基于 Rust 协程的任务调度器设计与应用

选题背景及其意义

在当今时代，云应用程序（例如网络搜索、社交网络、电子商务等）在日常生活中发挥着越来越更重要的作用。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。因此，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，且这些需求将会逐渐变得更加苛刻。因此，如何构建出低时延的服务已经成为一个重要的研究领域。

近年来，随着超低时延网络出现，网络传输对于构建低时延服务的影响越来越小，且相关的网络协议栈也被卸载到智能网卡上，因此，在 CPU 上运行的任务调度模块逐渐成为影响时延的主要因素。当前的主流思想是以线程这一高成本的抽象来表达任务，尽管可以通过线程来构建低时延的服务，但协程为实现这个需求带来了新的思路。当前，多种现代编程语言（C#、Go、Javascript、Python 和 Rust 等）先后引入协程的概念，提供异步函数的抽象，允许程序员通过关键字创建异步函数，通过允许异步函数的执行重叠来更好的利用计算资源并提高程序执行效率。

另外，任务的行为对于构建低时延的服务也存在着影响。任务在执行期间，存在着多种因素导致发生上下文切换（例如任务之间切换、任务主动调用系统调用或任务被中断抢占），这些上下文切换除了产生寄存器保存/恢复、调度器、以及处理器流水线刷新等直接开销外，还会产生缓存丢失等间接开销。协程具备上下文切换开销小的特点，使用协程来表达任务可以减少任务切换带来的开销，但还需要尽可能减少其他的上下文切换开销才能构建出低时延的服务。

因此，设计出以协程为单位的任务调度器，并且利用协程尽可能减少上下文切换对于构建低时延的服务具有非常重要的意义。

国内外研究现状

近年来，出现了许多低时延的服务，包括 FaRM、Memcached、MICA、RAMCloud 和 Redis 等，这些服务采用了各种新型技术来满足低时延的要求。优化技术大体可以分为以下两类：

1. 从任务调度方面进行优化。一种优化技术是保留额外的处理器核心用于满足高负载情况下的低时延要求（例如 shenango，它的运行时为每个应用程序分配一定数量的保证核心和突发核心），尽管现代的处理器能够在一个物理核心上提供多个硬件线程，但这会造成一定程度上的资源浪费问题；另一种优化技术则是通过抢占式调度来满足低时延的要求。例如，Shinjuku 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；而 Concord 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；但上述两种优化的技术存在着一定程度的

资源浪费，Arachne 针对这些资源浪费进行了优化，实现了一种新的用户级线程，它的运行时能够准确的知道分配给应用程序的处理器核心，并根据负载进行动态调整。

2. 从减少任务执行期间上下文切换开销进行优化。FlexSC 以及 Cassyopia 分别通过对系统调用进行异步化改造和批处理来减少上下文切换；Userspace Bypass 则将连续系统调用之间的用户态代码转移到内核中执行，从而减少了由系统调用造成的上下文切换；Kernel Bypass 技术则是将某些硬件设备直接暴露给应用程序，应用程序通过轮询来操作设备，减少了进出内核需要的上下文开销以及内核中的复杂的处理流程开销；Unikernel 与 Kernel Bypass 相似，但它更加精简，应用程序与驱动处于单地址空间中，系统调用直接转化为函数调用，消除了由系统调用带来的上下文切换开销；Privbox 则将内核中的系统调用代码使用沙箱技术安全的暴露给应用程序，尽管它没有减少由系统调用带来的上下文切换次数，但每次应用程序在使用系统调用提供的服务时不需要进入到内核，减小了单次系统调用的开销。

此外，还可将上述优化技术组合来取得更大程度的优化，例如，Shinjuku 以及 Concord 同时针对上述两方面进行了优化，它们的调度线程除了通知工作线程进行抢占外，还负责轮询内核旁路设备。但上述的优化技术并没有涉及到表达任务的模型优化，任务模型仍然使用线程这个高成本抽象，一方面，用线程定义的任务模型，其切换成本高，需要保存/恢复完整的硬件线程抽象（堆栈、通用寄存器、程序计数器等）。另一方面，线程难以和异步机制相结合，将线程模型与事件驱动模型相结合将会导致系统过于复杂（例如 29,30）。而协程作为一种轻量级的任务抽象，它具有比线程更小的切换开销，且它能与异步机制进行有效结合。例如，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，向应用程序提供了异步 I/O，从而满足低时延的要求。此外，基于 Rust 协程实现的为嵌入式设备上运行的异步程序生成框架 Embassy 在处理设备中断方面取得了令人激动的效果。

