基于软硬协同的中断响应和任务调度研究

(申请清华大学工学硕士学位论文)

培 养 单 位： 计算机科学与技术系

学　　科： 计算机科学与技术

研　究　生： 赵 方 亮

指 导 教 师： 向　　勇　副研究员

二〇二五年三月

Research on interrupt response and task scheduling based on software and hardware collaboration

Thesis submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Master of Science

in

Computer Science and Technology

by

Zhao Fangliang

Thesis Supervisor : Associate Professor Xiang Yong

March, 2025

学位论文公开评阅人和答辩委员会名单

公开评阅人名单

刘 XX	教授	清华大学
陈 XX	副教授	XXXX 大学
杨 XX	研究员	中国 XXXX 科学院 XXXXXXX 研究所

答辩委员会名单

主席	赵 XX	教授	清华大学
委员	刘 XX	教授	清华大学
	杨 XX	研究员	中国 XXXX 科学院 XXXXXXX 研究所
	黄 XX	教授	XXXX 大学
	周 XX	副教授	XXXX 大学
秘书	吴 XX	助理研究员	清华大学

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）按照上级教育主管部门督导、抽查等要求，报送相应的学位论文。

本人保证遵守上述规定。

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘 要

在请求完成后，I/O 设备会发出一个中断信号来通知 CPU，这会中断正在运行的任务并影响吞吐量，或者 CPU 主动轮询 I/O 设备的状态寄存器，消耗宝贵的 CPU 周期。在使用超高速 I/O 设备时，中断驱动和轮询机制的有害影响不容忽视。

为了解决这个问题，我们将任务调度器的功能卸载到中断控制器（TAIC）。TAIC 直接维护任务队列并唤醒被阻塞的任务，以实现快速唤醒机制。这种方法保护了 CPU 免受中断的影响，减少了 CPU 在中断响应以及维护调度队列方面的开销，从而降低了 I/O 响应延迟。我们将这一机制与网络卡驱动程序集成，并在 FPGA 平台上进行了验证。它减少了与中断相关的开销并降低了 CPU 利用率。在使用 YCSB 工作负载的 Redis 上的评估表明，TAIC 实现了低尾部延迟，并将吞吐量提升了 6%。

在通用操作系统中，随着并发量的增加，传统的多线程模型由于与内核多线程相关的高上下文切换成本而变得不足。在本文中，我们提出了一种新的并发模型，称为 COPS。COPS 采用基于优先级的协程模型作为基本任务单元，取代了高并发场景中的传统多线程模型，并为内核和用户空间协程提供了一个统一的基于优先级的调度框架。COPS 将协程作为操作系统中的第一类公民引入，以提供异步 I/O 机制，使用内核协程作为 I/O 操作和设备之间的桥梁，以及用户协程来连接应用程序和操作系统服务。

我们基于 COPS 设计了一个原型 Web 服务器，并在基于 FPGA 的系统上进行了广泛的实验，以评估 COPS。结果表明，所提出的模型在保持相对较低的开销的同时，相比于多线程模型，在大型并发应用中实现了高达四倍的吞吐量。

关键词：关键词 1；关键词 2；关键词 3；关键词 4；关键词 5

Abstract

An abstract of a dissertation is a summary and extraction of research work and contributions. Included in an abstract should be description of research topic and research objective, brief introduction to methodology and research process, and summary of conclusion and contributions of the research. An abstract should be characterized by independence and clarity and carry identical information with the dissertation. It should be such that the general idea and major contributions of the dissertation are conveyed without reading the dissertation.

An abstract should be concise and to the point. It is a misunderstanding to make an abstract an outline of the dissertation and words “the first chapter”, “the second chapter” and the like should be avoided in the abstract.

Keywords are terms used in a dissertation for indexing, reflecting core information of the dissertation. An abstract may contain a maximum of 5 keywords, with semi-colons used in between to separate one another.

Keywords: keyword 1; keyword 2; keyword 3; keyword 4; keyword 5

目 录

插图清单

附表清单

符号和缩略语说明

PI	聚酰亚胺

第 1 章　引言

1.1　课题研究背景与意义

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了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，数据中心需要每秒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，既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。

这些应用场景的复杂化对计算机系统的性能提出了复杂的要求，而操作系统作为计算机系统的核心软件组件，通过分层抽象机制实现了对底层硬件资源的统一管理，其进（线）程管理、内存管理、设备管理和文件系统等功能模块对物理层面的中央处理器（CPU）、存储器、输入输出设备以及网络接口等资源进行了标准化抽象，向上层应用提供了访问系统服务的接口。

