



# 华中科技大学

## 全国大学生计算机系统能力大赛 操作系统功能赛初赛报告

### 基于 eBPF 的沙箱运行优化机制

队长姓名 张纳百川

队员姓名 张纳百川、韦芳宇、王佳明

指导教师 黄卓、吴松

2024 年 5 月 31 日

# 目录

摘要.....	6
1. 介绍.....	6
1.1 项目背景.....	8
1.2 项目目标.....	9
2. 研究内容.....	10
2.1 Serverless.....	10
2.1.1 Serverless 技术特点.....	10
2.1.2 搭建平台的层次结构.....	11
2.2 WebAssembly.....	12
2.2.1 WebAssembly 技术的特点.....	12
2.2.2 Wasm 容器的应用优势.....	13
2.2.3 WasmEdge.....	14
2.3 调度策略的优化.....	14
2.3.1 调度策略的优化需求.....	14
2.3.2 Linux 中的调度策略.....	15
2.3.3 内核调度程序的实现.....	15
2.3.4 内核调度程序的部署.....	15
3. 设计.....	16
3.1 设计目标.....	16
3.2 eBPF.....	17
3.2.1 什么是 eBPF.....	17
3.2.2 eBPF 的优势.....	18
3.2.3 云环境下的 eBPF.....	18
3.3 ghOSt 的设计实现.....	19

3.3.1 内核到代理的通信.....	21
3.3.2 代理到内核的通信.....	23
3.3.3 集中式调度器.....	25
3.3.4 故障隔离和动态升级.....	27
<b>4. 实验效果 .....</b>	<b>28</b>
4.1 gh0St 所需开销和扩展性分析 .....	28
4.1.1 gh0St 文件大小 .....	28
实验设置.....	28
消息传递开销.....	29
本地调度.....	29
远程调度.....	29
全局代理的扩展性.....	29
4.2 与定制的集中式调度器的比较.....	30
4.2.1 比较的系统对象.....	31
4.2.2 单一工作负载比较.....	31
4.2.3 多工作负载比较.....	32
4.3 基于谷歌 Snap 的评估.....	32
4.3.1 当前工作线程的调度方式.....	33
4.3.2 实验设置.....	33
4.3.3 测试工作负载.....	33
4.3.4 gh0St 策略.....	33
4.3.5 尾延迟比较.....	34
4.4 基于谷歌 Search 的方案评估.....	35
4.4.1 测试工作负载.....	35
4.4.2 实验设置.....	36
4.4.3 gh0St 策略.....	36
4.4.4 CFS vs. gh0St.....	37
4.4.5 尾延迟.....	37

# 图目录

图 1.1 ghOSt 概略图.....	8
图 2.1 WASM VS Docker .....	13
图 3.1 Kubernetes 中的 eBPF .....	19
图 3.2 ghOSt 概览.....	20
图 3.3 ghOSt 策略管理.....	20
图 3.4 线程调度 .....	24
图 3.5 一个简化的全局代理的例子 .....	26
图 4.1 全局代理的可扩展性 .....	30
图 4.2 ghOSt 的抢占式策略.....	31
图 4.3 消息延迟 .....	35
图 4.4 使用 CFS 和 ghOSt 调度的基准结果 .....	37

表目录

表 3.1 ghOSt 消息和系统调用..... 22

表 4.1 ghOSt 微基准测试..... 29

## 摘要

本项目介绍了一种在用户空间调整内核调度策略的基础架构。该架构的设计旨在支持数据中心工作负载和平台快速发展的需求。

改进调度决策可以大幅提高工作负载的吞吐量、尾部延迟、可扩展性和安全性。然而，内核调度策略很难在大范围内有效实施、测试和部署。最近的研究表明，在定制的操作系统中，优化调度策略可以显著提升应用的运行性能。然而，如何在现有操作系统上实现这一目标并不容易。原因在于，在应用粒度上重新部署操作系统是不切实际的（特别是在多租户环境中）。本项目在 Linux 环境的用户空间实现了对内核进程调度策略的自定义。具体而言，本项目提供了状态封装、通信和操作机制，可以在用户空间代理中定制化内核的进程调度策略；用户可以使用任何语言来开发和优化策略，而无需重新启动主机。最后，我们展示了项目成果在实际工作负载（Google Snap 和 Google Search）上的性能和相关操作开销，证明了其有效性。

## 1. 介绍

CPU 调度在应用性能和安全性方面发挥着重要作用。为特定工作负载类型量身定制策略，可以大幅改善延迟、吞吐量、硬/软实时特性、能效、缓存干扰和安全性等关键指标。

例如，Shinjuku 请求调度程序优化了高度分散的工作负载（混合了短请求和长请求的工作负载），将请求尾部延迟和吞吐量提高了一个数量级。针对虚拟机工作负载的 Tableau 调度器显示，在多租户场景下，吞吐量提高了 1.6 倍，延迟提高了 17 倍。Caladan 调度器重点解决了前台低延迟应用程序和后台计算密集型应用程序之间的资源干扰问题，将网络请求尾部延迟提高了 11000 倍。

为了缓解最近出现的硬件漏洞，运行多租户主机的云平台必须快速部

署新的内核隔离策略，隔离应用程序之间的共享处理器状态。设计、实施和部署跨大型网络的新调度策略是一项艰巨的任务。这要求开发人员设计出能够平衡众多应用程序特定性能要求的策略。进一步地，调度策略的实现过程必须符合当前极其复杂的内核架构，否则一点微小错误会导致整个系统崩溃。而且，即使成功升级，升级的破坏性也会带来主机和应用程序停机的服务成本。这就造成了风险最小化与升级更新之间的关键性矛盾。

之前通过设计用户空间解决方案来提高内核性能和降低复杂性的尝试都存在明显不足，或需要对应用程序的实现进行大量修改，或需要专用资源才能实现高响应，抑或者需要对内核进行特定的应用程序修改。尽管 Linux 支持多种调度实现，但为每个应用程序量身定制、维护和部署不同的机制，在数据中心内是不切实际的。我们的目标包含两个：1）确定需要对内核进行哪些修改，以便在稳定的内核接口上实现用户空间的大量优化（和实验）；2）支持不断发展和变化的异构硬件类型，例如，AWS Nitro 和 DPU 等计算外设以及 GPU 和 TPU 等领域专用加速器是新型的紧密耦合计算设备，其存在完全超出了经典调度程序的范畴，因此亟需一种新的调度策略。

在本项目中，我们介绍了一种用于本地操作系统线程的调度程序 ghOSt。其目标是从根本上改变调度策略的设计、实施和部署方式。ghOSt 在实现 进程级调度的同时，还提供了用户空间开发的灵活性和部署的便捷性。ghOSt 为用户空间软件提供了抽象和接口，以制定复杂的调度策略，从 per-CPU 模型到全系统（集中式）模型。更重要的是，ghOSt 将内核调度机制与策略定义分离开来，便于用户自定义。

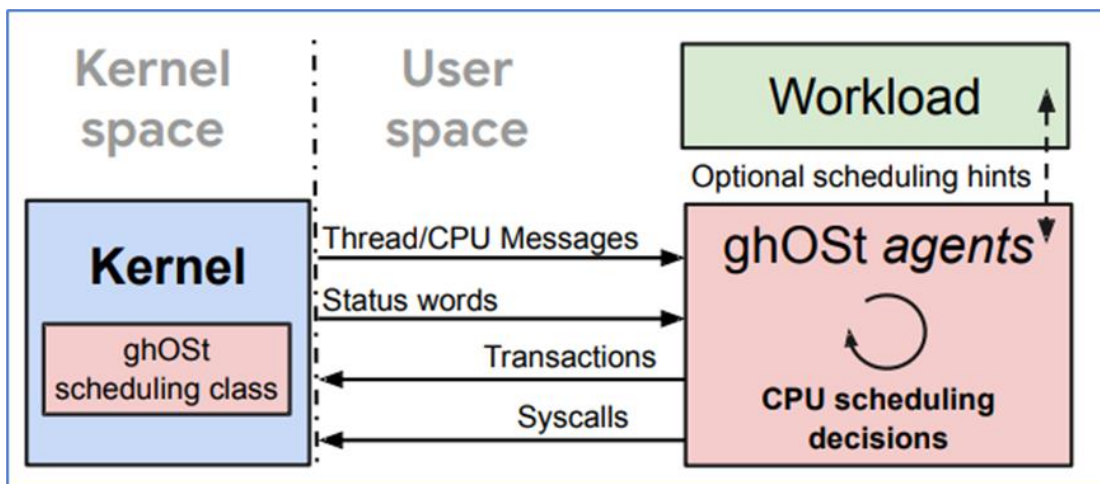


图 1.1 ghOSt 概略图

通过调度本地线程，ghOSt 可支持现有应用程序，而无需进行任何更改。在 ghOSt（图 1）中，调度策略逻辑运行在正常的 Linux 进程中，这些进程被称为代理，它们通过 ghOSt API 与内核交互。内核通过异步路径中的消息队列向代理通报所有托管线程的状态变化（如线程创建/阻塞/唤醒）。ghOSt 支持多策略并发执行、故障隔离以及为不同应用分配 CPU 资源。实验表明，ghOSt 的开销很小，从消息传递的 265 纳秒、上下文切换到代理的几百纳秒到调度线程的 888 纳秒，ghOSt 的调度开销仅略高于现有的内核调度器。

