



面向服务器无感知计算的可定制函数调度

张信民^{1,2,3,4}, 李星儒^{1,2,3,4}, 樊浩^{1,2,3,4*}, 黄卓^{1,2,3,4}, 吴松^{1,2,3,4}, 姚德中^{1,2,3,4},
金海^{1,2,3,4}, 余辰^{1,2,3,4}

1. 华中科技大学大数据技术与系统国家地方联合工程研究中心, 武汉 430074

2. 华中科技大学服务计算技术与系统教育部重点实验室, 武汉 430074

3. 华中科技大学集群与网格计算湖北省重点实验室, 武汉 430074

4. 华中科技大学计算机科学与技术学院, 武汉 430074

* 通信作者. E-mail: haofan@hust.edu.cn

收稿日期: 2024-11-10; 修回日期: 2025-01-23; 接受日期: 2025-02-10; 网络出版日期: 2025-02-27

国家自然科学基金 (批准号: 2022YFB4502001, 62032008, 62232012) 资助项目

摘要 服务器无感知计算凭借其弹性、高效和低成本的优势已成为下一代云计算的重要组成部分. 随着服务器无感知应用的复杂度和多样性的增加, 现有服务器无感知计算平台通常将不同应用的函数实例混合部署在统一服务器主机中, 并使用统一的 CPU 调度策略 (如 Linux 完全公平调度) 进行调度. 由于不同应用对性能、延迟以及调度公平性等的要求都不尽相同, 使得单一的全局调度策略, 难以满足多样化的调度需求. 此外, 实验发现, 函数实例的执行时间跨度较大, 短任务占比高, 且冷启动对任务完成时间影响显著. 这使得现有调度策略难以兼顾任务的效率与公平性, 尤其会增加短任务的调度延迟. 为解决上述问题, 我们提出并实现了一种调度隔离机制, 允许在同一服务器上为单个或多个应用使用独立的 CPU 调度策略. 此外, 我们设计了 FaaSchedule, 一种新的可定制调度策略, 综合考虑函数的执行时间、启动时间和等待时间, 并支持根据不同应用需求动态定制调度策略. 在 OpenWhisk 平台上的实验表明, FaaSchedule 显著降低了短任务的平均完成时间, 并有效支持调度隔离与策略定制.

关键词 CPU 调度, 服务器无感知计算, 冷启动, 调度隔离, 定制化内核

1 引言

服务器无感知计算 (serverless computing) 已经成为下一代云原生应用编程范式^[1]. 服务器无感知计算的目标是让开发者专注于应用程序的代码逻辑, 而无需关心与云服务器相关的复杂、低效的工作, 如资源分配回收、数据冗余、网络配置以及安全性防御等^[2]. 服务器无感知计算将应用拆分为细粒度的函数, 每个函数都可以单独部署、调用和运行^[3~5]. 当请求到达时, 服务器无感知平台会自动创建一个“函数实例”来执行该函数的代码以响应请求^[6]. 因此, 云服务提供商通常以函数即服

引用格式: 张信民, 李星儒, 樊浩, 等. 面向服务器无感知计算的可定制函数调度. 中国科学: 信息科学, 2025, 55: 481–499, doi: 10.1360/SSI-2024-0339

Zhang X M, Li X R, Fan H, et al. Customizable function scheduling for serverless computing. Sci Sin Inform, 2025, 55: 481–499, doi: 10.1360/SSI-2024-0339

务 (function-as-a-service, FaaS) 的形式向用户和开发者提供服务器无感知计算服务, 如 AWS Lambda¹⁾, Azure Functions²⁾和 Google Cloud Functions³⁾.

在服务器无感知计算平台中, 函数不仅是应用程序的基本执行单元, 同时也是云提供商的资源分配和计费单元^[7]. 服务器无感知计算平台根据负载的大小动态地扩展或缩小函数实例的数量, 可以轻松扩展出数以千计的函数实例, 为云应用提供了极致的弹性扩展能力^[8]. 此外, 每当函数实例空闲时, 服务器无感知计算平台可以回收分配给函数实例的计算和存储资源, 同时根据函数的资源使用量向用户收费, 这很大程度地节省了用户的云服务租赁成本. 极致的弹性扩展能力和细粒度的资源计费模式吸引了越来越多的复杂应用, 如线性代数计算^[9]、MapReduce 类任务^[10,11]和机器学习模型训练^[12~14]等, 迁移至服务器无感知计算平台^[15].

在服务器无感知计算平台中, 来自不同用户的各类应用的函数实例混合部署在服务器上. 不同的函数实例之间通过轻量级的隔离技术, 即容器, 实现资源隔离. 当一个调用请求被执行时, 服务器需要首先创建一个容器, 然后注入函数的代码, 并加载相关依赖库和进行初始化操作. 创建函数实例的过程时间较长, 这也被称为冷启动问题. 容器使用 Linux 内核提供的控制组和命名空间机制为每个函数实例提供了隔离的运行环境以确保函数实例能够正确、高效地执行. 对于 CPU 资源来说, 控制组可以使函数实例在指定的 CPU 上运行, 并设置并发函数实例之间使用 CPU 的权重. 在同一台服务器上, 基于容器的函数实例共享 Linux 内核, 所有的函数实例都在统一的 CPU 调度框架下被调度执行. 减少函数实例在调度过程中的排队时间和冷启动开销是提高函数执行性能的关键. 其中, 函数的冷启动问题已经得到了广泛的研究, 服务器无感知计算平台通常可以通过热启动避免初始化开销, 即保持一些处于空闲状态的函数实例, 以便在新的函数调用到达时可以直接复用这些函数实例. 然而, 如何在服务器无感知计算场景下优化函数调度策略仍然是一个具有挑战性的问题.

通过实验分析我们发现, 服务器无感知计算平台中的函数任务存在两个显著特征: (1) 函数任务执行时间的范围较广, 从几毫秒到几分钟跨越了 7 个数量级, 并且短任务占比很大; (2) 冷启动对函数任务完成时间有显著影响. 这两个特征导致现有的经典任务调度算法难以直接应用于服务器无感知计算平台. 例如, 默认的 Linux 调度器 CFS (completely fair scheduler) 是通用的公平性调度算法, 对工作负载不敏感, 即无论是长时间运行的任务还是短生命周期的任务, 都会在细粒度的时间片中获得 CPU 资源的比例份额. 这导致所有具有相同优先级的 CPU 任务都要花费“公平”的时间等待重新调度. 然而这对短任务而言是非常不利的, 因为 CFS 频繁地为短函数进行上下文切换, 导致函数存在很大的调度等待开销. 一些服务器无感知计算平台利用先来先服务 (first come first serve, FCFS) 算法进行函数任务的调度. 然而, FCFS 算法会导致队头阻塞 (head-of-line blocking) 问题, 即当队头的函数执行时间很长, 或者队头函数任务需要冷启动新的函数实例时, 会导致队列后面的请求在开始执行前需要等待较长时间. 这会使整个系统的平均函数完成时间增长. 短任务优先 (shortest job first, SJF) 和最短剩余时间优先 (shortest remaining time first, SRTF) 等, 可以实现最小的平均响应时间, 但是可能导致长任务长时间得不到执行. 并且 SJF 和 SRTF 只考虑了函数的执行时间, 而忽略了函数的启动时间, 使得一些需要冷启动的调用依然会导致队头阻塞问题. 最佳任务优先 (best job first, BJF) 和多队列调度 (multi-queue scheduling, MQS) 等算法会周期性地调度长任务执行, 从而避免长任务被饿死, 以保证一定的公平性. 然而由于函数任务的执行时间跨度极大, 这些方法难以精确地寻找最佳的平衡点来平衡响应时间和公平性.