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，还渗透至操作系统的其他功能模块中。例如，在进（线）程管理中，任务调度决定了进（线）程执行的顺序，针对不同的任务类型采取差异化的调度策略，既保证了交互式任务的响应速度，又维持了计算密集型任务的吞吐率；在设备管理模块中，中断机制作为连接应用程序与外部设备的核心桥梁，通过事件驱动的方式实现异步通信。当外部设备产生中断信号时，系统首先保存当前任务执行状态，随后调用中断服务程序解析设备事件，更新相关任务的状态，并将其重新加入调度队列，在中断返回路径上，触发任务调度机制进行决策，基于任务时效性要求评

估是否需要立即进行任务切换，从而有效降低 I/O 任务的等待延迟，从而满足应用程序低延时的需求；

针对操作系统中的任务调度机制展开研究，是计算机系统研究的重要方向之一。本文基于软硬协作的方法，展开对中断响应与任务调度机制的研究，旨在降低系统的响应延时、提高系统的吞吐量，增强人机交互体验，以应对复杂多样化的应用场景需求。

1.2　国内外研究现状

1.2.1　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进（线）程，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，从而营造出多进（线）程流畅运行的假象。

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，当中断处理完成后恢复现场，造成直接开销；并且，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[1-5]。

在早期，I/O 设备每秒只能完成几百个请求，传统的中断机制足以支持软件层面上的并发处理，由中断引起的开销在系统稳定性方面可以忽略不计。然而，现代高速 I/O 设备每秒可以完成数百万个请求[6-8]。如果使用传统的中断机制，如此高的 I/O 请求完成频率可能会导致不可接受的上下文切换开销。目前，针对这个问题，已经展开了大量的研究工作，主要研究方向如下：

- 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[9-12]。例如，IX[13] 和 DPDK[14]（以及 SPDK[15]）直接将设备暴露给应用程序，并在用户空间实现轮询，绕过了内核和中断的需求。Intel 的 DDIO[16] 和 ARM 的 ACP[17] 允许网络设备直接将传入数据写入处理器缓存，通过将内存映射的 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[18]。管海兵等人提出的 sEBP（智能基于事件的轮询模型）通过收集各种系统事件优化了网络卡的轮询机

制[19]。为了减少轮询中的高 CPU 开销，Linux 内核建议使用混合轮询[20-21]。这种方法避免轮询整个 I/O 等待时间。相反，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。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。这会导致更高的 CPU 利用率，尤其是在设备不太活跃的情况下[18,22-24]。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。

- 混合中断与轮询：除了单独使用中断或轮询机制外，一些研究[22,25] 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络卡数据，并在有可运行线程时切换到中断方式[25]。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[22]。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。
- 硬件支持的快速上下文切换：大量的研究致力于优化中断上下文切换的开销[26-29]。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[30]。多管道寄存器架构（Multi Pipeline Register Architecture，MPRA）处理器也已经展示了在几个时钟周期内完成上下文切换的能力，并且利用这一特性优化了中断处理[31-32]。然而，这些基于特定硬件平台的优化方法缺乏通用性。
- 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[33-38] 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[33]。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[34-35]。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[36-37]。市场上许多网络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[38]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

并与混合中断与轮询存在着相同的问题。

- 硬件智能：一些研究还在探索如何通过硬件增强来提升系统的智能水平。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[23]。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[39]。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[40]。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[41]，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高同步和互斥开销，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和互斥机制。

1.2.2　任务调度

任务调度机制直接决定了系统的性能表现，针对不同的应用场景需要的性能指标，目前已经存在着大量的研究工作。

源于中断机制的固有特性，低时延与高吞吐这两项指标是一对矛盾体。一方面，中断提供的抢占机制有效的缓解了传统轮询模式下的队头阻塞问题，保证关键 I/O 请求的优先处理，从而降低响应延时；另一方面，中断机制带来的上下文切换开销、缓存失效开销以及频繁中断造成的中断风暴等问题，会导致系统的吞吐量降低。很多研究工作围绕着这两项指标展开。Wierman 等人在论文[42] 中证明，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进最优的。并且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。在此基础上，Prekas 等人在开展 ZygOS[43] 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。近年来，有很多研究工作围绕着构建低时延服务进行。在上述的理论前提下，他们结合了操作系统的一些其他优化手段，成功构建了一些低时延服务[1,44-47]。Shenango 保留额外的处理器核心用于满足高负载情况下的低延时要求[46]，但在低负载的情况下存在着 CPU 资源浪费的问题；Shinjuku[45]

使用专门的调度线程为每个请求创建上下文来支持抢占和重调度，当工作线程上的某个请求耗时过多时，调度线程向该工作线程发起中断，让工作线程能够及时响应需要快速处理的请求；Concord[44] 则在 Shinjuku 的基础上进行了优化，它通过编译器插桩达到了用协作式调度近似 Shinjuku 中的抢占式调度的效果，减小了 Shinjuku 中工作线程由于中断抢占带来的开销，但它与 Shinjuku 都需要一个专用的调度核分发任务；Skyloft[47] 使用了 x86 架构下的用户态中断技术[48]，提供了一个通用且高效的用户态调度框架，满足了应用程序多样化的需求，但它只能在特定的硬件平台上发挥优势。

随着吞吐量需求的增加，任务调度的对象也开始发生变化，开始由多线程模型向协程这种轻量级、且易于与异步机制结合的任务模型转化。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[49] 在分布式仲裁系统中使用协程；Capriccio[50] 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[51] 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[52] 在中断耗时和中断延迟方面远胜于用 C 语言实现的 FreeRTOS[53]。但这些基于协程模型的任务调度仍然需要中断机制来保证低延时的性能需求。

这些研究取得了相当不错的成果，但仍然存在一些问题。这些研究工作大多是基于软件层面的优化，甚至某些工作是针对特定的硬件设备展开，由软件来适配硬件资源（CPU 等）以及硬件特性（中断机制、I/O 设备特性），其中存在着硬件资源浪费等问题，而围绕着硬件展开的工作，限制了只能在特定硬件平台上才能发挥优势，缺乏通用性。

在应对复杂多样的应用场景时，中断机制与任务调度机制是相辅相成，单纯从某一机制展开研究无法满足复杂多样化的应用场景需求，关于任务调度中的调度策略、任务模型的研究已经足够全面，而中断机制的固有特性也不足以做出新的突破。因此，本文尝试从软硬件结合的角度出发，展开对中断响应与任务调度机制的研究，以期在降低系统的响应延时、提高系统的吞吐量方面取得新的突破。

1.3　本文的主要研究内容与组织结构

任务调度涉及到任务模型、调度算法、同步互斥、中断处理、通信等多个领域，其中每个领域都有众多的研究课题，针对任务调度机制展开研究，是一项复杂的系统性工程。

本文第一章为引言。在这一章中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以及现实意义，然后立足于中断响应和任务调度的性能，对国内外相关研究进行了详细的分析并介绍了各自的优缺点。

本文第二章为相关技术研究，作者从任务模型出发，介绍了任务模型的变迁过程，并介绍了在任务变迁过程中涉及到的任务调度算法、I/O 模型的变化，并介绍了用户态中断技术的研究现状。

本文第三章基于前文的研究内容，展开了基于软硬协同的中断响应和任务调度设计。相较于传统的基于性能指标进行的任务建模，作者提出一种基于任务运行环境的建模方法，提出了一种统一的任务模型。在此统一任务模型的基础上，作者建立了一种软硬协同的任务状态模型，定义了一套软硬件交互的接口，实现了软硬件协作，共同维护任务状态的快速中断响应和任务调度机制。

本文第四章将第三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中断响应和任务调度的系统原型。

本文第五章对前两章所提出的设计方案与系统原型进行了验证试验，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。

论文第六章是总结与展望，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

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件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

在早期的操作系统中，任务以物理作业卡为载体形成离散的批处理单元，其建模遵循冯·诺伊曼提出的串行执行假设[54]，在内存中仅存放一个作业，每个作业作为封闭的地址空间实体独占系统资源，按照顺序自动执行磁带上的作业，虽然减少了人工操作时间，但 CPU 在等待 I/O 操作时仍会闲置，导致整体效率受限。这一问题促使多道程序设计（Multiprogramming）概念的诞生，允许多个程序同时驻留内存并共享 CPU 资源，当某个程序因 I/O 请求暂停时，CPU 可立即切换执行其他程序，显著提升了资源利用率和系统吞吐量，但多个程序之间没有形成保护边界。并且，批处理作业模型也因为缺乏人机交互性，在程序调试和实时响应方面存在局限。

批处理作业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。

2.1.2　进程模型

Saltzer 于 1966 年完善了进程作为虚拟处理器的形式化定义[55]，Dijkstra 在 THE 系统中首次提出“协同顺序进程”概念[56]。进程模型作为现代操作系统的核心抽象，借助了硬件内存管理单元（MMU，Memory Manage Unit）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，例如运行态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就绪态。

进程模型使用进程控制块（PCB，Process Control Block）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），并将 PCB 作为任务的唯一标识以及任务调度的基本单位，操作系统通过管理 PCB 来实现进程的创建、调度、终止等操作。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，将缩短了任务平均周转时间[57]，但此时进程既是资源容器（拥有文件描述符、内存映像等），又是调度实体，但重量级上下文切换制约了细粒度并发。

2.1.3　线程模型

由于硬件提供的地址空间隔离机制，进程间通信（IPC，Inter-Process Communication）需要依赖一些复杂机制（例如管道、套接字或共享内存），其上下文切换涉及特权级切换、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。并且，由于进程同时是资源分配的单位，创建和销毁的开销也较大，这些问题促使多线程模型的出现。

线程模型则将执行单元与资源所有权解耦，进程仍然为系统资源分配的单位，线程为执行和任务调度的单元，共享进程的资源（如内存和文件句柄等），降低通信的开销，每个线程在线程控制块（TCB，Thread Control Block）中维护了状态信息（包括程序计数器、通用寄存器等）。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。并且，硬件上的多核处理器技术（例如对称多处理器和非对称多处理器）①以及超线程技术②进一步推动了多线程模型的发展，进一步提高了计算机系统的性能。

多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。混合模型则结合两者优势，既降低开销又支持多核并行。

然而，线程模型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，这带来了额外的开销。例如，互斥锁的获取与释放操作

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。

在竞争激烈场景下可能导致线程阻塞，而自旋锁虽减少上下文切换但会导致 CPU 空转。

2.1.4　协程模型

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[58]：

- 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 continuations）的引入和多线程作为并发编程的“事实标准”采纳，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[59]。如今，现代编程语言（如 C++、Go、Rust、Python、Kotlin 等）都提供了不同程度的协程支持。

协程作为协作式任务单元，其无栈式设计（Stackless）相较于进（线）程模型，占用的内存资源更少。协程通过 CPS（continuation-passing style）或者状态机保存执行现场，将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。

2.2　任务调度

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，通过创新性的双队列轮转机制实现了时间复杂度优化，$O(1)$ 调度器采用基于优先级的分层队列架构，将可运行任务划分为 *active*（活跃队列）和 *expired*（过期队列）两个独立集合。当任务的时间片耗尽时，调度器会即时执行优先级重计算，而非等待所有任务时间片耗尽后才统一处理，这一设计显著降低了调度决策的时间复杂度。而 Linux 内核使用的公平调度算器（CFS，Completely Fair Scheduler）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。

在任务调度算法的建模和优化领域，现有研究通常围绕系统核心性能指标展开针对性设计。以 Concord[44] 为例，它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。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，建立任务最坏响应时间和优先级分配的数学模型，有效的降低了传统调度在众核环境下的调度开销[60]。

然而，现有研究多聚焦于单一或有限维度的性能指标建模，在动态可重构的系统中，这些确定性的模型与动态场景的适配不足，难以适应复杂应用场景的多样化需求。

2.3　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 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它任务。常见的 I/O 模型包括以下几类：

- 同步阻塞 I/O：应用程序发起 I/O 操作后，线程会完全阻塞，直到内核完成数据准备和拷贝到用户空间两阶段操作。例如发起 `read()` 系统调用时，若内核缓冲区无数据，线程进入休眠状态，直到数据到达后被唤醒。这个模型逻辑简单直观，但线程资源利用率较低，适用于低并发的场景，在高并发时需要创建大量线程而导致内存与调度开销增大。
- 同步非阻塞 I/O：应用程序通过 `fcntl()` 系统调用将文件描述符设置为非阻塞模式，当发起 `read()` 系统调用时，若内核无数据，立即返回 `EWOULDBLOCK` 错误，不会将线程阻塞在内核中，线程回到用户态可以执行其他操作，但仍然需要定期轮询 I/O 状态。这种模式减少了线程空闲等待的时间，提升了单线程利用率，但定期轮询导致了 CPU 空转，存在资源浪费，并且受到轮询间隔的影响，在数据准备就绪到应用程序进行处理存在一定的响应延时。
- I/O 多路复用：单个线程通过 `select()`、`poll()`、`epoll()` 等系统调用，阻塞在内核中，内核会监控多个文件描述符的 I/O 状态，当其中某个文件描述符就绪时，系统调用就会返回到用户态，线程可以执行 I/O 操作。这种模型适用于大量 I/O 事件的场景，避免了线程创建和销毁的开销，但这些机制需要内核维护额外的数据结构以及在内核与用户程序之间拷贝文件描述符集合，存在着一定开销。

- 异步 I/O：应用程序通过异步读/写接口发起 i/O 操作后，立即从内核态回到用户态执行其他操作，当 I/O 事件准备就绪，内核将数据拷贝至应用程序的缓冲区中，内核发送一个信号或者或者基于线程的回调函数来完成 I/O 处理。以信号机制作为异步通知机制为例，应用程序使用 `sigaction` 系统调用注册 `SIGIO` 信号后，回到用户态执行其他的任务，当 I/O 准备就绪时，内核向应用程序发送 `SIGIO` 信号，当应用程序下一次进入内核并返回到用户态时，会进入预先注册的信号处理函数中，完成 I/O 事件的处理，在信号处理函数尾部，通过 `sigreturn` 系统调用使应用程序从被打断处恢复执行。这种机制避免了轮询造成的 CPU 开销，但由于信号处理函数需要等到应用程序下一次进入内核并返回时才能执行，这与同步非阻塞 I/O 模型存在着类似的响应延时，并且增加了额外的系统调用开销。此外还存在其他的异步通知机制，例如用户态中断技术。异步 I/O 模型通过解耦 I/O 操作与程序执行流，突破了传统同步模型的性能瓶颈，在构建高并发的云服务场景下效果显著。

在选择 I/O 模型时，需综合考虑应用场景的负载类型、性能需求、并发规模及系统资源限制等因素。例如，高并发短连接场景（如 Web 服务器）通常采用 I/O 多路复用模型以高效管理大量连接，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；对于简单低负载应用（如嵌入式设备），阻塞式 I/O 因其实现简单且资源消耗低可能是更优选择。此外，跨平台兼容性、开发复杂度（如回调机制）和操作系统特性（如 Linux 对异步 I/O 的模拟实现）等因素也需要考虑。

2.4　用户态中断

用户态中断（User-Level Interrupts）作为近年来计算机体系结构和操作系统领域的研究热点，旨在绕过传统内核态中断处理的高开销，通过硬件与软件的协同设计实现用户态程序的低延迟异步事件响应。其核心思想是通过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，在开启内核页表隔离机制后，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[47]、uProcess[61] 使用了 Intel 提出的 UINTR（User Interrupts）扩展[48] 来实现用户态的抢占式调度。此外，RISC-V 指令集[62] 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[63]。

用户态中断技术除了可以为实现用户态抢占式调度提供了硬件原生支持，还可通过内核旁路（Kernel Bypass）和零拷贝（Zero-Copy）机制显著加速 IPC。用户

态中断充当通知机制，中断信号直接触发用户态处理逻辑，并且通过共享内存或其他机制绕过多层内核协议栈与内存拷贝操作，从而消除传统 IPC 中因系统调用、上下文切换和数据复制导致的开销。这对于高性能通信具有重要影响，但用户态中断技术也面临着安全隔离性（如恶意中断注入、中断嵌套）、跨平台兼容性（需依赖特定 CPU 架构）以及调试复杂性（缺乏内核调试接口）等挑战。

2.5　本章小结

本章围绕中断响应与任务调度的相关技术展开探讨。首先从硬件体系结构的迭代升级切入，揭示了计算机系统如何通过软硬件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、吞吐量等性能指标的需求展开对任务调度算法以及 I/O 模型的讨论，最后介绍了用户态中断技术的研究现状。这些技术的发展和应用为本文后续的研究工作提供了理论基础和技术支持，尤其是用户态中断技术，它打破了内核与用户态的壁垒，推动了软硬件深度融合的进程。

第 3 章　基于软硬协同的中断响应和任务调度设计

第 4 章　基于软硬协同的中断响应和任务调度实现

第 5 章　性能评估

5.1　测试环境

5.2　特征分析

5.3　微基准测试

5.4　综合测试

5.5　本章小结

第 6 章　总结与展望

参考文献

[1] Adam, et al. IX: A protected dataplane operating system for high throughput and low latency [C]//11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). Broomfield, CO: USENIX Association, 2014: 49-65.

[2] Gruss, et al. KASLR is Dead: Long Live KASLR[C]//Bodden E, Payer M, Athanasopoulos E. Engineering Secure Software and Systems. Cham: Springer International Publishing, 2017: 161-176.

[3] Livio, et al. FlexSC: Flexible system call scheduling with Exception-Less system calls[C]//9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10). Vancouver, BC: USENIX Association, 2010: 33–46.

[4] David, et al. Context switch overheads for linux on arm platforms[C]//ExpCS '07: Proceedings of the 2007 Workshop on Experimental Computer Science. New York, NY, USA: Association for Computing Machinery, 2007: 3–es.

[5] Tsafrir, et al. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)[C]//ExpCS '07: Proceedings of the 2007 Workshop on Experimental Computer Science. New York, NY, USA: Association for Computing Machinery, 2007: 4–es.