## 1.1 项目背景

在服务器无感知计算（Serverless Computing）中，现有的主流技术是利用沙箱容器技术，如 AWS Firecracker 或者阿里云沙箱容器，来实现强隔离的安全执行环境，但是也带来更大的资源消耗。虽然现在阿里云沙箱容器经过优化可以实现 300ms 的冷启动速度，但是还无法满足函数毫秒级的启动要求。目前需要通过的调度策略，预留一定的空闲实例才可以满足，但是这样也引入了更多的资源消耗。

**WebAssembly 技术：**WASM 是一个新的 W3C 规范，是一个通用、开放、安全的底层虚拟机抽象。其可以轻松实现毫秒级冷启动时间和极低的资源消耗。同时 WASM 字节码比原生机器码有更高的安全级别。此外，WASI 实现了细粒度基于能力的安全模型，遵循最小权限原则。在执行过程中，



WASI 应用只能访问由依赖注入指明的确切资源集，这种方式与传统粗粒度的操作系统级隔离相比，进一步收敛了安全攻击面。正因如此，WASM 得到了 Serverless、IoT/边缘计算等社区的广泛关注。Fastly、Cloudflare 等厂商相继发布了基于 WebAssembly 技术实现了更加轻量化的 Serverless 服务。

**内核调度机制：**当前服务器无感知计算平台会同时运行大量函数实例来完成用户的业务需求，主机操作系统内核对函数实例仅仅支持完全公平的调度策略（completely Fair Scheduler, CFS）。然而，不同负载具有不同的运行特征，适合不同的调度策略，需要云厂商根据当前负载情况自适应调整平台负载运行策略。当前开发者调整内核调度机制均需要重新编译和替换操作系统内核，效率非常低。为此，亟需在多实例环境中，支持内核调度器动态扩展调度策略，以满足不同负载的性能需求。此外，沙箱运行过程，其虚拟指令与主机操作系统的机器指令存在语义鸿沟，使得最终执行时操作繁复低效，导致性能比不使用沙箱的情况慢 2-3 倍。为此，需要针对负载运行特征，优化底层指令的执行，提高运行速度。

**eBPF 技术：**eBPF 技术,在用户程序触发系统调用时注入预定义的代码，动态聚合分析操作行为，并对非法行为进行控制。该技术是目前热门的内核可编程机制，提供多语言无侵入的观测能力，兼具高性能高扩展特性，生态发展很好，未来前景广大，但当前其主要集中在网络性能优化和安全领域。我们创新性将其应用到操作系统内核的任务调度器和函数负载的底层执行中，为服务器无感知计算负载提供更加灵活高效的调度策略和执行优化，有利于后期的技术落地和社区推广。

出于对以上技术背景以及实际问题的探究，本赛题拟基于 eBPF 技术提高 wasm 沙箱在云环境中的运行效率，降低运行时间，提高应用端到端的响应时间。

## 1.2 项目目标

本项目拟基于 eBPF 技术提高 wasm 沙箱在云环境中的运行效率，降低运行时间，提高应用端到端的响应时间。

目标一：测试当前负载运行过程存在的性能问题，并分析其根本原因

目标二：基于 eBPF 实现对多沙箱实例的细粒度调度机制，保证不同沙箱都具有较高的运行效率

目标三：基于 eBPF 实现对单沙箱实例的任务运行加速机制，保证单个沙箱在运行过程中的高效率

## 2. 研究内容

### 2.1 Serverless

#### 2.1.1 Serverless 技术特点

Serverless 是一种云原生开发模型，允许开发人员构建和运行应用程序而无需管理服务器。开发人员可以将其代码简单的打包部署在无服务器，最大化利用云的弹性可扩展性构建自己的应用程序。

Serverless 技术主要有如下优点：

- 自动弹性扩展：根据实际请求自动横向扩展，动态分配资源，确保应用具有高可扩展性
- 用户无感知：开发者无需关心底层的基础设施和运维工作，平台无缝集成其他云服务，从而让开发者专注于代码编写和业务逻辑的实现，大幅度提高开发效率
- 为价值付费：从用户角度，避免传统服务器和基础设施、运维工作的成本，同时精确计算实际使用的资源付费；从平台开发者角度，细粒度的资源分配避免服务器的闲置浪费

Serverless 计算具有广阔的前景，然而其也存在安全和性能方面的问题。在 Serverless 计算中，现有的主流技术是利用沙箱容器技术，如 AWS Firecracker 或者阿里云沙箱容器，来实现强隔离的安全执行环境，但是也带来更大的资源消耗。虽然现在阿里云沙箱容器经过优化可以实现 300ms 的冷启动速度，但是还无法满足函数毫秒级的启动要求。目前需要通过的调

度策略，预留一定的空闲实例才可以满足，但是这样也引入了更多的资源消耗。

### 2.1.2 搭建平台的层次结构

平台搭建拟利用 Knative、Kubernetes 等开源框架进行部署，初步采用 containerd 作为 High-Level 运行时进行镜像管理、容器创建，并采用轻量级的 Wasm 虚拟机作为沙箱环境以减小沙箱初始化开销。并将 eBPF 应用到操作系统内核的任务调度器和函数负载的底层执行中，为服务器无感知计算负载提供灵活高效的调度策略和执行优化。

#### **Kubernetes:**

Kubernetes 也称为 K8s，是用于自动部署、扩缩和管理容器化应用程序的开源系统。Kubernetes 在容器编排领域占据主导地位，已成为云原生技术生态系统的事实标准。

#### **Knative:**

Knative 是一个开源的、用于构建、部署和管理现代 Serverless 工作负载的平台。它提供了一套构建、部署和运行容器化应用程序的工具和组件，使开发人员能够更轻松地构建云原生应用。Knative 的设计目标是简化开发人员对于 Serverless 应用的开发流程，同时提供自动的弹性扩缩容和事件驱动的特性。

#### **容器运行时、Containerd:**

容器运行时负责管理和运行容器的软件组件，它负责启动、停止、管理容器的生命周期，并提供容器与宿主操作系统之间的隔离。

Containerd 是一个开源的 High-Level 容器运行时，旨在提供一个通用的容器管理平台，支持容器的创建、运行、存储和销毁等操作。它借鉴了 Docker 最初的容器运行时代码，并且已成为 CNCF 的一个项目。Containerd 构建在 OCI 标准之上，可以与 Kubernetes 配合使用，为容器化应用提供一致的运行环境。

Containerd 的稳定性、可靠性和广泛的支持使其成为许多生产环境中首

选的容器运行时。我们项目中选择 containerd 也是出于对其稳定性、可靠性和与其他容器生态系统的集成性的考量。

## 2.2 WebAssembly

WebAssembly（简称 Wasm）是一种低级的、跨平台的字节码格式，旨在提供高性能的跨浏览器的执行环境。它最初由 Mozilla、Google、Microsoft 和 Apple 等公司共同推动开发，于 2015 年宣布，并于 2017 年成为 W3C 的 Web 标准。

WebAssembly 的设计目标是在 Web 浏览器中提供更高性能的执行环境，使得诸如游戏、多媒体应用等高性能要求的应用程序能够在 Web 上运行得更加流畅。与 JavaScript 相比，WebAssembly 更加接近底层硬件，因此可以实现更高的性能。Wasm 除了应用在浏览器中，也可以应用到 out-of-web 环境中。通过 WASI（WebAssembly System Interface，Wasm 操作系统接口）标准，Wasm 可以直接与操作系统打交道。通过已经在各种环境实现了 WASI 标准的虚拟机，我们就可以将 wasm 用在嵌入式、IOT 物联网以及甚至云，AI 和区块链等特殊的领域和场景中。

### 2.2.1 WebAssembly 技术的特点

WebAssembly 技术的特点包括：

1. 跨平台性：WebAssembly 可以在任何支持的平台上运行，包括浏览器、服务器、物联网设备等。
2. 安全性：由于 WebAssembly 的代码是在沙箱环境中运行的，因此具有良好的安全性，可以防止恶意代码对系统的破坏。
3. 高性能：WebAssembly 的字节码可以被高效地解析和执行，因此具有很高的性能，适用于需要高性能的应用场景。
4. 语言无关性：WebAssembly 不依赖于任何特定的编程语言，因此可以由各种编程语言编译生成。

WebAssembly 的应用领域包括 Web 前端开发、服务器端应用、游戏开

发、物联网等，它正在逐渐成为一个重要的跨平台执行环境，为开发者提供了更多的选择和灵活性。

2.2.2 Wasm 容器的应用优势

Aspect	WASM	Container
Size	Several MBs	Several 10s or 100s of MBs
Startup time	milliseconds	seconds
Performance Speed	Almost native speed	Far away from native speed
Runs in Web browser	Yes	No
Cross-platform/Portability	High	Specific to CPU architectures
Standards	Both the W3C and OCI	OCI
System Interactions	Use WASI to access underlying OS	Container itself contains an OS and file system

图 2.1 WASM VS Docker

Wasm 容器的应用相较于其他传统容器有以下优势：

1. 轻量化和高效性能：Wasm 容器将应用程序打包成 WebAssembly 模块，这些模块通常比传统的容器镜像更轻量级，因为它们只包含应用程序的逻辑和依赖，而不包括操作系统或其他系统级组件。这种轻量化的设计使得 Wasm 容器在部署和执行时具有更高的效率和速度。
2. 跨平台和环境隔离：Wasm 容器可以在任何支持 WebAssembly 的平台上运行，无需依赖特定的操作系统或硬件架构。这种跨平台的特性使得应用程序可以更容易地在不同的环境中进行部署和迁移，并且能够实现与底层操作系统和硬件的隔离，提高了应用程序的安全性和稳定性。
3. 快速启动和低资源消耗：由于 Wasm 容器不包含完整的操作系统，因此其启动速度通常比传统的容器快得多，并且在资源消耗方面更为节省。这使得 Wasm 容器特别适用于需要快速启动和低资源消耗的场景，例如边缘计算、物联网等领域。
4. 与现有生态系统的兼容性：Wasm 容器可以与现有的容器生态系统（如 Kubernetes、Docker 等）无缝集成，通过现有的容器编排工具进行部

署和管理。这使得开发者能够利用已有的工具和流程，更容易地采用 Wasm 容器技术，并将其应用于实际生产环境中。

综上所述，Wasm 容器通过轻量化、跨平台、环境隔离、快速启动和与现有生态系统的兼容性等优势，为应用程序的部署和执行提供了更高效、更灵活和更安全的解决方案，使其在各种场景下都具有广泛的应用前景。

### 2.2.3 WasmEdge

WasmEdge 是一个开源的 WebAssembly 运行时（Wasm runtime），专为边缘计算场景设计。它旨在提供高性能、安全可靠的运行环境，支持在边缘设备上运行基于 WebAssembly 的应用程序和服务。WasmEdge 的设计注重轻量化和高效性能。它采用了一系列优化技术，包括 AOT（Ahead-of-Time）和 JIT（Just-in-Time）编译，以及基于 Wasm 的沙箱机制，以实现高性能和安全性的平衡。这使得 WasmEdge 成为了在资源受限的边缘设备上运行复杂应用的理想选择。

WasmEdge 支持多种编程语言，并提供了丰富的扩展性和灵活性，使开发者能够轻松地构建和部署各种类型的边缘应用程序，包括物联网（IoT）、边缘 AI 和边缘计算等领域的应用。此外，WasmEdge 还提供了丰富的工具链和文档资源，以帮助开发者更好地理解和使用该平台，从而加速边缘计算应用的开发和部署过程。

## 2.3 调度策略的优化

### 2.3.1 调度策略的优化需求

大型云提供商和用户都有需求去部署新的调度策略，以优化在日益复杂的硬件拓扑结构上运行的关键工作负载的性能，并通过在单独的物理内核上隔离不受信任的线程，提供针对新硬件漏洞的保护。不同的工作负载具有不同的资源需求以及运行特性，如果为特定工作负载类型量身定制相应的调度策略，则可以大幅改善延迟、吞吐量、硬/软实时特性、能效、缓存干扰和安全性等关键指标。例如，（1）Shinjuku 请求调度程序优化了高度分散的工作负载（混合了短请求和长请求的工作负载），从而将请求尾部

延迟和吞吐量提高了一个数量级；（2）在多租户场景下，通过使用专门应用于虚拟机工作负载的 Tableau 调度器，可以将吞吐量提高至 1.6 倍，将延迟提高 17 倍；（3）Caladan 调度器重点解决了前台低延迟应用程序与后台计算密集型应用程序之间的资源干扰问题，从而将网络请求尾部延迟提高了 11000 倍之多。

### 2.3.2 Linux 中的调度策略

Linux 支持通过调度类实施多种策略。类可按优先级排序：在抢占式调度策略下，使用优先级较高的类调度的线程会抢先使用优先级较低的类调度的线程。为了优化特定应用程序的性能，开发人员可以修改调度类，以更好地满足应用程序的需求。例如，确定应用程序线程的优先级以减少网络延迟，或使用实时策略调度应用程序线程以满足数据库查询的最后期限。因此原则上，云提供商或应用程序开发人员也可以创建一个全新的类，针对特定服务来进行优化。然而，在内核中实施和维护这些策略不是一件易事，这就导致许多开发人员转而使用现有的通用策略，如 Linux 中的完全公平的调度策略（completely Fair Scheduler, CFS）。因此，Linux 中现有的类在设计上尽可能支持更多的用例，但同时由于其仍具有一定的局限性，通过使用这些过于通用的类来优化高性能应用程序，以最大限度地提高硬件的使用效率，依旧是一项具有挑战性的工作。

### 2.3.3 内核调度程序的实现

当开发人员开始设计新的内核调度程序时，他们会发现这些调度程序很难实现、测试和调试。调度程序通常用低级语言（C 语言和汇编语言）编写，无法利用现有的库，也无法使用当下流行的调试工具进行调试。同时，调度程序依赖于复杂的同步原语，包括原子操作、RCU、任务抢占和中断，并与之交互，这使得开发和调试变得更加困难。长期维护也是一项挑战。Linux 很少合并新的调度类，尤其不可能接受经过高度调整的非通用调度程序。

### 2.3.4 内核调度程序的部署

**部署调度程序较为困难。**要对调度策略进行部署更改，就需要在大范围内部署新的内核，这对云提供商来说极具挑战性。要在计算节点上安装新的内核，云提供商必须迁移或终止计算节点的相应工作，让计算节点进入静态，安装新的系统软件，让系统守护进程重新初始化，最后等待新分配的应用程序并准备好再次提供服务。而且，新调度程序的初始部署仅仅是个开始。在部署第一个试用版本后，云提供商将频繁地进行修改，以消除漏洞和调整性能。例如，谷歌曾指出，磁盘服务器上有一个调度程序漏洞，而这个漏洞从被发现到被修复为止，导致了总计数百万美元的收入损失。

**用户空间的局限性。**用户空间调度的是用户线程。其实质是将  $M$  个用户线程分配到  $N$  个本地线程上。而这是不可预测的。具体地，用户空间调度程序可以控制哪个用户线程在某个本地线程上运行，但它无法控制该本地线程何时实际运行或在哪个 CPU 上运行。更严重的是，内核还会对持有用户级锁的线程重新编排。这极大限制了用户空间调度程序的有效性。通常地，开发人员会分配专用 CPU 运行特定本地线程，从而保证隐式控制用户线程的运行。然而，由于专用 CPU 不能与其他应用程序共享，因此这种方案会造成严重资源浪费。

**为每个工作负载定制调度程序/操作系统是不切实际的。**现有在内核调度策略上的工作通常针对的是特定工作负载。这些系统可以为其目标负载提供了较好的性能，但它们是系统内核深度耦合，难以适用于其它工作负载。例如，Shenango 对系统内核进行了深度修改，并只能为网络密集型负载提供单线程调度策略，如果添加额外策略还需要大量修改。

## 3. 设计

### 3.1 设计目标

ghOSt 的目标是为调度程序的设计、优化和部署引入一种新的范式。我们在设计 ghOSt 时考虑了以下要求：

- **策略应易于实施和测试。**随着新的工作负载、内核和异构硬件进入数据



中心，新的调度策略内核补丁应该要易于部署。

- **调度策略的精细化。**云提供商和用户需要针对各种优化目标设计相应的调度策略，包括以吞吐量为导向的工作负载、延迟敏感型工作负载、实时性工作负载等。

- **多样化调度需求。**Linux 内核中的调度策略是以 CPU 为粒度进行调度决策。最近的研究表明，通过集中轮询的方式调度线程，可以有效改善延迟敏感型负载的性能。ESXi NUMA 和 Minos 等系统也证明了以比单个 CPU 更粗的粒度（如每个 NUMA 节点）进行协调的好处。但这种更粗粒度的方式很难在现有 Linux 内核架构，因为现有调度架构是在以 CPU 为粒度的基础上实现的。

- **支持多种调度策略的混部。**随着计算节点性能的不断增强，单台服务器上运行的负载和租户数量越来越多。自定义调度策略必须支持多个调度策略并行执行。

- **无中断更新和故障隔离。**在大型设备上升级操作系统会导致昂贵的停机时间。升级时必须迁移或重新启动分配给每台主机的工作，主机本身也必须重新启动。这个耗时的过程会显著降低服务质量。因此，自定义调度策略必须在升级节点的同时保持服务正常运行，新策略的部署、更新、回滚甚至崩溃，都不会导致机器重启。

## 3.2 eBPF

### 3.2.1 什么是 eBPF

eBPF 是一项革命性的技术，起源于 Linux 内核，它可以在特权上下文中（如操作系统内核）运行沙盒程序。它用于安全有效地扩展内核的功能，而无需通过更改内核源代码或加载内核模块的方式来实现。

通过允许在操作系统中运行沙盒程序的方式，应用程序开发人员可以运行 eBPF 程序，以便在运行时向操作系统添加额外的功能。然后在 JIT 编译器和验证引擎的帮助下，操作系统确保它像本地编译的程序一样具备安

全性和执行效率。

### 3.2.2 eBPF 的优势

- **独立：**使用 eBPF 提供的新功能不需要修改 linux 内核。
- **安全：**eBPF 提供验证机制，它可以确保只有在运行安全的情况下才加载 eBPF 程序。
- **热插拔：**eBPF 程序可以动态载入内核或从内核中移除。不需要重启机器，也不需要预先关闭可能触发 eBPF 功能的进程。
- **高效：**eBPF 程序是一种非常有效的指令添加方式。一旦加载并经过 JIT 编译，程序就能在 CPU 上以本地机器指令的形式运行。此外，在处理每个事件时，无需在内核和用户空间之间转换。
- **灵活：**对于性能跟踪和安全可观察性而言，eBPF 的另一个优势是可以在内核中过滤相关事件，然后再将其发送到用户空间。如今的 eBPF 程序可以收集系统中各种事件的信息，并使用复杂、定制的程序过滤器，只向用户空间发送相关的信息子集。