本文首先基于 CFS 调度器提出了一个调度隔离框架, 该框架将服务器的全局调度和应用的局部调度解耦, 使局部调度可以根据具体场景为应用进行个性化配置调度策略. 然后, 我们针对服务器无感知计算平台中并发函数任务执行时间跨度范围大的特点, 提出了新的 CPU 调度原则: 通过在长任

1) <https://aws.amazon.com/lambda/>.

2) <https://azure.microsoft.com/en-us/services/functions>.

3) <https://cloud.google.com/functions>.

务不超过或略微超过 IDT (invocation deadline time, 在完全公平的调度算法下, 任务的完成时间) 的条件下优先调度短任务, 可以在不损害或仅略微损害公平性的情况下提高平均响应时间. 基于该原则, 我们提出了一种新的可定制的调度算法 FaaSchedule, 它综合考虑了函数的执行时间和启动时间以及等待时间. FaaSchedule 不仅保证每个函数任务的实际完成时间都不晚于其在完全公平的调度算法下的完成时间, 而且能够尽可能地减少函数任务的平均完成时间. 此外, FaaSchedule 支持动态调整调度策略, 即通过参数设置动态调整调度策略在公平性和效率之间的权衡, 从而支持定制符合应用需求的调度策略. 策略包括: (1) 完全公平调度策略; (2) 先来先服务调度策略; (3) 追求最低平均完成时间的短任务优先策略; (4) 在不损害长任务公平性的前提下, 最小化平均完成时间; (5) 完全自定义策略, 在可接受范围内适当牺牲长任务公平性, 以最小化平均完成时间. 我们在 OpenWhisk⁴⁾平台上实现了 FaaSchedule 的原型系统, 并使用服务器无感知计算负载数据集进行了实验. 实验结果表明, 相对于现有的调度算法, FaaSchedule 可以显著降低短任务的平均完成时间, 并几乎不会对长任务造成影响. 相关代码已开源⁵⁾.

2 背景与动机

服务器无感知计算已经成为下一代云编程范式^[1,3], 它提供了将应用程序分解为细粒度函数的能力. 服务器无感知计算平台可以独立地部署和执行这些函数, 并根据每个函数的实际负载大小弹性的扩展函数实例, 提供了极致的弹性扩缩能力^[2,3,6]. 因此, 服务器无感知计算通常由主要的商业云服务提供商以功能即服务 (FaaS) 的形式提供, 如 AWS Lambda, Azure Functions 和 Google Cloud Functions. 更具吸引力的是, 服务器无感知计算仅对实际使用的资源收费, 为租户提供了显著的成本效益^[1,8]. 因此, 预计服务器无感知计算在本十年内将实现 23% 的复合年增长率⁶⁾⁷⁾.

2.1 服务器无感知计算平台概述

图 1 展示了一个通用的服务器无感知平台架构以及调用调度的流程. 从函数任务启动到执行, 需要 3 个不同的调度过程. (1) 负载均衡器将来自不同触发器发起的调用分发到一个工作节点. (2) 在工作节点内, 所有调用都会被分配到一个函数实例中进行处理, 每个函数实例可以并发处理多个调用, 这往往取决于用户的初始配置, 如在 Knative 中, 默认每个函数实例可以同时处理 100 个调用. 如果当前存在一个函数实例有足够并发容量, 则该函数任务会直接分配给这个实例进行热启动. 如果没有合适的函数实例, 则创建一个新的实例来处理该任务, 称为冷启动. (3) 最后, 函数实例会为每个请求创建一个线程或进程, 被内核调度器 (如 Linux 的完全公平调度器 CFS) 调度在执行 CPU 上执行. 我们的研究主要集中在最后一个阶段, 这是服务器无感知平台效率的一个关键组件^[16~18].

2.2 Linux 调度机制

Linux 原生支持的调度策略主要包括 3 种: 先进先出 (first-in first-out, FIFO)、轮转调度 (round-robin, RR) 和完全公平调度 (CFS). 在 Linux 中上述调度策略可以同时存在, 即可以为不同的进程设置不同的调度策略. 在 Linux 中 FIFO 和 RR 属于实时调度类, 拥有更高的执行优先级, 即在存在实时调度类的进程时, CFS 调度的进程将得不到调度机会.

4) <https://openwhisk.apache.org/>.

5) <https://github.com/CGCL-codes/faaschedule.git>.

6) <https://www.reportsanddata.com/report-detail/serverless-architecture-market>.

7) <https://www.cbinsights.com/research/serverless-cloud-computing>.

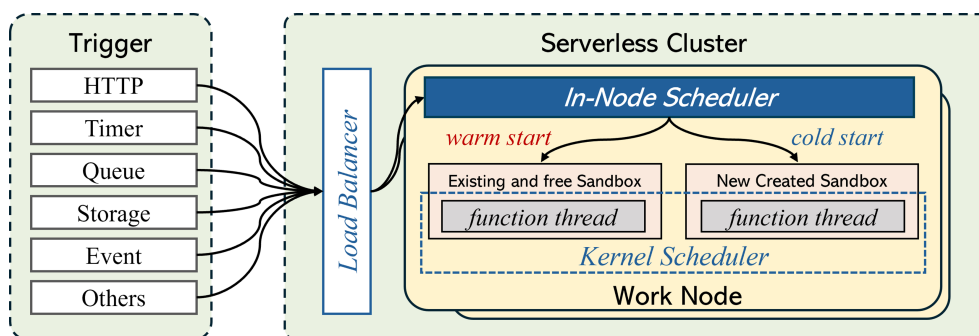


图 1 (网络版彩图) 通用服务器无感知计算平台架构.

Figure 1 (Color online) Architecture of a general serverless platform.

2.2.1 基础调度策略

FIFO 和 RR 是最基础的调度策略. 当 FIFO 任务开始运行时, 它会一直运行直到完成. 当系统高度整合并且在高负载下有排队任务时, FIFO 调度可能会影响响应时间. RR 可以用来优化响应能力. RR 将 CPU 任务分配给一个时间片, 然后切换到下一个排队任务. 然而, 由于 CPU 任务的执行被划分为多个时间片, RR 牺牲了周转时间.

2.2.2 公平性调度策略

CFS 是 Linux 默认的调度器, 其主要目标是实现 CPU 任务级别的公平性: 无论是长时间运行的任务还是短生命周期的任务, 都会在细粒度的时间片中获得 CPU 资源的比例份额. CFS 依据任务的权重 (优先级) 将物理时间分割为细粒度的时间片, 分配给所有的 CPU 任务. CFS 使用虚拟运行时间 (vruntime) 机制来跟踪每个任务的 CPU 时间使用情况. vruntime 记录任务使用的 CPU 时间, 并按任务优先级加权. 在多核系统中, 每个物理 CPU 核心都有自己的运行队列, 运行队列采用按 vruntime 排序的红黑树 (RB 树). 任务首先被 CFS 分配到一个运行队列, 任务在运行队列中所在的位置大致决定了它未来何时能够执行; 在每次调度时 (即每个时间片的结束), CFS 从 RB 树中选择 vruntime 最小的下一个任务.

2.2.3 组调度

为了避免某些应用通过大量创建子进程来争夺更多的 CPU 时间份额, 从而破坏 CFS 调度器所追求的公平性, Linux 引入了组调度机制. 在这种机制下, CFS 调度器的调度实体不仅可以是单个线程, 还可以是一个线程组. 这样, CFS 调度器从传统的单一红黑树结构演化为层级红黑树结构.

组调度的层级结构以根调度组为起点, 根据需求动态生成其他调度组. 每个 CPU 维护一个运行队列, 每个调度组在对应 CPU 上也有自己的子运行队列. 在这种模式下, 线程组作为调度实体, 可以进一步包含子层级的运行队列, 形成层级嵌套的结构. 在调度过程中, 调度器首先在根调度组中选择一个调度实体, 判断其是线程还是线程组. 如果是线程, 直接将其作为下一次运行的任务; 如果是线程组, 则进入该组的子运行队列, 继续选择调度实体.

调度组的设计为实现 CPU 调度策略的隔离提供了基础. 由于每个并列的调度组之间是相互独立互不干扰的, 他们仅受到更上层的调度控制. 我们可以将需要不同的调度策略的应用的函数实例分配到不同的调度组中, 然后允许每个调度组拥有自己的调度策略即可.

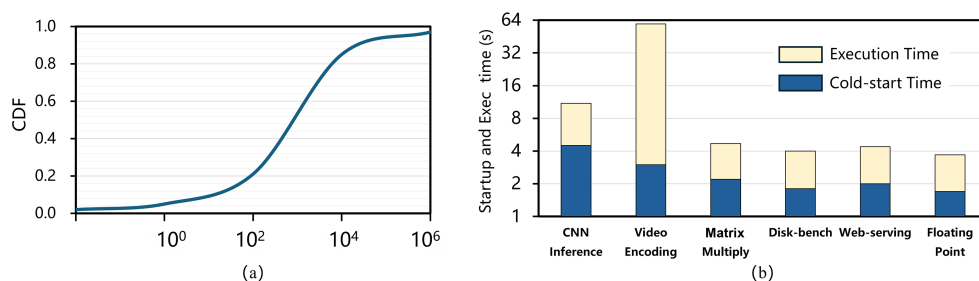


图2 (网络版彩图) 服务器无感知计算应用的执行时间特征。(a) 函数执行时间 (ms); (b) 服务器无感知应用冷启动和执行时间分解。

Figure 2 (Color online) Execution time characteristics of serverless computing applications. (a) Function execution time (ms); (b) serverless APP cold-start and execution time breakdown.

2.3 函数任务的特征分析

函数任务的执行时间特征对于制定有效的调度策略至关重要。为了获取服务器无感知调用的特性, 首先, 本文使用来自 Azure Functions 的追踪数据⁸⁾分析了真实服务器无感知计算工作负载的执行时间。随后, 本文使用 OpenWhisk 分析并解构了 6 个服务器无感知基准函数的执行过程, 这些函数来自文献 [19~21], 涵盖了常见类型的服务器无感知应用程序。结果如图 2 所示。通过这些分析, 我们识别出了两个主要的时间特性: 首先, 函数任务的执行时间跨度范围极广, 且较短的任务占据了较大比例。函数任务的执行时间是指从任务被调度到沙箱到函数执行完成的时间段。我们发现执行时间跨越了 7 个数量级, 范围从毫秒到小时不等。此外, 我们的分析揭示, 超过 57.98% 的调用执行时间少于 1 s, 这一发现与 AWS Lambda 的报告⁹⁾一致。其次, 冷启动对调用运行时间的影响不可忽视。一个函数的执行时间包括两个部分: (1) 创建函数实例并初始化执行环境; (2) 执行函数代码 (之前称为“执行时间”)。对于热启动, 第 1 部分的时间可以视为零。对于冷启动, 我们的测试表明, 创建和初始化所花费的时间可能构成显著的开销。冷启动开销最多可以占到总执行时间的 95.68%。

2.4 问题分析

CPU 调度策略对于服务器无感知计算平台至关重要^[17]。然而, 现有的调度算法往往未能充分考虑服务器无感知函数任务的特性, 可能导致平台总体性能下降和并发函数任务的公平性问题。

服务器无感知计算节点内调度算法。 现有服务器无感知计算平台的任务调度算法主要采用先到先服务 (FCFS) 策略^[22]。这与 Linux 内核支持的 FIFO 策略类似。然而, FCFS 通常会导致队头阻塞, 并且由于其主要考虑任务等待时间, 它往往对短任务不公平。节点内调度器考虑的是尽快地将请求派发给具体的函数实例, 因此可以采用简单高效的 FCFS 算法。但是我们无法在内核调度层面忽略这些问题, 特别是对于任务执行时间差异大且短任务居多的服务器无感知计算场景中。

任务调度算法。 最短作业优先 (SJF) 和最短剩余时间优先 (SRTF) 能够实现最小的平均响应时间, 并且在许多领域得到了广泛的研究^[23~25]。然而, 这些方法会导致长任务无法执行。最佳作业优先 (BJF)^[18, 26] 和多队列调度 (MQS)^[27] 试图平衡短任务与长任务之间的公平性, 但它们都忽视了服务器无感知计算中昂贵的冷启动开销, 他会显著影响调度策略对短任务和长任务的判定。此外用户难以根据应用的需求定制化 BJF 和 MQS 策略。

Linux 提供的调度算法。 经典调度器如 Linux 的完全公平调度器 (CFS) 和轮询调度 (RR) 基于优先级和时间片调度进程, 以实现对各种工作负载的公平性。然而它们更多的是追求公平而非性能。随着服务器无感知计算工作负载的不断整合, 通常会有数千个并发运行的函数进程, 共享主机上有限的

8) <https://github.com/Azure/AzurePublicDataset>.

9) <https://www.datadoghq.com/state-of-serverless-2020/>.

CPU 核心. 任务的权重仅表示相对的 CPU 份额, 而不是用户期望根据函数的资源配置分配给该函数的绝对 CPU 份额. 一旦被抢占, 任务需要在运行队列中等待下次运行的机会. 这对短任务而言是难以接受的, 因为 CFS 频繁地为短函数进行上下文切换, 导致函数存在很大的调度等待开销, 这一开销甚至超过函数实际执行时间^[28].

因此, 越来越多的研究致力于为应用程序设计定制化的调度策略. 例如, 文献 [29] 指出, CFS 调度器采用类似于后来先服务 (last come first served, LCFS) 的模型, 可能导致较高的尾延迟. 为此, 该研究建议针对同一进程的子进程采用先来先服务 (FCFS) 策略, 从而提升大型交互式网站服务器场景的性能. 文献 [30] 则分析了 CFS 调度器中类似轮询的行为如何加剧 Java 应用中多线程的堆争用, 提出采用先进先出 (FIFO) 调度策略能够将垃圾回收时间开销减少 21%. 此外, 文献 [31] 通过修改 CFS 调度器, 使其在与其他工作负载共享服务器时, 能够为低延迟工作负载维持高服务质量 (quality of service, QoS). 文献 [32~34] 等则在不同场景下采用核心重新分配、抢占以及工作窃取等技术进行优化. 这些专用的优化策略能够根据特定需求进行定制, 从而显著提升系统性能.

然而在服务器无感知计算的平台中, 大量需求不同的应用混合部署, 上述定制化的调度策略都会影响整个服务器的调度策略. 在服务器无感知计算平台中实现为不同应用的定制化调度策略, 需要解决如何在同一台服务器上并行运行多种调度策略, 同时确保这些策略在隔离的环境中运行, 不会对其他应用的调度策略造成影响. 现有的 Linux 内核虽然支持多个调度策略, 可以为不同的进程设置不同的调度策略, 包括 FIFO, RR 和 CFS 等^[35]. 然而 FIFO 和 RR 调度是实时调度类, 拥有超越采用 CFS 策略的进程的高优先级, 即当存在实时调度类的进程时, CFS 调度的进程将得不到调度机会. 这使得在服务器无感知计算平台中无法单独为某个应用设置 FIFO 或 RR 调度策略.

全局调度器或负载均衡算法. ENSURE^[36], FnSched^[37] 和 FaaSRank^[38] 基于低延迟 I/O、数据局部性和热实例缓存来优化函数的部署. Wukong^[39], Sequoia^[40] 和 SLA^[15] 为服务器无感知工作流应用程序应用了集群调度技术的变体. Atoll^[41] 和 Orion^[42] 通过使用用户提供的性能约束来优化函数 DAG 的执行. Hermod^[16] 考虑了热启动和负载均衡. 这些工作负载全局的负载均衡, 与我们的工作正交的.

3 函数任务调度指标的量化

本节主要介绍平均响应时间和公平性的量化方法, 并介绍面向服务器无感知计算调度算法设计思想. 主要符号及其含义如表 1 所示.

3.1 函数任务的平均响应时间

函数任务的响应时间是指从触发器启动任务到函数执行完成的持续时间. 调用的平均响应时间 (invocation average response time, IART) 是评估调度策略的一个关键指标, 因为它直接影响用户体验和服务器无感知平台的整体性能. 我们定义 IART 如下:

$$\text{IART} = \frac{1}{|I_w|} \sum_i^{|I_w|} \left(T_i^w + T_{F[i]}^s + T_{F[i]}^e \right), \quad (1)$$

其中, T_i^w 表示调用在调度队列中的等待时间, 这完全由调度策略决定. BJS 和 SRTF 策略正是通过最小化等待时间实现了最小的平均响应时间. 此外, IART 还包括前面提到的启动时间 ($T_{F[i]}^s$), 即创建沙箱并初始化执行环境的时间, 以及执行时间 ($T_{F[i]}^e$), 即执行函数代码的时间. T_{start} 对于热启动为零, 对于冷启动是一个固定值.

表 1 符号表.

Table 1 Symbol table.

Symbol	Description
$F = \{f_1, \dots, f_n\}, n \in \mathbb{N}$	The set of functions of serverless application.
$I = \{i_1, \dots, i_m\}, m \in \mathbb{N}$	The set of all invocations of serverless application.
$I_w, I_w \subset I$	The set of invocations that arrive at a certain working node.
$I_u, I_u \subset I_w$	The set of invocations unfairly scheduled in a certain worker node.
$I_q, I_q \subset I_w$	The set of invocations queued in a certain work node.
$T_f^e, f \in F$	The execution time of f , excluding the scheduling waiting time, in ms.
$T_f^s, f \in F$	The startup time of the f , in ms. 0 if warm start.
$T_f^r = T_f^e + T_f^s, f \in F$	The running time of the f , including T_f^e and T_f^s , in ms.
$T_i^w, i \in I_w$	The scheduling wait time for invocation i , in ms.
$F[i], F[i] \in F, i \in I$	The function corresponding to the invocation i .
$P_i, i \in I_q$	The scheduling priority of the invocation i .

3.2 并发函数任务的公平性

公平性是评估调度策略的另一个关键指标,特别是在多租户和不同类型函数混合的服务器无感知计算平台中. 我们将具有无限小时间片的轮询 (RR) 调度器作为公平性上限, 因为 RR 确保所有任务可以并发执行且平等地共享 CPU 时间. 基于此, 我们定义以下两个公平性指标:

$$\begin{cases} \text{AIF} = \frac{1}{|I_u|} \sum_{i=1}^{|I_u|} \frac{T_{\text{fair}}}{T_{\text{actual}}}, & |I_u| > 0, \\ \text{AIF} = 1, & |I_u| = 0, \\ \text{RIF} = \frac{1}{|I_w|} \sum_{i=1}^{|I_w|} \min\left(1, \frac{T_{\text{fair}}}{T_{\text{actual}}}\right), & |I_w| > 0, \end{cases} \quad (2)$$

其中, 将函数在前面提到的完全公平条件下的执行时间定义为 T_{fair} , 而在实际调度策略下的执行时间定义为 T_{actual} . 我们通过 T_{fair} 与 T_{actual} 的比值来计算每个调用的公平性感知. 比值越小, 表示感知到的不公平性越大, 而比值越大, 表示感知到的不公平性越小. 如果 T_{actual} 小于或等于 T_{fair} , 我们将该比值设为 1, 表示该调用认为当前的调度策略是完全公平的.

我们将绝对调用公平性 (absolute invocation fairness, AIF) 定义为所有受到不公平对待的任务的这一比值的平均值, 将相对调用公平性 (relative invocation fairness, RIF) 定义为所有任务的这一比值的平均值. AIF 和 RIF 的取值范围为 $(0, 1]$, 值越高表示调度策略对任务的公平性越好. 当所有调用的 T_{actual} 等于 T_{fair} 时, 表示完全公平的调度策略. RIF 放宽了公平性的要求, 认为通过牺牲少数任务的公平性来提高平均响应时间的策略是相对公平的. 例如在大量短任务和少量长任务并发的场景下使用的最短作业优先 (SJF) 策略.

权衡. 平衡响应时间和公平性是调度研究中的一个著名挑战^[23~25, 43, 44]. 在完全公平的调度策略中, 任务必须等到下一个调度周期才能继续执行, 直到耗尽其时间片. 这显著增加了短任务的周转时间, 从而对系统的平均响应时间产生不利影响. 相反, 像 SRTF 和 SJF 这样的策略通过延迟长任务来优先考虑短任务, 导致对长任务的重大不公平性.

3.3 调度示例

图 3 展示了不同算法的调度示例. 在此示例中, 一个单核服务器服务于两个函数, FuncA 和 FuncB. FuncA 包含 9 个函数任务: $A_1, \dots, A_9$, 第 1 个任务在 t_0 到达, 每个任务需要一个时间单位来执行.

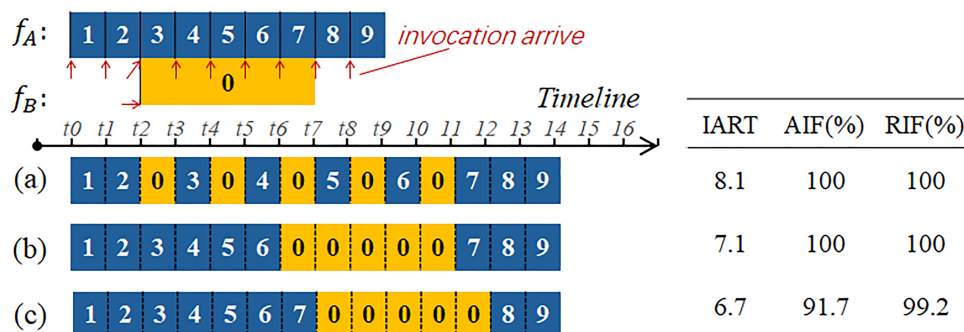


图 3 (网络版彩图) 一个在响应时间和公平性之间权衡的简单示例。

Figure 3 (Color online) A simple example of the trade-off between response time and fairness.

FuncB 只有一个任务 B_0 , 它在 t_2 到达, 执行需要 5 个时间单位。

算法 (a) 表示完全公平的理想情况, 其 (IART, RIF, AIF) 为 (8.1, 1, 1)。在完全公平的情况下, B_0 在 t_{11} 完成, 我们称之为调用截止时间 (IDT)。这意味着只要 B_0 在 t_{11} 之前完成, 它就认为调度是公平的。算法 (b) 考虑到这一点, 安排 B_0 在 t_6 运行, 并在 t_{11} 完成。结果, 调用 A_3 到 A_6 相比算法 (a) 更早完成。算法 (b) 产生的三元组为 (7.1, 1, 1), 表明平均响应时间减少了 12.3%, 且没有牺牲任何公平性。算法 (c) 将 B_0 安排在 t_7 运行, 并在 t_{12} 完成。这使得 B_0 多等待了一个时间单位, 但允许 A_3 提前 5 个时间单位完成。算法 (c) 产生的三元组为 (6.7, 0.917, 0.992), 表明平均响应时间减少了 17.3%, 但 AIF 减少了 8.3%, RIF 减少了 0.8%。

这个例子使我们重新审视在服务器无感知上下文中平衡效率和公平性的问题: 不同函数调用的执行时间可以有很大差异 (如在第 2.3 小节中讨论的), 这意味着非常长的任务相比非常短的任务对公平性延迟的容忍度要高得多。例如, 延迟一秒钟对分钟级调用可能是微不足道的, 但对毫秒级调用来说却是一个显著的损失。这意味着通过在不超过或略微超过 IDT 的条件下优先调度短任务, 我们可以在不损失或仅轻微损失公平性的情况下提高平均响应时间。

4 调度隔离框架

CPU 调度隔离的主要目的是为运行在同一服务器节点上的服务器无感知应用提供多样化、可定制的调度解决方案, 并保证服务器主机上不同应用之间相互隔离。如 2.2.3 小节所述, 组调度机制是一种天然的分层和隔离机制, 不同调度组之间是相对独立的, 他们之间会根据统一的 CFS 调度器进行调度, 但每个调度组内部的任务和其他调度组的任务是相互隔离的。基于此, 我们可以将服务器无感知应用的函数调度, 根据是否属于同一个应用, 抽象为两个调度层级: 全局调度层级和局部调度层级, 如图 4 所示。每个应用或多个具有类似调度需求的应用可以被分配到一个调度组中, 全局调度层级的调度器负责调度不同调度组之间的调度, 而局部调度层级的调度器负责调度调度组内部的任务。

在现有的 Linux 调度器中, 全局调度器和局部调度器都必须使用同一个调度策略, 即 CFS 调度器, 如图 4 所示, 局部调度器受控于全局调度器而无法单独地为每个应用或调度组提供不同的调度策略。然而, 分层结构本身赋予了局部调度器一定的灵活性, 各调度组内部的资源调度不会影响全局调度器中的其他调度组, 从而实现了调度隔离性。这种隔离性确保了不同调度组之间的调度状态互不干扰。在此基础上, 局部调度器可以引入新的任务调度策略, 而不会破坏调度组之间的隔离性, 从而保证服务器能够在应用之间实现性能和安全的无感知隔离。在 CFS 的调度过程中, 运行队列的管理 (如虚拟运行时间 $vruntime$ 的计算和排序) 以及资源分配 (如 $cpu.shares$ 参数的设置) 仅在同一层级内生效, 对父级或子级没有直接影响。这种架构使得全局调度器与局部调度器在调度行为上彼此独立且透明。

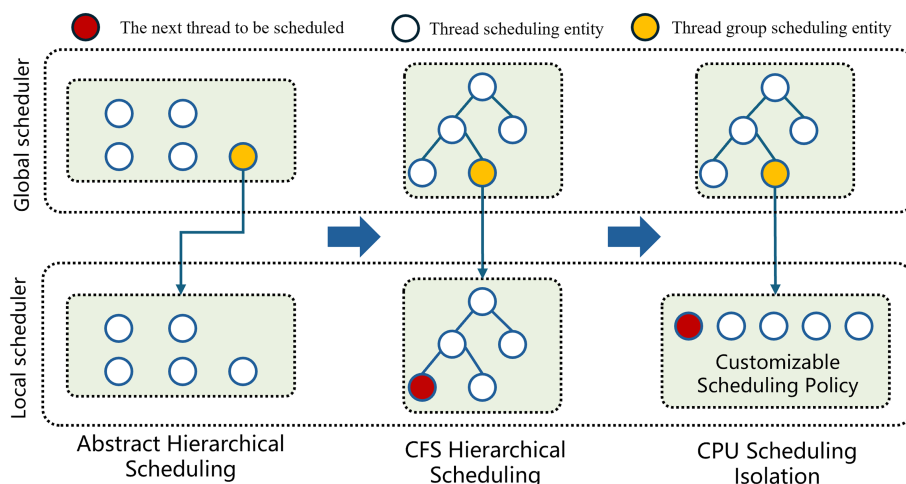


图 4 (网络版彩图) 基于双层调度架构的调度隔离机制。

Figure 4 (Color online) Scheduling isolation mechanism based on two-layer scheduling architecture.

两层调度结构中的父子级隔离性, 为每个局部调度器根据其具体需求定制调度策略提供了可能性。

如图 4 所示, 我们提出了一个调度隔离框架, 该框架将全局调度器和局部调度器解耦, 使局部调度器可以根据具体的应用场景进行个性化配置调度策略。具体而言, 全局调度器依然使用 CFS 调度器负责调度不同调度组之间的调度, 确保每个调度组中的应用能够根据其配置的 CPU 资源比例获得相应的 CPU 时间片。每当全局调度器选择一个调度组时, 它会将该调度组的任务分配给局部调度器, 由局部调度器根据调度组内部的定制的调度策略进行调度。这种调度隔离框架的设计确保了全局调度器和局部调度器之间的透明性, 使得全局调度器上的调度行为不会影响局部调度器上的调度行为, 同时局部调度器上的调度策略也不会影响全局调度器上的调度行为。这种调度隔离框架的设计使得每个服务器无感知计算应用都可以根据其特定的调度需求配置局部调度器以实现个性化的调度策略。需要注意的是, 每个调度组中并不只有一个应用, 而是可以包含多个应用, 这些应用可以共享调度组内部的调度策略, 前提是这些应用有相似的调度需求。

然而在服务器无感知计算中, 由于函数调用的特性, 传统的调度策略如 FCFS, SJF, SRTF 等往往无法满足服务器无感知计算的需求 (详见第 2.4 小节), 因此我们需要设计一种新的调度策略以允许用户可以通过简单的配置来实现不同的调度策略, 这也符合服务器无感知计算的尽可能减少用户的负担的设计理念。因此我们提出了 FaaSchedule 调度器, 该调度器可以根据用户的需求配置不同的调度策略, 以实现不同的调度目标。FaaSchedule 调度器的设计将在第 5 节中详细介绍。

5 FaaSchedule: 面向服务器无感知计算的可定制调度策略

5.1 目标与挑战

目标. 我们的目标是设计一个可定制的 CPU 调度策略, 能够在保持长任务的公平性或将其公平性影响降至最低的前提下, 减少短任务的响应时间; 此外, 用户可以根据实际的应用需求, 通过简单的参数修改去定制调度策略, 选择在牺牲多少公平性的基础上换取最高的性能收益。

挑战. (1) 如何确定任务的优先级? 当调用接近或已经达到任务截止时间 (IDT) 时, 其优先级应为最高, 因为延迟此类调用会导致过度的公平性损失。短任务的优先级应高于长任务, 以减少响应时间。调度器需要计算每个调用的优先级, 以选择最合适的调用进行调度。(2) 如何设置任务对调度延迟的容忍度? 调度器设计应允许服务器无感知平台轻松配置全局容忍度, 以便在不同场景中调整调度策略。(3) 如何最小化调度开销, 并在短时间内做出调度决策? 调度器的开销应最小化, 以确保调度过程

不会成为瓶颈,尤其是在高负载情况下.(4) 如何处理冷启动的不确定性? 冷启动对调用的执行时间有显著影响,且在调度时难以确定某个调用是否会发生冷启动,因为热实例可能被保活机制回收.

5.2 面向服务器无感知计算的短任务优先调度

我们引入了 FaaSchedule, 一种为服务器无感知计算设计的新 CPU 调度器. 它是一个基于优先级的调度器, 优先调度执行时间短的任务和接近其调用截止时间 (IDT) 的任务. FaaSchedule 为调用引入了一个新的优先级度量 $P(i)$. 对于调用 i , 其 $P(i)$ 的计算公式如下:

$$\begin{cases} P(i) = T_{F[i]}^r - \alpha \cdot T_i^w \cdot (1 - \varepsilon), \\ \alpha = \min \left(\frac{M_{\text{sche_group}}}{\sum_i^{I_{rw}} M[i]}, 1 \right). \end{cases} \quad (3)$$

FaaSchedule 优先调度具有最低 $P(i)$ 值的调用进行执行.

在式 (3) 中, $F[i]$ 表示与调用 i 对应的函数 f . $T_{F[i]}^r$ 表示函数运行时间, 即从调用被调度器选择到函数执行完成的时间. T_i^w 是调用 i 在调度器队列中的等待时间. $T_{F[i]}^r$ 和 $P(i)$ 之间呈正相关关系, 这意味着更长的函数运行时间会导致 $P(i)$ 的值更高, 从而优先级更低. 相反, T_i^w 与 $P(i)$ 呈反相关关系, 等待时间越长, $P(i)$ 值越小, 优先级越高. 这两个基本因素构成了 $P(i)$ 的基础, 代表了 FaaSchedule 对两类任务的优先级排序: (1) 短任务; (2) 接近其调用截止时间 (IDT) 的任务.

虚拟执行时间 (virtual execution time, VET). 我们不直接使用调用的等待时间, 而是将其乘以一个动态缩放因子 α . $\alpha \cdot T_i^w$ 表示虚拟执行时间 (VET), 即任务在完全公平的条件下理论上应该执行的时间. 由于一个完全公平的算法没有等待时间, T_i^w 测量的是从任务到达开始到我们计算 T_i^w 这一时刻所经过的时间. 当节点上有足够的资源来并行执行所有任务时, T_i^w 将等于实际执行时间, 因为任务可以连续获得时间片, 因此 α 的值为 1. 如果一个局部调度组总内存 ($M_{\text{sche_group}}$) 小于所有活跃任务和排队任务的内存需求总和 ($\sum_i^{I_{rw}} M[i]$), 任务必须交替使用 CPU 资源, 导致实际执行时间小于 T_i^w , 即 $\alpha \cdot T_i^w$. 例如, 如果两个任务每个都需要一个核心, 但只有一个核心可用, 则在时间 t 后, 它们的实际执行时间将是 $t/2$, 因此 α 为 0.5.

我们基于内存而不是 CPU 来计算 α , 因为在服务器无感知计算环境中, 内存和 CPU 资源是正相关的. 也就是说, 分配给一个函数的内存越多, 它获得的 CPU 资源也越多. 服务器无感知平台通常只提供接口供用户配置函数的内存资源. 调度组的总内存配置是由上层的全局调度器决定的. 如果当前节点中只存在一个调度组, 那么它将占据整个节点的物理内存, 反之, 如果存在多个调度组, 则根据每个调度组内的函数实例数量及其内存配置为每个调度组分配内存配额. 值得注意的是, α 与函数的内存设置无关, 因为它本质上反映了 sche_group 处理能力下降了多少, 而与特定函数的内存配置无关.

函数任务的运行时间 ($T_{F[i]}^r$) 与虚拟执行时间 (VET) ($\alpha \cdot T_i^w$) 之间的差异, 可以理解为函数的剩余虚拟运行时间 (remaining virtual runtime, RVRT), 使得 FaaSchedule 本质上成为一个最短剩余虚拟运行时间优先 (shortest remaining virtual runtime first, SRVRT) 算法. 随着长任务的等待时间增加, 其 RVRT 减小, 优先级逐渐增加, 从而确保了 FaaSchedule 系统中的公平性. 具有较小 RVRT 值的短任务将在完全公平的算法下始终比具有较大 RVRT 值的任务先完成. 因此, FaaSchedule 通过优先调度具有较小 RVRT 值的任务, 确保不会损失公平性.

公平性容忍度. 为了应对挑战 2, 我们引入了容忍度的概念, 用符号 ε 表示. 容忍度是一个可以在服务器无感知计算平台上轻松配置的全局参数, 其值范围在 $[0, 1]$ 之间, 表示任务可以容忍的公平性损失的程度. 当 ε 为 0 时, 表示任务不允许任何公平性损失, 调度器将完全基于公平性排序任务. 相反, 当 ε 为 1 时, 表示任务忽略公平性, 追求最佳性能, 调度器将优先调度短任务.

容忍度利用了长任务和短任务对时效性的不同敏感度. 当两个任务的执行时间相近时, 容忍度对它们的优先级影响较小. 然而, 当任务之间的执行时间差异较大时, 容忍度可以使得短任务更频繁地

获得优先调度. 这在执行时间差异显著的服务器无感知计算工作负载中尤为有效. 例如, 当 ε 设置为 10% 时, 一个 10 s 的任务可以容忍额外 1 s 的等待, 而一个 10 ms 的任务只能容忍额外 1 ms 的等待. 这个设置使得短时间任务能够更及时地被调度. 在这个例子中, 10% 的容忍度允许一个 10 s 的任务推迟到 100 个 10 ms 的任务后, 这显著减少了短任务的平均响应时间, 同时对长任务的影响最小.

5.3 FaaSchedule 的工作流程

获取时间信息. 为了计算 $P(i)$, FaaSchedule 需要函数的执行时间信息, 包括 $T_{F[i]}^s$, $T_{F[i]}^e$ 和 $T_{F[i]}^W$. 其中, $T_{F[i]}^s$ 表示函数 f 的冷启动时间, 包括创建容器实例、注入函数代码以及初始化执行环境的时间. 对于特定的函数 f , 冷启动开销是固定的, 因为每个 f 实例的冷启动过程是相同的. 执行时间 $T_{F[i]}^e$ 可以通过历史数据或预测模型来估算, 因为函数的执行时间是稳定的, 只有在函数代码发生变化时才会有所变化. 此前的研究已证明预测函数执行时间的准确性. $T_{F[i]}^s$ 和 $T_{F[i]}^e$ 都可以离线计算, 不会影响调度过程的性能. FaaSchedule 将这些信息存储在一个全局的函数执行时间表中, 以便在调度过程中快速访问. 至于 $T_{F[i]}^W$, FaaSchedule 会记录每个调用的到达时间, 并在更新 $P(i)$ 时计算它.

基于优先级的调用选择. 当一个运行中的请求完成, 并且节点的队列中有待处理的请求时, FaaSchedule 会选择优先级最高的请求来执行. 所有待处理的调用根据其优先级组织成一棵红黑树, 这使得 FaaSchedule 能够高效地选择优先级值最小的调用. 每当一个调用完成时, 一个新的热实例被创建, 这会影响与该新热实例关联的调用的优先级. FaaSchedule 随后会重新计算这些调用的优先级, 并将其与当前优先级最低的调用进行比较, 从中选择优先级更低的调用进行执行. 这种方法确保系统能够动态适应调用特性变化, 同时保持高效且公平的调度过程.

5.4 FaaSchedule 的优先级更新机制

每当有新请求到达时, 我们需要更新所有队列中请求的优先级并重新排序. 这会引入一定的计算开销, 特别是在服务器无感知计算平台负载较高时. 对于同一个函数, 较早的请求总是先被处理, 这意味着同一个函数的所有请求调度顺序可以视为遵循先来先服务 (FCFS) 策略. 因此, 维护每种函数类型的二叉树是足够的. 当选中的一个请求时, 如果该函数还有其他请求, 优先级更新可以基于该函数中最早到达请求的等待时间进行. 尽管优先级管理的粒度是每个请求, 但由于同一函数中, 较早到达的请求天然优先于后续请求, 因此只需在函数级别维护优先级排序队列, 这大大减少了维护的复杂度. 来自微软开源服务器计算平台 Azure Functions 的数据表明, 大约 0.6% 的无感知计算函数应用生成了 90% 的函数任务. 因此, 聚合函数请求可以有效减少函数请求调度的开销.

5.5 调度策略定制

FaaSchedule 方法可以通过修改式 (3) 中的参数, 适配成各种不同的变体, 从而实现其他更简单或更专门化的调度策略. 例如, 当缩放因子 α 固定为 1, 并且冷启动开销 T_i^s 始终为零时, 可以实现以下几种变体: (1) 将 ε 设置为 0, 这样只使用函数的执行时间和请求的等待时间作为优先级衡量标准, 从而实现一个最佳作业优先 (BJF) 策略, 能够减少平均响应时间, 并防止长任务饿死; (2) 将 ε 设置为 1, 这样只考虑函数的执行时间, 从而实现一个最短作业优先 (SJF) 策略; (3) 仅保留到达时间 T_i^w , 则实现一个先来先服务 (FCFS) 策略; (4) 通过调整容忍度 ε , 我们可以在不同的场景中根据公平性和性能的不同需求平滑调整调度策略.

FaaSchedule 的最大优势是可以根据应用的特点定制调度策略. 一般性的指导原则如下:

- 如果组成应用的函数类型单一, 且函数执行时间接近, 则可以使用 FaaSchedule 的 FCFS 变体, 以最小化调度的开销;
- 如果组成应用的函数类型单一, 但函数执行时间范围较大, 则可以考虑使用 FaaSchedule 的 SJF 变体, 以优化某个函数负载处理的性能.

表 2 实验设置.

Table 2 Experimental setup.