[6] Intel. Intel® Optane™ SSD DC P4800X Series (375GB, 2.5in PCIe x4, 3D XPoint™) Product Specifications[EB/OL]. 2024[2024-05-12]. https://www.intel.com/content/www/us/en/products/sku/97161/intel-optane-ssd-dc-p4800x-series-375gb-2-5in-pcie-x4-3d-xpoint.html.

[7] Digital W. How to Read the Ultrastar Model Number[EB/OL]. 2024. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/data-center-drives/ultrastar-nvme-series/data-sheet-ultrastar-dc-sn200.pdf.

[8] Intel. Intel® 82598EB 10 Gigabit Ethernet Controller[EB/OL]. 2024[2024-05-12]. https://www.intel.com/content/www/us/en/products/sku/36918/intel-82598eb-10-gigabit-ethernet-controller/specifications.html.

[9] Caulfield A M, De A, Coburn J, et al. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories[C/OL]//2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. 2010: 385-395. DOI: 10.1109/MICRO.2010.33.

[10] Yang J, Minturn D B, Hady F T. When poll is better than interrupt[C/OL]//10th USENIX Conference on File and Storage Technologies (FAST 12). San Jose, CA: USENIX Association, 2012. https://www.usenix.org/conference/fast12/when-poll-better-interrupt.

[11] Vučinić D, Wang Q, Guyot C, et al. DC express: Shortest latency protocol for reading phase change memory over PCI express[C/OL]//12th USENIX Conference on File and Storage Technologies (FAST 14). Santa Clara, CA: USENIX Association, 2014: 309-315. https://www.usenix.org/conference/fast14/technical-sessions/presentation/vucinic.

[12] corbet. Driver porting: Network drivers[EB/OL]. 2024[2024-05-12]. https://lwn.net/articles/30107/.

[13] Belay A, Prekas G, Klimovic A, et al. IX: A protected dataplane operating system for high throughput and low latency[C/OL]//11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). Broomfield, CO: USENIX Association, 2014: 49-65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[14] DPDK. Data plane development kit.[EB/OL]. 2024[2024-05-12]. https://www.dpdk.org/.

[15] SPDK. Storage Performance Development Kit[EB/OL]. 2024[2024-05-12]. https://spdk.io/.

[16] Intel. Intel data direct I/O technology (Intel DDIO): A primer.[EB/OL]. 2024[2024-05-12]. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf.

[17] Nayak R J, Chavda J B. Comparison of Accelerator Coherency Port (ACP) and High Performance Port (HP) for Data Transfer in DDR Memory Using Xilinx ZYNQ SoC[C]//Satapathy S C, Joshi A. Information and Communication Technology for Intelligent Systems (ICTIS 2017) - Volume 1. Cham: Springer International Publishing, 2018: 94-102.

[18] Chen X, Yu G, Cheng H. Approach to external events of real-time operating system based on polling[C/OL]//2010 Second International Conference on Computer Modeling and Simulation: Vol. 1. 2010: 459-462. DOI: 10.1109/ICCMS.2010.312.

[19] Guan H, Dong Y, Tian K, et al. Sr-iov based network interrupt-free virtualization with event based polling[J/OL]. IEEE Journal on Selected Areas in Communications, 2013, 31(12): 2596-2609. DOI: 10.1109/JSAC.2013.131202.

[20] Le Moal D. I/o latency optimization with polling[C]//Vault Linux Storage and Filesystems Conference. 2017.

[21] Stephen Bates H. Linux Optimizations for Low Latency Block Devices | SNIA[EB/OL]. 2017 [2024-05-12]. https://www.snia.org/educational-library/linux-optimizations-low-latency-block-devices-2017.

[22] AlQahtani S A. A novel hybrid scheme of interrupt enabling - disabling and polling (edp) for high-speed computer networks[C/OL]//2007 International Symposium on Communications and Information Technologies. 2007: 341-345. DOI: 10.1109/ISCIT.2007.4392042.

[23] Gao G, Hum H, Theobald K, et al. Polling watchdog: Combining polling and interrupts for efficient message handling[C/OL]//23rd Annual International Symposium on Computer Architecture (ISCA'96). 1996: 179-179. DOI: 10.1145/232973.232992.

[24] Lee G, Shin S, Jeong J. Efficient hybrid polling for ultra-low latency storage devices[J/OL]. J. Syst. Archit., 2022, 122(C). https://doi.org/10.1016/j.sysarc.2021.102338.

[25] Langendoen K, Romein J, Bhoedjang R, et al. Integrating polling, interrupts, and thread management[C/OL]//Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96). 1996: 13-22. DOI: 10.1109/FMPC.1996.558057.

[26] Yamasaki N. Responsive multithreaded processor for distributed real-time processing[C/OL]//International Workshop on Innovative Architecture for Future Generation High Performance Processors and Systems (IWIA'06). 2006: 44-56. DOI: 10.1109/IWIAS.2006.36.

[27] Suito K, Fujii K, Matsutani H, et al. Dependable responsive multithreaded processor for distributed real-time systems[C/OL]//2012 IEEE COOL Chips XV. 2012: 1-3. DOI: 10.1109/COOLChips.2012.6216589.

[28] Wada R, Yamasaki N. Fast interrupt handling scheme by using interrupt wake-up mechanism [C/OL]//2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW). 2019: 109-114. DOI: 10.1109/CANDARW.2019.00027.

[29] Lopez T A, Yamasaki N. Prioritized asynchronous calls for parallel processing on responsive multithreaded processor[C/OL]//2022 Tenth International Symposium on Computing and Networking (CANDAR). 2022: 46-55. DOI: 10.1109/CANDAR57322.2022.00014.

[30] Dodiu E, Gaitan V, Graur A. Custom designed cpu architecture based on a hardware scheduler and independent pipeline registers - architecture description[C]//2012 Proceedings of the 35th International Convention MIPRO. 2012: 859-864.

[31] Gaitan N C, Gaitan V G, Moisuc E E C. Improving interrupt handling in the nmpra[C/OL]//2014 International Conference on Development and Application Systems (DAS). 2014: 11-15. DOI: 10.1109/DAAS.2014.6842419.

[32] ZAGAN I, GAITAN N C, GAITAN V G. An approach of nmpra architecture using hardware implemented support for event prioritization and treating[J/OL]. International Journal of Advanced Computer Science and Applications, 2017, 8(2). http://dx.doi.org/10.14569/IJACSA.2017.080206.

[33] Salah K. To coalesce or not to coalesce[J/OL]. AEU - International Journal of Electronics and Communications, 2007, 61(4): 215-225. https://www.sciencedirect.com/science/article/pii/S1434841106000063X. DOI: https://doi.org/10.1016/j.aeue.2006.04.007.

[34] Ahmad I, Gulati A, Mashtizadeh A, et al. Improving performance with interrupt coalescing for virtual machine disk io in vmware esx server[J]. VMware Inc., Palo Alto, CA, 2009, 94304.

[35] Ahmad I, Gulati A, Mashtizadeh A J. vic: Interrupt coalescing for virtual machine storage device io[C/OL]//USENIX Annual Technical Conference. 2011. https://api.semanticscholar.org/CorpusID:16541915.

[36] Dong Y, Xu D, Zhang Y, et al. Optimizing network i/o virtualization with efficient interrupt coalescing and virtual receive side scaling[C/OL]//2011 IEEE International Conference on Cluster Computing. 2011: 26-34. DOI: 10.1109/CLUSTER.2011.12.

[37] Guan H, Dong Y, Ma R, et al. Performance enhancement for network i/o virtualization with efficient interrupt coalescing and virtual receive-side scaling[J/OL]. IEEE Transactions on Parallel and Distributed Systems, 2013, 24(6): 1118-1128. DOI: 10.1109/TPDS.2012.339.

[38] Tai A, et al. Optimizing storage performance with calibrated interrupts[C]//15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). New York, NY, USA: Association for Computing Machinery, 2021: 129-145.

[39] Gomes T, Garcia P, Salgado F, et al. Task-aware interrupt controller: Priority space unification in real-time systems[J/OL]. IEEE Embedded Systems Letters, 2015, 7(1): 27-30. DOI: 10.1109/LES.2015.2397604.

[40] Erwin J D, Jensen E D. Interrupt processing with queued content-addressable memories[C/OL]//AFIPS '70 (Fall): Proceedings of the November 17-19, 1970, Fall Joint Computer Conference. New York, NY, USA: Association for Computing Machinery, 1970: 621–627. https://doi.org/10.1145/1478462.1478553.

[41] Scheler F, Hofer W, Oechslein B, et al. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system[C/OL]//Henkel J, Parameswaran S. Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2009, Grenoble, France, October 11-16, 2009. ACM, 2009: 167-174. https://doi.org/10.1145/1629395.1629419.

[42] Wierman A, Zwart B. Is tail-optimal scheduling possible?[J]. Operations research, 2012, 60 (5): 1249-1257.

[43] Prekas G, Kogias M, Bugnion E. Zygos: Achieving low tail latency for microsecond-scale networked tasks[C]//Proceedings of the 26th Symposium on Operating Systems Principles. 2017: 325-341.

[44] Iyer R, Unal M, Kogias M, et al. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling[C]//Proceedings of the 29th Symposium on Operating Systems Principles. 2023: 466-481.

[45] Kaffes K, Chong T, Humphries J T, et al. Shinjuku: Preemptive scheduling for {μsecond-scale} tail latency[C]//16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). 2019: 345-360.

[46] Ousterhout A, Fried J, Behrens J, et al. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads[C]//16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). 2019: 361-378.

[47] Jia Y, Tian K, You Y, et al. Skyloft: A general high-efficient scheduling framework in user space[C]//Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. 2024: 265-279.

[48] Mehta S. x86 User Interrupts support[EB/OL]. 2021[2024-09-02]. https://lwn.net/Articles/869140/.

[49] Luo X, Shen W, Mu S, et al. DepFast: Orchestrating code of quorum systems[C/OL]//2022 USENIX Annual Technical Conference (USENIX ATC 22). Carlsbad, CA: USENIX Association, 2022: 557-574. https://www.usenix.org/conference/atc22/presentation/luo.

[50] von Behren R, Condit J, Zhou F, et al. Capriccio: scalable threads for internet services[C/OL]// SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2003: 268–281. https://doi.org/10.1145/945445.945471.

[51] Zhang I, Raybuck A, Patel P, et al. The demikernel datapath os architecture for microsecond-scale datacenter systems[C]//Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021: 195-211.

[52] Embassy. Modern embedded framework, using Rust and async.[EB/OL]. 2023[2024-01-04]. https://github.com/embassy-rs/embassy.

[53] Dion. Async Rust vs RTOS showdown! - Blog - Tweede golf[EB/OL]. 2022[2025-03-04]. https://tweedegolf.nl/en/blog/65/async-rust-vs-rtos-showdown.

[54] Wikipedia. 冯诺伊曼结构[EB/OL]. 2024[2025-03-05]. https://en.wikipedia.org/wiki/Von_Neumann_architecture.

[55] Saltzer J H. Traffic control in a multiplexed computer system[D]. Massachusetts Institute of Technology, 1966.

[56] Dijkstra E W. The structure of the “the”-multiprogramming system[J]. Communications of the ACM, 1968, 11(5): 341-346.

[57] Ritchie D M, Thompson K. The unix time-sharing system[J]. Communications of the ACM, 1974, 17(7): 365-375.

[58] Marlin C D. Coroutines: a programming methodology, a language design and an implementation: No. 95[M]. Springer Science & Business Media, 1980.

[59] Moura A L D, Ierusalimschy R. Revisiting coroutines[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2009, 31(2): 1-31.

[60] 葛文博. 众核环境下嵌入式实时操作系统中调度处理的研究[D]. 中国电子科技集团公司电子科学研究院, 2023.

[61] Lin J, Chen Y, Gao S, et al. Fast core scheduling with userspace process abstraction[C/OL]// SOSP '24: Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2024: 280 – 295. https://doi.org/10.1145/3694715.3695976.

[62] RISC-V. RISC-V Ratified Specifications[EB/OL]. 2025[2025-03-06]. https://riscv.org/specifications/ratified/.

[63] Pinto S, Garlati C. User mode interrupts[Z]. 2019.

致　谢

衷心感谢我的导师向勇老师对本人的精心指导，他的言传身教将使我终身受益。

感谢实验室的同学廖东海，以及各位学弟。

感谢清华大学的各位老师，以及为我提供帮助的同学。

感谢我的母亲，因为她的支持和鼓励，我能够顺利完成学业。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

个人简历、在学期间完成的相关学术成果

个人简历

1999 年 01 月 09 日出生于湖南湘潭县。

2022 年 9 月考入清华大学计算机系攻读计算机科学与技术专业硕士至今。

在学期间完成的相关学术成果

学术论文：

[1] Zhao F, Liao D, Wu J, et al. Cops: A coroutine-based priority scheduling framework perceived by the operating system[C]//2024 International Conference on Computer and Information Technology. 2024.

指导教师评语

论文提出了……

答辩委员会决议书

论文提出了……

论文取得的主要创新性成果包括：

1. ……

2. ……

3. ……

论文工作表明作者在 ××××× 具有 ××××× 知识，具有 ×××× 能力，论文 ××××，答辩 ××××。

答辩委员会表决，（× 票/一致）同意通过论文答辩，并建议授予 ×××（姓名）×××（门类）学博士/硕士学位。