### 3.2.3 云环境下的 eBPF

在容器中运行的应用程序，如果运行在同一台机器上，就会共享同一个内核。在 Kubernetes 环境中，这意味着某个节点上一个 pod 中的所有容器都在使用同一个内核。

如图 3.1 所示，当我们使用 eBPF 程序对该内核进行调试时，该节点上的所有容器化工作负载对这些 eBPF 程序都是可见的。

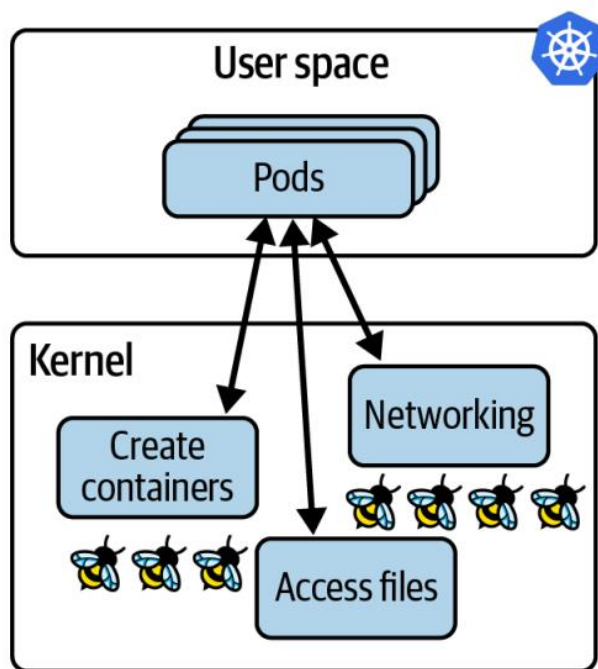


图 3.1 Kubernetes 中的 eBPF

使用 eBPF 工具，赋予了节点上进程的可见性，再加上动态加载 eBPF 程序的能力，给予我们在云原生计算中的超能力：

- 我们不需要改变应用程序，甚至不需要改变其配置方式，就能使用 eBPF 工具对其进行检测。
- 只要将其加载到内核中并连接到事件，eBPF 程序就能开始观察已有的应用程序进程。

### 3.3 ghOSt 的设计实现

图 3.2 总结了 ghOSt 的设计。用户空间代理做出调度决策，并指导内核如何在 CPU 上调度本机线程。ghOSt 的内核端被实现为一个调度类，类似于常用的 CFS 类。该调度类为用户空间代码提供了丰富的 API 来定义任意调度策略。为了帮助代理做出调度决策，内核通过消息和状态字向代理公开线程状态。然后，代理通过事务和系统调用来指示内核的调度决策。

接下来，我们将描述 ghOSt 如何使能 per-CPU 调度和集中式调度。per-CPU 调度通常使用负载均衡和作业窃取来平均整个系统的负载。而集中式调度，则有一个全局实体不断地监控整个系统，并为其权限下的所有线程

和 CPU 进行决策。

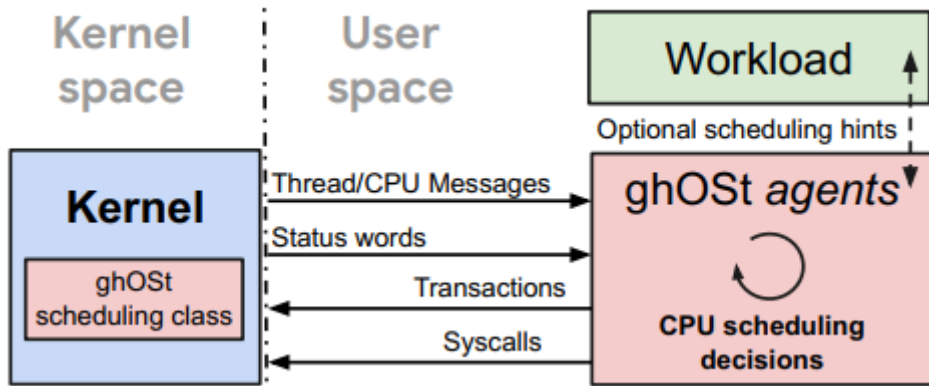


图 3.2 ghOSt 概览

值得说明的是，本节中提到的所有线程都是原生线程。我们将逻辑执行单元称为 CPU。例如，我们假设一台具有 56 个物理内核和 112 个逻辑内核（超线程）的机器具有 112 个 CPU。ghOSt 使用 enclaves(安全区)在一台机器上支持多个当前策略。一个系统可以按 CPU 粒度划分为多个独立的安全区，每个安全区都运行自己的策略，如图 3.3 所示。从调度的角度来看，安全区是孤立的。当在一台机器上运行不同的工作负载，不同的 NUMA 节点、不同的 AMD-CCX 都是不同的安全区。安全区也有助于隔离故障，限制代理崩溃对其所属安全区的损害。

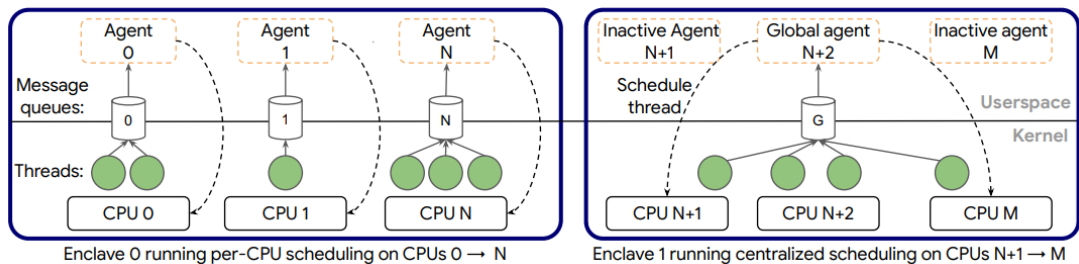


图 3.3 ghOSt 策略管理

为了实现我们的设计目标，调度策略逻辑是在用户空间代理中实现的。代理可以用任何语言编写，并通过标准工具进行调试，使其更易于实现和测试。为了实现容错和隔离，如果一个或多个代理崩溃，系统将回退到默认的调度程序，如 CFS。当一个新的 ghOSt 用户空间代理启动时，该机器仍然完全正常工作——要么是最后一个已知的稳定版本，要么是一个带有修复程序的新版本。

由于具有崩溃恢复特性，更新调度策略相当于重新启动用户空间代理，而无需重新启动计算机。此属性可用于各种硬件和工作负载的实验和快速策略定制。开发人员可以对策略进行调整，只需重新启动代理即可。

无论调度方式是 per-CPU 还是集中式，ghOSt 管理的每个 CPU 都有一个本地代理，如图 3.3 所示。在 per-CPU 的情况下，每个代理负责自己 CPU 的线程调度决策。在集中式情况下，单个全局代理负责调度安全区中的所有 CPU。每个代理都是在 Linux pthread 中实现的，所有代理都属于同一个用户空间进程。

### 3.3.1 内核到代理的通信

为了让代理为其权限下的线程做出调度决策，内核必须向代理公开线程状态。一种方法是将现有的内核数据结构内存映射到用户空间中，例如 `task_structs`，这样代理就可以检查它们来推断线程状态。然而，这些数据结构的可用性和格式因内核和内核版本而异，所以用户空间策略的实现不得不与内核版本紧密耦合。另一种方法是通过 `sysfs` 文件以一个 `/proc/pid/...` 方式公开线程状态。然而，文件系统 API 不适合快速路径操作，因为块设备的 `open/read/fseek` 等操作过于缓慢和复杂（例如，需要错误处理和数据解析），所以难以支持微秒级的响应。

对此，我们需要一个内核-用户空间接口，它既快速又不依赖于线程的底层内核实现。受分布式系统的启发，我们使用 `messages`(消息)作为一种有效而简单的解决方案。在 ghOSt 中，内核使用表 3.1 中列出的消息通知用户空间代理线程状态的更改。例如，如果一个线程曾被阻塞并且现在可以运行，内核会发布一条 `thread_WAKEUP` 消息。此外，内核还会用 `timer_TICK` 消息通知代理定时器的滴答声。为了帮助代理验证他们是根据最新的状态做出决定的，消息也有序列号，后面将对此进行解释。

<u>Messages</u>	<u>Syscalls</u>
THREAD_CREATED	AGENT_INIT()
THREAD_BLOCKED	START_GHOST()
THREAD_PREEMPTED	TXN_CREATE()
THREAD_YIELD	TXNS_COMMIT()
THREAD_DEAD	TXNS_RECALL()
THREAD_WAKEUP	CREATE_QUEUE()
THREAD_AFFINITY	DESTROY_QUEUE()
TIMER_TICK	ASSOCIATE_QUEUE()
	CONFIG_QUEUE_WAKEUP()

表 3.1 ghOSt 消息和系统调用

消息通过消息队列传递给代理。在 ghOSt 下调度的每个线程都被分配一个单独的队列，关于该线程状态更改的所有消息都被传递到该队列。在 per-CPU 的例子中，每个线程都被分配到一个队列，该队列对应于它要运行的 CPU（图 3.3，左）。在集中式示例中，所有线程都被分配到全局队列（图 3.3，右）。CPU 事件的消息（如 TIMER\_TICK）被路由到与 CPU 相关联的代理线程的队列。

尽管有很多方法可以实现队列，但 ghOSt 选择在共享内存中使用自定义队列来有效地处理代理唤醒（如下所述）。我们认为现有的队列机制对 ghOSt 来说是不够的，因为它们只存在于特定的内核版本中。例如，BPF 系统通过 BPF 环形总线将 BPF 事件传递到用户空间，而最新版本的 Linux 也通过 io\_uring 将异步 I/O 消息传递到用户区域。这两种都是快速无锁定的环形分离器，可同步消费者/生产者访问。但是，较旧的 Linux 内核和其他操作系统不支持它们。

在 ghOSt 的 enclave(安全区)初始化之后，安全区中只有一个默认队列。代理进程可以使用 create/destroy\_QUEUE() API 创建/销毁队列。添加到 ghOSt 的线程被隐式地分配用于将消息发布到默认队列。代理可以通过 ASSOCIATE\_QUEUE() 更改该分配。

当消息被产生到队列中时，队列可以被选择性地配置为唤醒一个或多个代理。代理可以通过 CONFIG\_QUEUE\_wakeup() 配置唤醒行为。在每个

CPU 的例子中，每个队列只与一个 CPU 关联，并配置为唤醒相应的代理。在集中式示例中，队列由全局代理持续轮询，因此唤醒是多余的，可不进行配置。代理唤醒使用标准内核机制来唤醒被阻塞的线程。这包括识别要唤醒的代理线程，将其标记为可运行，可选地向目标 CPU 发送中断以触发重新调度，以及执行到代理线程的上下文切换。

代理根据通过消息观察到的系统状态进行操作。然而，当代理做出调度决策时，新消息可能会到达队列，这可能会改变该决策。与集中式调度示例相比，单 CPU 示例的这一挑战略有不同（请参见 4.2.2 和 4.2.3 节）。无论哪种方式，我们都使用代理/线程序列号来解决这一挑战：每个代理都有一个序列号  $A_{seq}$ ，每当消息被发布到与该代理相关联的队列时，该序列号就会递增。我们在 4.2.2 中解释了我们在每个 CPU 的例子中使用  $A_{seq}$ 。每个线程都有一个序列号  $T_{seq}$ 。当每个线程发布新的状态更改消息 MT 时，该序列号都会递增。当管理器弹出队列时，它会接收到一条消息及其相应的序列号： $(M_T, T_{seq})$ 。我们在 4.2.3 节中解释了如何将  $T_{seq}$  用于集中式调度示例。ghOSt 允许代理通过状态字有效地轮询有关线程和 CPU 状态的辅助信息，将其映射到代理的地址空间。为了简洁起见，我们只讨论了使用状态词向代理公开序列号  $A_{seq}$  和  $T_{seq}$ 。当内核更新线程或代理程序的序列号时，它也会更新相应的状态字。然后，代理可以从共享映射中的状态字中读取序列号。

### 3.3.2 代理到内核的通信

我们现在描述代理如何指示内核下一个调度哪个线程。

代理通过提交事务将调度决策发送到内核。代理必须能够调度其本地 CPU（per-CPU 情况）以及其他远程 CPU（集中式情况）。提交机制必须快速，并扩展到数百个核心。对于 per-CPU 的例子，理论上一个系统调用接口就足够了。对于集中式情况，代理需要有效地向多个 CPU 发送调度请求，然后检查这些请求是否成功。因此，共享存储器接口更合适。此外，使用共享内存中的事务作为调度接口将允许在未来将调度决策卸载到能够访问该内存的外部设备。



受事务内存和数据库系统的启发，我们设计了自己的事务 API。其支持具有原子语义的快速分布式提交操作。具体地，代理使用 TXN\_CREATEE

( ) 辅助函数在共享内存中打开一个新事务。代理既写入要调度线程的 ID，也写入要调度线程的 CPU 的 ID。在每个 CPU 的示例中，每个代理只调度自己的 CPU。当事务被填充时，代理通过 TXNS\_COMMIT (系统调用，启动提交过程并触发内核上下文切换) 将其提交给内核。简化示例如图 3.4 所示。

```
1 void Agent::PerCpuSchedule() {
2     DrainMessageQueue(); // Read messages from queue
3     Thread *next = runqueue_.Dequeue();
4     if (next == nullptr) return; // Runqueue empty.
5     // Schedule thread:
6     Transaction *txn = TXN_CREATE(next->tid, my_cpu);
7     TXNS_COMMIT({txn});
8     if (txn->status != TXN_COMMITTED) {
9         // Txn failed. Move thread to end of runqueue.
10        runqueue_.Enqueue(next);
11        return;
12    }
13    // The schedule has succeeded for `next`.
14 }
```

图 3.4 线程调度

在集中式调度示例中，为了允许 ghOST 扩展到每秒数百个 cpu 和数十万个事务，我们必须降低系统调用的昂贵成本。我们通过引入组提交来摊销交易成本。组提交也减少了发送到其他 cpu 的中断的数量，类似于 Caladan。代理通过将所有事务传递给 TXNS\_COMMIT() 系统调用来提交多个事务。这个系统调用将昂贵的开销分摊到几个事务上。最重要的是，它通过使用大多数处理器中存在的批处理中断功能来分摊发送中断的开销。内核没有依次发送多个中断(每个事务一个)，而是向远程 cpu 发送单个批处理中断，从而节省了大量开销。

在每个 CPU 的示例中，提交事务的代理将其 CPU 交给它正在调度的目标线程。在代理运行时发送到队列的消息不会引起唤醒，因为代理已经在运行。但是，队列中的新消息可能来自更高优先级的线程，并且如果代理



意识到这一点，将会影响调度决策。代理只有在下一次唤醒时才有机会检查该消息，这已经错过了正确的时机。接下来，我们通过 `per-cpu` 示例的序列号来解释如何解决这个问题。具体的，我们使用代理序列号  $A_{seq}$  来解决这个问题。代理通过检查代理线程的状态词来轮询其变量  $A_{seq}$ 。当将新消息发布到与代理相关联的队列时，将增加变量  $A_{seq}$ 。操作顺序为：1) 读变量  $A_{seq}$ ；2) 从队列中读取消息；3) 做出调度决策；4) 将事务发送给 `TXNS_COMMIT()`。如果与事务一起发送的变量  $A_{seq}$  比内核观察到的当前变量  $A_{seq}$  更旧(即向代理的队列中发布了新消息)，则该事务被认为是“stale”，并将以 `ESTALE` 错误失败。然后代理耗尽其队列以检索更新的消息并重复该过程。`ghOSt` 的消息传递和组调度会导致高达  $5\mu s$ ；在集中式调度模型中，一个线程可能会等待整个集中式调度循环，直到一个调度决策被代表它提交。对此，`ghOSt` 允许通过自定义 BPF 程序来降低这一开销。自定义程序由代理附加到内核的 `pick_next_task()` 函数。当 CPU 空闲并且代理还没有发出事务时，BPF 程序发出自己的事务，选择一个线程在该 CPU 上运行。BPF 程序通过共享内存窗口与代理通信，进入代理的地址空间。代理使用 BPF 基础结构来调度 `cpu` 上的线程。BPF 程序使用 `libbpf` 嵌入到代理二进制文件中，因此该程序本质上是代理本身的扩展。

### 3.3.3 集中式调度器

现在我们解释 `ghOSt` 是如何构建集中式调度策略。

对于集中式调度，`ghOSt` 需要有一个全局代理轮询每个消息队列，并为管理的所有 `cpu` 做出调度决策。如果指定的 CPU 已经运行了一个线程，则事务将抢占前一个线程，以支持新线程。调度器代码的简化示例如图 3.5 所示。

```

1 void GlobalAgent::CentralizedSchedule() {
2     DrainMessageQueue();
3     map<Cpu, Thread*> assignments;
4     // GetIdleCPUs() will return all available CPUs.
5     for (const Cpu& cpu : GetIdleCPUs()) {
6         Thread *next = runqueue_.Dequeue();
7         if (next == nullptr) break; // Runqueue empty.
8         assignments[cpu] = next; // Run `next` on `cpu`.
9     }
10    // Now send transactions for all assignments:
11    vector<Transaction*> txns = Schedule(assignments);
12    for (const Transaction *txn : txns) {
13        // Check if `txn` committed successfully.
14        if (txn->status != TXN_COMMITTED) {
15            Thread *next = GetThreadFromTID(txn->tid);
16            // Transaction failed. Re-enqueue.
17            runqueue_.Enqueue(next);
18            continue;
19        } } }
20 vector<Transaction*> GlobalAgent::Schedule(
21     const map<Cpu, Thread*>& assignments) {
22     vector<Transaction*> txns;
23     for (const auto& [cpu, next] : assignments) {
24         Transaction *txn = TXN_CREATE(next->tid, cpu);
25         txns.push_back(txn);
26     }
27     TXNS_COMMIT(txns);
28     return txns;
29 }

```

图 3.5 一个简化的全局代理的例子

为了支持微秒级的调度，全局代理必须连续运行，因为任何抢占都会导致调度延迟。为了防止由高优先级内核调度类触发的全局代理抢占，ghOSt 为所有代理分配了高优先级内核。换句话说，机器中的其他线程(无论是 ghOSt 还是非 ghOSt)都不能抢占代理线程。ghOSt 通过以下方式维护系统的稳定性。所有非活动代理立即放弃，腾出它们的 `cpu`。当一个非ghost线程需要在全局代理的 CPU 上运行时，全局代理执行一个“热切换”到另一个 CPU 上的一个不活跃的代理。例如，如果内核 CFS 调度器试图调度一个在  $CPU_{global}$  上的线程，全局代理将首先找到一个空闲的 CPU ( $CPU_{idle}$ )，然后唤醒在  $CPU_{idle}$  上不活动的代理作为新的全局代理。当  $CPU_{idle}$  运行全局

代理后，旧的全局代理就会退出，从而允许 CFS 线程在  $CPU_{global}$  上运行。

在某些情况下，全局代理可能对线程的状态存在不一致。例如，线程  $T$  可能会发布一个 `THREAD_WAKEUP` 消息。全局代理收到此消息，并决定在  $CPU_f$  上调度  $T$ 。同时，系统中的某些实体调用了 `sched_setaffinity()`，导致一个 `THREAD_AFFINITY` 消息，禁止  $T$  在  $CPU_f$  上运行。对比，我们用线程序列号来解决这个问题。具体地，每个排队的消息  $MT$  都用线程序列号  $T_{seq}$  标记为  $(M_T, T_{seq})$ 。当代理为线程  $T$  提交事务时，它会发送事务以及它所知道的  $T$  的最新序列号  $T_{seq}$ 。当内核接收到事务时，它验证  $T_{seq}$  相对于事务中的线程是否是最新的。否则，事务失败并出现 `ESTALE` 错误。

### 3.3.4 故障隔离和动态升级

ghOSt 的设计目标之一是在现有系统上易于使用。因此，ghOSt 管理的线程要能与系统中的其他线程进行良好的交互。对此，我们通过在内核调度类层次结构中为 ghOSt 的内核调度程序类分配一个比默认调度程序类(通常是 CFS)更低的优先级来实现这个目标。其目的是使得系统中的大多数线程可以抢占 ghOSt 线程。ghOSt 线程的抢占将导致创建 `THREAD_PREEMPT` 消息，触发相关代理(在不同的高优先级调度类中运行)做出调度决策。由于更新调度策略(即代理)不需要重新启动内核或应用程序，ghOSt 支持快速部署。许多生产服务可能需要几分钟到几小时才能启动，特别是填充内存缓存时。类似地，我们希望尽量减少客户机虚拟机的中断。这些长时间运行的应用程序在计划中的代理更新或计划外的代理崩溃期间继续正确运行。ghOSt 通过以下两种方式实现动态升级:(a)在保持安全区基础设施完好无损的情况下替换代理，或(b)摧毁安全区并从头开始。具体而言，用户空间代码可以查询代理是否附加到安全区上。为了升级代理，我们同时运行新旧代理;新代理将阻塞，直到旧代理崩溃或退出并且不再连接。新的代理从内核中提取安全区中所有线程的状态并恢复调度。如果此进程失败，内核或用户空间代码都可以破坏安全区。破坏安全区会杀死该安全区中的所有代理，但保持系统中其他安全区的完整，并自动将被破坏安全区中的所有线程移回 CFS。此时，线程仍然正常工作，但由 CFS

而不是 ghOSt 调度。

需要说明的是，ghOSt 或任何其他内核调度器中的调度错误都会导致系统级的影响。例如，ghOSt 线程在持有内核互斥锁时可能会被抢占，如果它没有被调度太长时间，它可能会暂时使其他线程(包括 CFS 或其他 ghOSt enclaves 中的线程)停滞。类似地，如果没有调度关键线程(如垃圾收集器和 I/O 轮询器)，机器将逐渐停止工作。作为一种安全机制，ghOSt 会自动摧毁行为不端的安全区。例如，当内核检测到代理没有用户在用户可配置的毫秒数内调度可运行线程时，它将销毁一个安全区。

## 4.实验效果

我们对 ghOSt 的评估集中在三个问题上：(a) ghOSt 特定操作的开销是多少 (§4.1)；(b) 使用 ghOSt 实现的调度策略与先前的工作（如 Shinjuku）相比表现如何 (§4.2)；以及 (c) ghOSt 是否是大规模和低延迟生产工作负载的可行解决方案，包括 Google Snap (§4.3)、Google Search (§4.4)。

### 4.1 ghOSt 所需开销和扩展性分析

#### 4.1.1 ghOSt 文件大小

表 2 显示了 ghOSt 的代码行数 (LOC)，其内核代码比 CFS 少 40%。用户空间的策略可以很短（几百行代码），因为它们利用了用户空间库中的通用函数。这反映了在高级语言中定义策略的一个优势：更灵活的抽象，使得复杂性可以集中在调度决策上。

#### 实验设置：

实验平台为 Linux 内核 4.15，机器配置有双插槽 Intel Xeon Platinum 8173M @ 2GHz、每个插槽 28 核、每核 2 逻辑核心。表 3 总结了 ghOSt 的基本操作开销。

1. Message Delivery to Local Agent	725 ns
2. Message Delivery to Global Agent	265 ns
3. Local Schedule (1 txn)	888 ns
<b>Remote Schedule (1 txn for 1 CPU)</b>	
4. Agent Overhead	668 ns
5. Target CPU Overhead	1064 ns
6. End-to-End Latency	1772 ns
<b>Group Remote Schedule (10 txns for 10 CPUs)</b>	
7. Agent Overhead	3964 ns
8. Target CPU Overhead	1821 ns
9. End-to-End Latency	5688 ns
10. Syscall Overhead	72 ns
11. pthread Minimal Context Switch Overhead	410 ns
12. CFS Context Switch Overhead	599 ns

表 4.1 ghOSt 微基准测试

## 消息传递开销（第 1-2 行）：

在 CPU 传递到本地代理的操作包括将消息添加到队列、上下文切换到本地代理和出队消息。此开销（725 ns）主要由上下文切换（410 ns）决定。在集中式示例中，传递到全局代理（265 ns）包括将消息添加到队列和在全局代理内出队消息。

## 本地调度（第 3 行）：

提交事务并在本地 CPU 上执行上下文切换，直到目标线程运行。此操作开销（888 ns）略高于 CFS 上下文切换开销（599 ns）。

## 远程调度（第 4-9 行）：

在集中式调度模型中，代理端提交事务并发送跨处理器中断（IPI）。目标 CPU 处理 IPI 并执行上下文切换。代理的开销（668 ns）设定了每个代理的理论最大吞吐量为  $109/668 = 1.5\text{M}$  每秒调度线程。将 10 个不同 CPU 的事务分组可以通过摊薄 IPI 开销将理论最大值提高到  $10 * 109/3964 = 2.52\text{M}$  每秒调度线程。根据这些数据，单个代理理论上可以为 100 CPU 服务器的每个 CPU 每秒调度大约 25,200 个线程。

## 全局代理的扩展性（图 4.1）：

为了展示全局代理的扩展性，我们分析了一种简单的轮转策略。该策略管理 FIFO 运行队列中的所有线程，并在 CPU 空闲时立即在其上调度线程。代理在每次提交时尽可能多地分组事务。我们在默认的微基准测试机器上运行了实验，该机器使用 Skylake 处理器，以及一个带有 Haswell 处理器（每个插槽 18 个物理核心，每个核心两个逻辑核心，2.3 GHz）的双插槽机器。结果如图 4.1 所示。两条线条的模式相似，我们在图中注释了 Skylake 线条：陡峭的上升阶段 ❶ 表明随着可调度工作的 CPU 增多，全局代理每秒调度的事务数量也增加。当我们将全局代理与执行工作的 ghOSt 线程共同定位在同一个物理核心上时，出现了下降 ❷。超线程争夺物理核心管道中的资源，从而降低了全局代理的性能。最后，下降 ❸ 来自于全局代理调度远程插槽中的 CPU。调度这些 CPU 需要进行内存操作并在 NUMA 插槽之间发送中断，这会产生更高的开销。

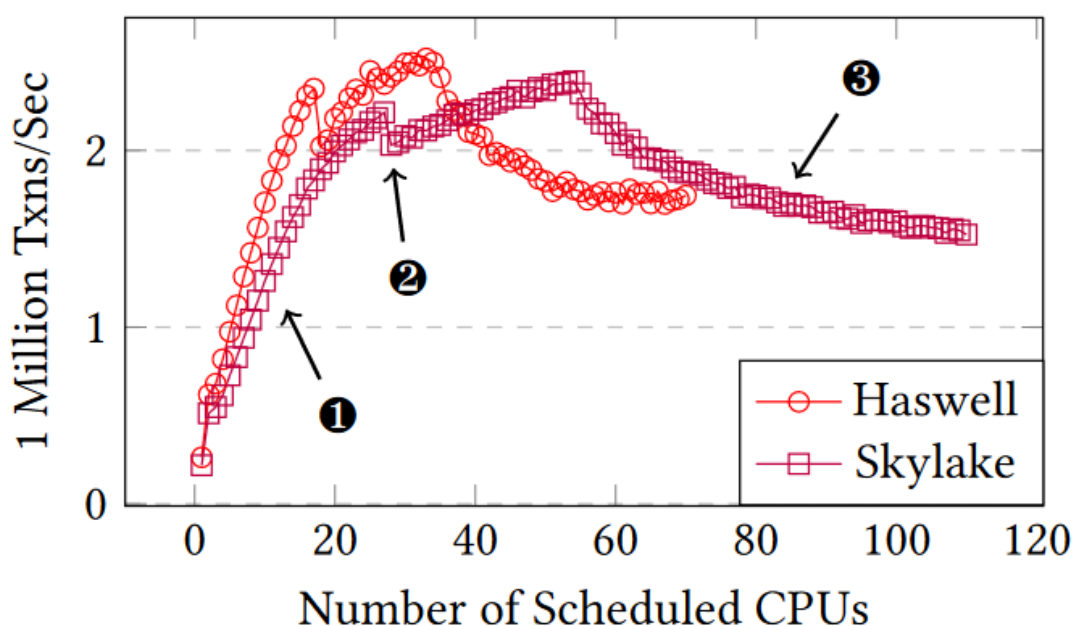


图 4.1 全局代理的可扩展性

## 4.2 与定制的集中式调度器的比较

我们现在将 ghOSt 与当前的工作进行比较，该系统使用集中式策略来调度微秒级工作负载。实验在双插槽 Intel Xeon CPU E5-2658 的一个插槽上运行（每个插槽 12 个核心，每个插槽 24 个逻辑核心，2.2 GHz）。

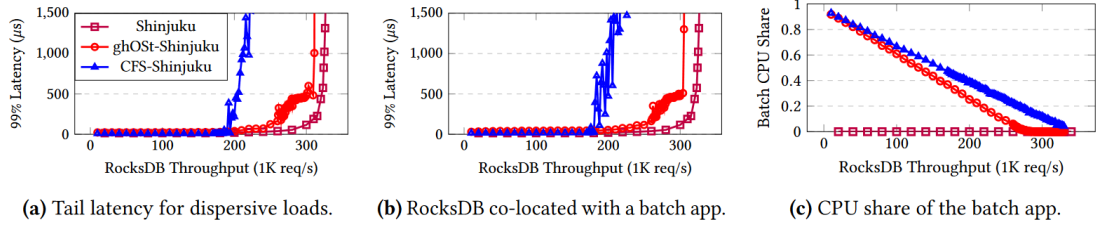


图 4.2 ghOSt 的抢占式策略

### 4.2.1 比较的系统对象

我们比较了 Shinjuku 调度方法的三种实现。我们在所有系统中使用一个物理核心生成负载。由于 Shinjuku 系统的 Dune 驱动程序无法编译，它将运行在 Linux 4.4 上。其他对比对象（ghOSt-Shinjuku 和 CFS-Shinjuku）运行在应用了我们 ghOSt 补丁的 Linux 4.15 上。

1. 原始 Shinjuku 系统：它使用 20 个自旋工作线程绑定到 20 个不同的超线程，以及一个运行在专用物理核心上的自旋调度线程。自旋线程防止其他线程在其 CPU 上运行（图 6c）。调度线程将管理到达的请求，并把它们分配给工作线程。每个请求最多运行一个时间片，然后被抢占并添加到 FIFO 的末尾。
2. 在 ghOSt 上实现的 Shinjuku 调度策略：我们在用户空间使用集中式模型实现了 Shinjuku 调度策略。ghOSt-Shinjuku 全局代理最初在自己的物理核心上启动，但可以自由移动 (§3.3)。我们维护了一个由 200 个工作线程组成的池，由负载生成器分配请求。全局代理维护一个可运行工作线程的 FIFO 队列，并将它们调度到其余的 20 个 CPU 上。值得注意的是，ghOSt 允许系统中的其他负载使用任何空闲的 CPU，如图 4.2b-c 所示。
3. 非抢占版本的 Shinjuku：为了全面对比，我们还在 Linux CFS 上实现了一个非抢占版本的 Shinjuku。这个 CFS-Shinjuku 版本没有使用虚拟化功能和用于抢占的中断功能。

### 4.2.2 单一工作负载比较

与 Shinjuku 论文中一样，我们生成的工作负载每个请求包含一个对内存中 RocksDB 键值存储的 GET 查询（约 6 微秒）并进行少量处理。我们分

配的处理时间如下：99.5% 的请求为 4 微秒，0.5% 的请求为 10 毫秒。每个工作线程的分配时间片在强制抢占并返回到 FIFO 之前为 30 微秒。CFS-Shinjuku 是非抢占式的，因此所有请求都运行到完成。我们的结果如图 4.2a 所示。即使 ghOSt 的 Shinjuku 策略实现代码行数比定制的 Shinjuku 少 82%，ghOSt 在尾延迟方面与 Shinjuku 差不多。ghOSt 在高负载下的尾延迟略高于 Shinjuku，并且其最大吞吐量在 Shinjuku 的 5% 以内。这一差异反映了 ghOSt 为每个请求调度线程的额外开销。由于缺乏抢占，CFS-Shinjuku 比其他两个系统的吞吐量少约 30%。

### 4.2.3 多工作负载比较

当 RocksDB 负载较低时，使用空闲的计算资源来服务低优先级的批处理应用程序。原始的 Shinjuku 系统调度请求，不能管理任何其他本地线程。图 6c 显示，当我们将批处理应用程序与 Shinjuku 管理的 RocksDB 工作负载共同运行时，即使 RockDB 负载较低，批处理应用程序也无法获得任何 CPU 资源。为了高效地同时运行低延迟和批处理工作负载，可以考虑使用面向线程的集中式调度系统，如 Shenango。当应用程序负载较轻时，调度器将剩余的 CPU 周期分配给批处理应用程序。然而，Shenango 不适合处理执行时间不固定的请求，因此 RocksDB 工作负载的尾延迟比 Shinjuku 差得多。我们扩展了 ghOSt-Shinjuku 策略。该策略监控 RocksDB 的负载，并将空闲周期分配给批处理应用程序。图 4.2b 显示，我们修改后的 ghOSt 策略产生的尾延迟与原始 ghOSt 策略相同。如图 4.2c 所示，在保持 RocksDB 尾部延迟不变的同时，ghOSt 现在与批处理应用程序共享空闲的 CPU 周期。批处理应用程序可以利用的计算量类似于在 CFS 下运行时 nice 值为 19 的情况，而 RocksDB 的 nice 值为 -20。ghOSt 的几行代码结合了 Shinjuku 和 Shenango 的最佳特点，无需任何应用程序更改。

## 4.3 基于谷歌 Snap 的评估

我们现在评估 ghOSt 作为软实时内核调度器的效果，其主要管理 Snap 的工作线程。Snap 是一种用户空间的包处理框架，负责与 NIC 硬件的交互并运行自定义网络和安全协议。



### 4.3.1 当前工作线程的调度方式

Snap 始终保持至少一个工作线程进行轮询。当出现突发的网络负载时，Snap 可能会唤醒并随后使其他工作线程并进入睡眠状态。这些频繁的唤醒/睡眠需要调度器迅速干预，以避免增加延迟。如果通过现有的实时调度器（如 SCHED\_FIFO）来保证低延迟则会使系统不稳定，因为它可能会饿死同一台机器上的其他应用程序。因此，我们在生产中部署了 MicroQuanta，这是一种自定义的软实时调度器，保证在任何时间段（例如 1 毫秒）内，最多有一个时间片（例如 0.9 毫秒）分配给每个数据包处理工作线程。此策略确保工作线程获得运行时间，同时不会饿死其他线程。然而，这也会导致最多 0.1 毫秒的网络中断。

### 4.3.2 实验设置

我们使用了两台机器，每台机器有两块 Intel Xeon Platinum 8173M 处理器（每块处理器 28 个物理核，每个核有 2 个逻辑核，2GHz），383 GB DRAM 和 100Gbps NIC 及匹配的交换机。我们的测试是在单个插槽上进行，即每台机器有 56 个逻辑 CPU。

### 4.3.3 测试工作负载

我们的测试工作负载由六个客户端线程组成，每秒向另一台机器上的六个服务器线程发送 10k 条消息，并接收对等大小的回复。一个客户端线程发送 64 字节的消息，其他五个客户端线程每个发送 64kB 的消息。所有情况下的总带宽为 51.86Gbps。在所有实验中，客户端和服务线程由 Linux CFS 调度。在我们的 ghOSt 实验中，工作线程由 ghOSt 调度而不是 MicroQuanta。我们在两种模式下运行测试。在安静模式下，客户端/服务器线程是其机器上的唯一显式工作负载。在负载模式下，机器还运行 40 个额外的对抗线程，这些线程在网络流量较低时尝试使用服务器或 Snap 线程未使用的空闲 CPU 资源。