主要研究内容

通过目前基于 Rust 协程的应用以及低时延服务的发展现状来看，将协程用于构建低时延服务是可行的。Rust 协程必须通过运行时的调度才能运行，因此实现高效的运行时是研究工作中的第一个重难点；然而，使用协程来表达任务后，面临着一个非常严重的问题，即阻塞在 IO 上的协程，需要高效的唤醒机制。若单独使用一个协程进行轮询，由该协程唤醒其他阻塞协程，则这个协程将退化成线程。若使用中断机制进行唤醒，由于中断是一种抢占式的强制控制流转移，这需要保存和恢复大量的寄存器，这两种唤醒机制悖离了使用协程的初衷。因此设计一种高效的协程唤醒机制是研究工作中的第二个重难点；此外，如果在协程运行期间，由于其他因素导致发生了频繁的上下文切换，这同样是不能容忍的，故而如何利用协程减少上下文切换开销是研究工作中涉及到的第三个重难点。

基于上述重难点，本文将从以下几个方面展开研究：

1. 协程调度机制研究。研究协程调度问题的关键在于其协作式的调度如何实现快速响应的目标。因此，需要分析协程、线程、进程之间的关系，分析协程上下文切换开销以及执行时间的统计特性，从其特性出发，用协作式调度来拟合抢占式调度。

2. 协程唤醒机制研究。分析当前协程唤醒相关方法及其时间开销，分析协程与中断机制之间的鸿沟，将中断机制与协程相适配，实现一种高效的协程唤醒机制。
3. 减小上下文切换次数以及单次开销。分析任务本身的工作负载，统计分析各个操作的时间开销，对任务中涉及到上下文切换的操作，使用协程以及异步机制进行改造，期望减小上下文切换带来的开销，减小任务本身需要的时间开销，从而构建出低时延的服务。

研究方案及难点

在基于 Rust 协程的任务调度器中，协作式调度是其显著特点，但由于请求具有随机性和不可预测性，因此需要一定程度的抢占调度才能够保证及时对请求作出响应。

首先，我们应当分析目前使用协作式调度拟合抢占式调度的一些研究成果及相关方法，分析两者之间可能存在的平衡关系以及落实到协程上的具体契合点。

其次，针对协程唤醒机制，我们需要解决何时唤醒协程的问题。传统的通过轮询或中断的方式都与使用协程的初衷相悖。而由于 Rust 协程被实现为有限状态机，其中跨越暂停点的变量被保存在不会移动的内存块上，让协程恢复执行所需要的信息仅仅是协程本身的数据结构指针。因此，唤醒协程的操作被极大的简化为将协程的数据结构指针保存到调度器的就绪队列中。因此，我们可以设计出一种专门的硬件设备，它接收中断，并负责唤醒与中断相关的协程。

再次，我们可以利用硬件来帮助实现异步系统调用，减小由于系统调用造成的上下文切换开销。

最后，我们在模拟环境以及现实应用负载下，使用设计的基于 Rust 协程的任务调度器，构建系统原型以及上层应用，测试相关的性能指标，验证调度器能够有效构建出低时延服务。

预期成果和可能的创新点

在对协程调度、唤醒与中断机制的分析、总结基础上，通过设计新的调度机制与唤醒机制，弥补协作式调度高响应延迟的缺陷，得到近似于抢占式调度的响应时延。并且利用协程减小上下文切换开销，构建出微秒级的低时延服务。

其中，可能的创新点有：一种新的硬件设备，将协程调度、唤醒与中断处理从 CPU 上卸载；一种基于硬件的系统调用加速机制，通过硬件转发系统调用，实现系统调用异步化以及批处理。

论文工作计划

论文工作的总体时间安排：

1. 2023年 11 ～ 12 月，开题准备工作，查找基于 Rust 协程的任务调度器相关资料，了解国内外的发展动态，细化设计方案，理清实施思路。
2. 2024年 1 ～ 6 月，搭建实验环境，构建系统原型。
3. 2024年 7 ～ 8 月，针对原型系统，构建上层应用，测试相关性能数据。

4. 2024年 9 ～ 12 月，完成论文写作。

参考文献

1. FlexSC: flexible system call scheduling with exception-less system calls
2. Userspace Bypass: Accelerating Syscall-intensive Applications
3. Privbox: Faster System Calls Through Sandboxed Privileged Execution
4. Cassyopia: Compiler Assisted System Optimization
5. Unikernels: library operating systems for the cloud
6. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling
7. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems
8. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads
9. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency
10. Arachne: Core-Aware Thread Management
11. Scheduler activations: effective kernel support for the user-level management of parallelism
12. Shared-stack cooperative threads
13. Capriccio: scalable threads for internet services
14. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)
15. Context switch overheads for Linux on ARM platforms
16. A case against (most) context switches
17. Quantifying the cost of context switch
18. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications
19. Compiler support for lightweight context switching
20. Balancing register pressure and context-switching delays in ASTI systems
21. When poll is more energy efficient than interrupt
22. Lightweight Preemptible Functions
23. Revisiting coroutines
24. CAT: Context Aware Tracing for Rust Asynchronous Programs
25. A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems
26. Process-Based Simulation with Stackless Coroutines
27. DepFast: Orchestrating Code of Quorum Systems
28. M3X: Autonomous Accelerators via Context-Enabled Fast-Path Communication
29. Cooperative Task Management without Manual Stack Management
30. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives