和 YCSB-F 的比较结果，TAIC 的中断处理能力将 Redis 在频繁写操作场景中的性能提升了 6% 到 7%。对于涉及更多读操作的 YCSB-B、YCSB-C 和 YCSB-D 工作负载，则优化有限，仅带来了 1% 到 2% 的提升。论文在对 Arceos 进行适配时，仅对网卡驱动模块进行了适配，没有对核心的任务调度模块进行优化（软件适配模块见图），因此取得的性能优化比较有限。但这些结果仍然可以证明，TAIC 的中断处理能力能够减小应用程序与 IO 交互时的轮询开销。此外，由于 Arceos 缺少对中断的支持，论文没有比较与原本的中断机制的性能差异。

5.3.2　ReL4 异步 IPC 优化

ReL4 是一款用 Rust 编写的，兼容 seL4 基本系统调用的高性能异步微内核，它借鉴了 seL4 的权限管理、内存管理和 SMP 设计等基本框架，但在 IPC 机制和任务调度方面进行了优化和改进。seL4 使用同步 IPC 作为进程间通信的主要方式，并且借助需要内核转发的 notification 机制来实现用户态的中断处理和异步通信，然而，seL4 中的同步 IPC 机制会导致大量的上下文切换和二级缓存失效，同时由于阻塞等待，系统无法充分利用多核的性能，导致 seL4 难以满足应用逐渐增加的性

能需求。ReL4 针对这些 seL4 存在的问题，在用户态实现了一套异步协程运行时，使用田凯夫设计的 uintr 用户态中断控制器提供的通知机制，实现了用户态的异步 IPC 机制，减小了特权级切换的开销。论文使用 TAIC 提供的任务通知能力，对 ReL4 的异步 IPC 机制展开了进一步优化，减小了用户态中断和任务分发的开销。

以 ReL4 IPC 中的 Call 为例，如图，收发双方（客户端进程和服务端进程）由若干个工作（*worker*）任务和分发（*dispatcher*）任务共同组成，其中 *worker* 任务负责发起请求和处理响应，dispatcher 任务负责分发请求和响应。具体的通信流程为：

- 客户端的 *worker* 任务将请求提交给运行时，并阻塞自己；
- 运行时根据请求参数生成存放在共享缓冲区中的请求条目，并检查共享缓冲区中的标志位，判断服务端的 *dispatcher* 任务是否处于阻塞状态，如果服务端的 *dispatcher* 任务不处于阻塞状态，则无需任何操作；如果 *dispatcher* 任务处于阻塞状态，运行时将共享缓冲区中的标志位清空后，向服务端发送通知。
- 服务端的 *dispatcher* 任务被唤醒后，从共享缓冲区中读出请求并分发给对应的 *worker* 任务；
- *worker* 任务完成处理后将结果提交给运行时并阻塞自己，运行时将结果写回到共享缓冲区中再按照相同的方式判断是否需要唤醒发送方的 *dispatcher* 任务。

根据通知机制以及使用方式，ReL4 IPC 路径可以分为以下几种方式：

1. 基于 uintr 用户态中断机制的通知：其中一方的运行时需要发送用户态中断给对方，对方在用户态中断处理函数中唤醒 *dispatcher* 任务，由该任务来唤醒具体的处理请求的任务，即图中的虚线箭头表示的流程；
2. 基于 TAIC 任务通知机制的通知：使用 TAIC 存在两种方式，一种方式是使用 TAIC 直接唤醒对方的 *dispatcher* 任务，即图中的黑色箭头流程；另一种方式是使用 TAIC 直接唤醒对方处理请求的 *worker* 任务，即图中的红色箭头流程；这两种方式都会减小由于用户态中断导致的开销，对缓存友好，并且可以将中断处理与任务调度结合起来，将 TAIC 的两种能力结合起来，提高系统的性能。

为了全面地评估 TAIC 对 ReL4 IPC 机制的影响，本节在 ReL4 中构造了一个服务端进程和客户端进程进行 ping-pong 测试，图中的一条环路表示一条客户端与服务端之间的连接，连接的数量即为测试的并发度。论文测试在不同并发量的场景下，TAIC 对 ReL4 异步 IPC 机制的影响，测试的结果如图所示。

因为 ReL4 的异步运行时实现了自适应机制，可以批量提交异步 IPC，在并发

度增加时，客户端会将多个 IPC 请求通过一次通知来让服务端进行处理，因此，并发度增加时，TAIC 对 ReL4 的异步 IPC 机制的影响逐渐减小。在连接数为 1 时，使用 uintr 硬件控制器时，每发起一次 IPC 请求就需要发送一次用户态中断，相反使用 TAIC 则没有用户态中断的开销，将原本每次 IPC 开销降低了 45.7%。在多核环境下，因为接收方进程和发送方进程同时在线，TAIC 唤醒任务后，接收方或者发送方进程的运行时可以快速取出 *dispatcher* 任务并将请求或结果分发给对应的 *worker* 任务，而单核情况下则接收方和发送方不能同时在线，存在特权级切换等开销。因此，TAIC 在多核环境下的优势更加凸显，它将每次 IPC 开销降低了 65.8%。