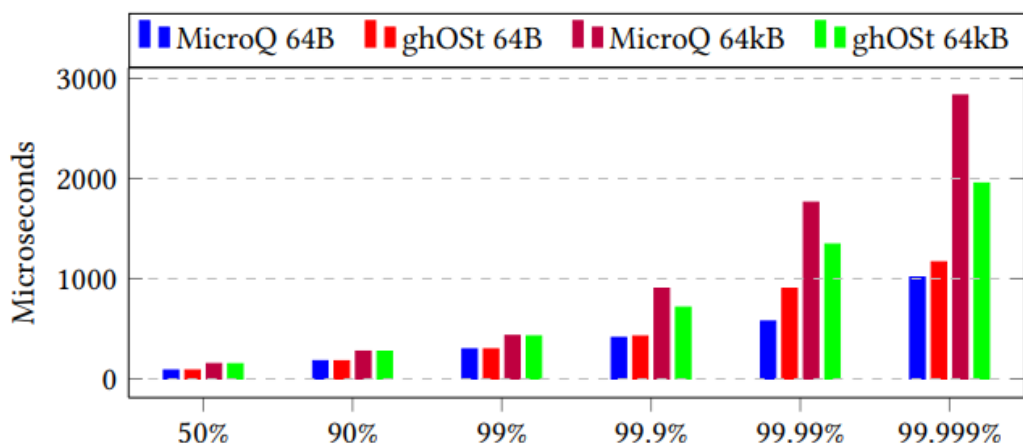
### 4.3.4 ghOSt 策略

我们用 ghOSt 策略替换了 MicroQuanta。该策略管理 Snap 的工作线程

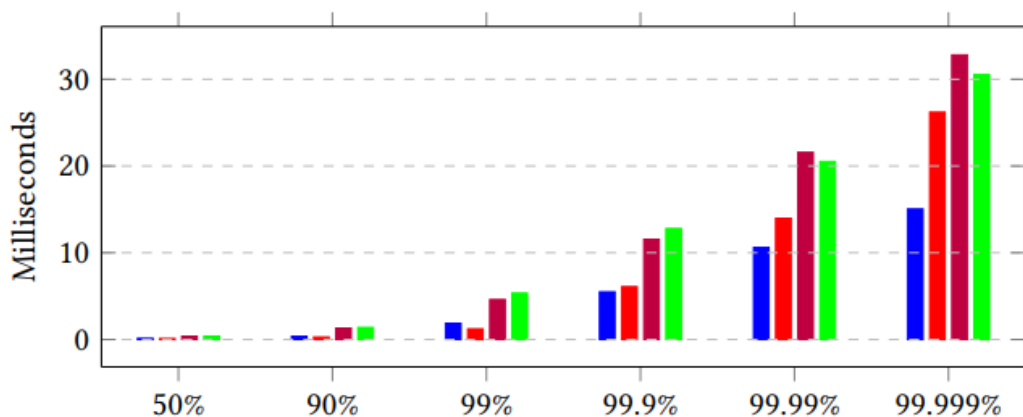
和对抗线程。全局代理尝试找到空闲 CPU 来调度其线程，优先级严格高于对抗线程。在安静模式测试中，工作线程经常被客户端/服务器线程和其他由 CFS 调度的本地线程（例如，重要但周期性的守护进程）抢占。在负载模式测试中，Snap 工作线程会抢占对抗线程，但不能抢占由 CFS 管理的线程。对抗线程只有在 CFS 线程和 Snap 都有空闲资源时才运行。

### 4.3.5 尾延迟比较

图 7 比较了客户端请求在机器中使用两种调度器时的往返尾延迟。我们分别为 64B 和 64kB 消息展示尾延迟。对于 64B 消息，ghOSt 的性能与基线相似或好 10%。对于 99.99% 及以上的情况中，ghOSt 的延迟高出最多 1.7 倍。由于 64B 消息的包处理计算量很小，因此，当流量突发时，存在较多 64B 消息，ghOSt 的调度事件开销就更加明显。对于 64kB 消息，ghOSt 在 99 % 的性能与基线相似（在 15% 范围内）。对于 99.9 % 及以上，ghOSt 的尾延迟降低了 5% 到 30%。64kB 消息需要更多处理时间（用于复制数据），因此调度事件较少。ghOSt 在某些情况下表现优于 MicroQuanta，是因为当服务器进程线程使 CPU 繁忙时，它可以重新定位工作线程。一个非常简单的 ghOSt 策略在不需要修改 Snap 的情况下，性能与定制的内核调度器相似。ghOSt 允许快速实验和性能优化，而这在内核调度器中要困难得多。



(a) Only networking load (quiet test).



(b) With additional load (loaded test).

图 4.3 消息延迟

## 4.4 基于谷歌 Search 的方案评估

我们现在评估 ghOst 作为谷歌搜索查询服务上的调度程序的性能。

### 4.4.1 测试工作负载

基准测试包括一个在单独机器上运行的查询生成器，发送三种不同的查询类型，分别为 A、B 和 C。A 类型查询是 CPU 和内存密集型查询。B 类型查询计算量小，但需要访问 SSD，由根据需要唤醒的短生命周期工作线程处理。C 类型查询是长生命周期的 CPU 密集型负载。在数据包输入时，所有查询首先由若干服务器线程处理，这些线程创建子查询。一些子查询必须由特定 NUMA 节点上的工作线程处理，以利用数据局部性优势。

## 4.4.2 实验设置

我们在两台 AMD Zen Rome 处理器的机器上评估 ghOSt，每台机器共有 256 个 CPU（2 个插槽，每个插槽 64 个物理核心，每个核心 2 个逻辑核心）。

## 4.4.3 ghOSt 策略

我们使用 ghOSt 的集中式模型实现了一个策略，即由单个全局代理调度所有 256 个 CPU。在启动时，全局代理首先使用 sysfs 生成系统拓扑模型。然后，全局代理使用拓扑信息根据 NUMA 偏好调度线程，并在可能时优先在 CCX 上运行线程。全局代理维护一个按线程运行时间排序的小顶堆，其中运行时间最短的线程优先执行。

当生成新的工作线程时，其 CPU 掩码设置（通过 `sched_setaffinity()`）为其查询数据所在插槽的 CPU 集合。该 CPU 掩码作为 `THREAD_CREATED` 消息的一部分发送给全局代理。当 ghOSt 全局代理希望运行其运行队列前端的下一个线程时，它将线程的 CPU 掩码与空闲 CPU 集合相交。如果交集为空，代理跳过该线程并调度运行队列中的下一个线程，在调度循环的下一次迭代中重新访问跳过的线程。

当每个线程在拥有 L1、L2 或 L3 缓存的 CPU 上运行时，搜索查询性能会有所改善。对于每个线程调度事件，我们的 ghOSt 策略将线程分配给离上次运行的 CPU 最近的空闲 CPU 目标。策略首先在线程上次运行的 CPU 的 L1 和 L2 缓存域内搜索可用 CPU。如果没有找到空闲 CPU，策略将搜索范围扩展到 CCX（L3 缓存）域。如果这也失败，则回退，寻找上次运行线程的 CCX 的最近邻，以避免跨 CCX 通信延迟而产生的昂贵线程迁移成本。

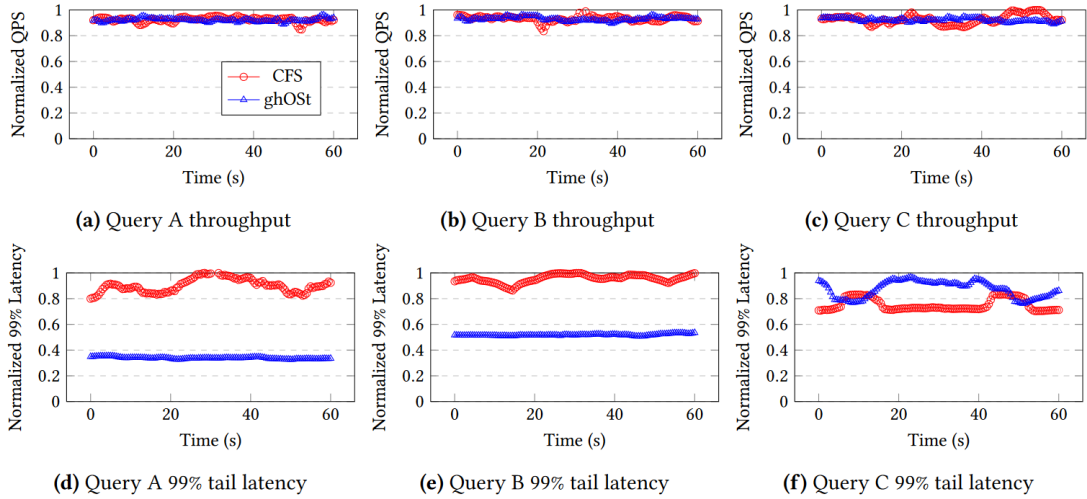


图 4.4 使用 CFS 和 ghOSt 调度的基准结果

#### 4.4.4 CFS vs. ghOSt

图 4.4 比较了使用 CFS 和 ghOSt 策略在 60 秒内的规范化查询延迟和吞吐量。图 4.4a-c 显示，ghOSt 提供了与 CFS 相当的吞吐量。CFS 和 ghOSt 都考虑 NUMA 插槽和 CCX 布局。NUMA 和 CCX 优化对于实现与 CFS 的性能相当至关重要，因为它们分别提供了 27% 和 10% 的吞吐量提升。在短 ghOSt 策略中迭代优化 NUMA 和 CCX 布局比在内核 CFS 代码中实验更容易。每次修改 ghOSt 代理只需重新启动代理的进程，而任何对 CFS 的修改都需要安装内核和重启。

#### 4.4.5 尾延迟

图 4.4d-f 显示，对于查询类型 A 和 B，ghOSt 比 CFS 大约减少了 40-45% 的尾延迟，而对于查询类型 C，尾延迟相当。在进行插槽和 CCX 感知优化之前，ghOSt 策略导致查询类型 A 的延迟几乎增加了 2 倍，对于查询类型 B 和 C 的延迟与 CFS 相当（即在 10% 范围内）。查询类型 A 受内存限制，从拓扑优化中受益最大。查询类型 B 访问内存和 SSD，而查询类型 C 主要受计算限制。对于 B 和 C，ghOSt 策略从一开始就有优势，因为全局代理在整个系统中能迅速反应，在微秒级别重新平衡负载。