5.4　本章小结

本章从任务调度、中断处理和任务通知三个维度展开对 TAIC 的评估。在微基准测试中，TAIC 将 tokio 的任务调度中的负载均衡开销降低了 30%，将任务通知的开销降低了 98.8%。在综合测试中，TAIC 将基于 Arceos 实现的 Redis 服务的性能提升了 6% 到 7%，将 ReL4 中的异步 IPC 开销降低了 45.7% 和 65.8%。根据微基准测试和综合测试的结果来看，基于 TAIC 展开的软硬协同优化设计的系统相较于基于传统机制的系统，在性能上有了明显提升，验证了基于软硬协同的任务调度和中断响应机制的有效性。但是，由于时间有限，本章没有按照第 3 章和第 4 章中描述的设计与实现，对系统进行彻底改造，而是在现有的系统上分别对各个部分进行了测试，在测试中还存在着不足之处，不能完全体现基于软硬协同的任务调度和中断响应机制的优势。

第 6 章　总结与展望

6.1　总结

论文在总结现有研究工作的基础上，针对任务调度和中断响应机制中存在的问题和挑战，围绕着任务模型展开了基于软硬协同的任务调度和中断响应机制的研究。论文的主要贡献包括：

（1）论文引入了基于执行流的任务模型，将进程、线程、协程统一到基于执行流的任务模型中。论文在已有的进程、线程和协程任务模型的基础上，分析了它们在系统中各自承担的角色，剥离出进程具备的隔离特性，线程和协程具备的并发特性，建立了基于执行流的统一任务模型，从而使得处于不同地址空间、特权级的任务统一到一个维度上，能够在一个调度器中进行调度，降低了调度的复杂性。

（2）论文设计了基于软硬协同的任务状态维护方法，减小了任务调度过程中的开销。论文基于执行流的任务模型，为任务设计了分层复合型任务标识，使得硬件可以通过任务标识来感知任务所处的地址空间、特权级以及优先级等信息。硬件在内部维护了就绪队列、阻塞队列以及正在运行的任务标识，从而可以感知任务所处的状态，通过在这些队列之间迁移任务标识来实现高效的任务状态变迁以及负载均衡机制。论文基于软硬协同的任务状态维护方法实现了硬件化的任务调度器，实验结果表明，使用硬件化的任务调度器的 tokio 运行时的调度开销降低了 30%。

（3）论文实现了硬件响应中断机制，消除了中断机制导致的负面影响。论文基于硬件化的任务调度器，设计了硬件响应中断机制。软件首先维护好阻塞的任务与等待的中断事件的关系，当硬件收到中断信号后，直接将处于阻塞状态的任务唤醒，不仅避免了中断信号打断 CPU 所导致的开销，而且避免了软件定期轮询的开销，经过硬件中断响应机制改造后的网卡驱动使得 Redis 的吞吐量提升了 7%。论文还将中断信号的发起方从外部设备扩展到软件上运行的任务，扩展实现了基于硬件的任务通知机制，建立了跨越不同地址空间、特权级的任务之间的直接通信通道，避免了大量的地址空间、特权级切换开销，将基于信号机制的任务通知的开销降低了 98.8%，将 ReL4 中的异步 IPC 开销降低了 45.7% 和 65.8%。

6.2　展望

论文虽然取得了一些进展，但由于时间和条件关系，目前的工作还不十分完善，存在一些问题有待进一步解决：

（1）统一任务模型存在的调度挑战。论文将处于不同层次的任务统一到基于执行流的任务模型上，提出将调度实体的粒度对齐到基于执行流的任务模型上。但处于不同层次的任务所担任的角色不同，如何将这些多层次的任务统一到一个维度来进行调度，满足处于不同层次的任务的需求，这需要展开进一步的研究。

（2）软硬协同深度不足。论文中展开的实验仅仅对部分软件模块进行了适配，以展示硬件的不同功能对系统的性能优化，未来还需要展开对软件的全面适配或重新设计所有与调度相关的软件模块，以全面提升系统整体性能。

（3）硬件资源的限制。硬件化的任务调度器要求在硬件中保存任务标识，硬件资源的消耗随着任务数量的增加而增加。一方面，需要进一步优化已有的实现，减少硬件资源使用量；另一方面，需要进一步推动软硬协同，在硬件资源有限的条件下，针对应用的需求来合理分配资源，将部分对性能需求较高的任务相关的调度和通知由硬件来完成，其余的任务调度和通知则由软件来完成，从而满足不同应用的需求。