Name	Information	Name	Information
Processor	Xeon(R) Gold 5117 @ 2.00 GHz, 56(2*28) cores b	DRAM	2 TB, INTEL SC2KB019T8
SSD	256 GB, DDR4, 2666 MT/s	Network	1 Gigabit LAN bandwidth
QEMU	6.2.0	Linux kernel	5.15.0
Ubuntu	22.04-TLS	Network	1 Gigabit LAN bandwidth
OpenWhisk	1.0.0	Docker	23.0.5

如果组成应用的函数类型多样,特别是混合了交互性强、延迟敏感的短任务,则可以根据长任务对延迟的敏感度以及与短任务之间的执行时间差距来设定 ε 值:

- 如果长任务对延迟不敏感,则可以将 ε 值设置为 90% 以上,这样在不饿死长任务的情况下,最大限度地优先执行短任务;
- 如果长任务的执行时间远大于短任务,则可以将 ε 值设置为较小的值,这样可以在稍微损害长任务公平性的前提下,使短任务获得显著的性能提升.例如,长短任务执行时间差距超过 3 个数量级时,将 ε 设置为 10%,可以优先运行至少 100 个短任务;
- 如果长任务不允许任何性能损失,则应将 ε 设置为 0,这样可以确保在完全不影响长任务的前提下,优先调度短任务;
- 此外,用户还可以通过离线运行函数,调整 ε 值以拟合应用的性能与公平性之间的关系,从而选择理想值.

6 实验评估

我们通过真实工作负载追踪和无服务器函数基准测试对 FaaSchedule 进行了实验评估.评估的目标是回答以下问题:

- 调度隔离机制是否能有效地支持在 CFS 调度组中实现一个新的调度策略?
- 调度隔离机制是否会引入开销?
- 调度隔离机制的隔离性效果如何?
- FaaSchedule 在处理服务器无感知计算负载时的表现如何?用户是否能自由地定制调度策略?
- FaaSchedule 的调度开销如何?

6.1 实验环境配置

实验系统配置.本文中调度隔离框架相关的实验在配置为 56 核、256 GB 内存的物理服务器上进行测试;而与可定制调度器 FaaSchedule 相关的实验则在由 9 台虚拟机组成的集群上进行,其中 1 台作为控制节点,其余 8 台作为工作节点.我们在该集群上部署了修改版的 OpenWhisk 框架,包含 FaaSchedule 及其所需组件.每个工作节点配置为 32 核、128 GB 内存的虚拟机.表 2 总结了实验系统的配置和软件堆栈.

Benchmark.我们使用 Hackbench 来测试调度隔离相关的实验. Hackbench 是一个用于测试 Linux 调度器的基准测试工具,它模拟了多个进程之间的通信和同步.其工作原理类似于聊天应用程序,可以通过参数设定有多少组实体进行相互通信,以及每个实体通信的循环次数即每个实体发送信息的数量.在服务器无感知计算中,可以将每个实体视为一个函数实例,循环次数或者说信息数量为函数调用时提供的参数.

真实 FaaS 工作负载.我们在实验中使用了 Azure Functions Trace 数据.该追踪记录了超过

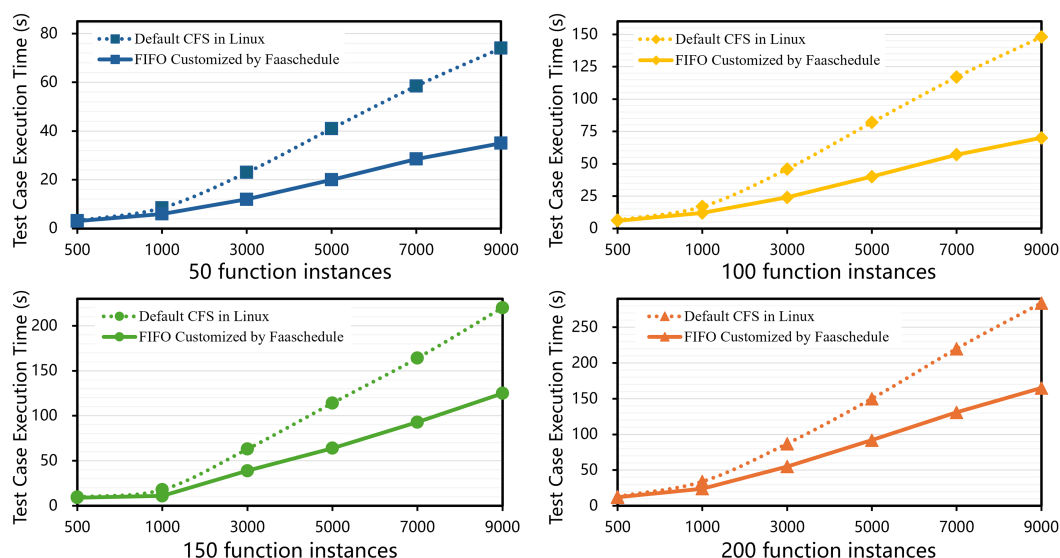


图 5 (网络版彩图) CPU 调度隔离 FIFO 策略与默认 CFS 策略运行结果对比图.

Figure 5 (Color online) Comparison of the running results of the CPU scheduling isolation FIFO strategy and the default CFS strategy.

50000 个唯一函数的执行时长、内存大小和调用时间戳, 时间跨度为 14 天. 由于整个追踪数据是在微软的云平台上生成的, 无法在我们的实验环境中复制此规模, 因此选择了一个较小的样本, 符合我们集群的规模. 选择这些样本至关重要, 因为它们需要充分反映不同类型无服务器函数的特征, 以证明 FaaSchedule 的通用性和工作负载无关性. 我们根据执行时长为标准, 从数据集中的每个四分位数中抽取函数样本, 共计 200 个函数样本. 该数据集包含短任务和长任务, 可以测试 FaaSchedule 在典型工作场景中的性能.

6.2 调度隔离有效性

为了展示调度隔离机制的有效性, 我们使用 FaaSchedule 调度器的 FCFS 变体 (见 5.5 小节) 作为局部调度器的调度策略, 然后与 Linux 原生的 CFS 调度器进行比较. 图 5 展示了函数实例数量配置为 50, 100, 150, 200, 函数实例发送信息数量逐步递增下的运行结果. 结果表明, 不同函数实例数配置下应用受策略配置的影响趋势基本一致, 即伴随着函数实例数量的增加, 默认的 CFS 调度策略以及经过调度隔离后实现的 FIFO 调度策略的运行时间展现出的一致增加趋势. 并且 CPU 调度隔离条件下, 局部调度器中采用的 FIFO 策略可以明显提升应用的执行效率, 并且伴随着每个函数实例发送信息的数量即函数执行时间的增加, 调度隔离的 FIFO 策略的性能提升更加明显, 这证明了 CPU 调度隔离的有效性和必要性.

为了更直观地展示 CPU 调度隔离中独立任务调度带来的性能提升, 我们将不同配置下的函数实例运行的结果进行标准化处理, 即展示 FIFO 策略和 CFS 策略下的运行时间之比. 如图 6 所示, 结果表明 CPU 调度隔离提供的 FIFO 在 CPU 调度隔离中显著提升了应用的执行效率. 从实验结果来看, 随着每个函数实例负载请求的增加, FIFO 下的运行时间相较于默认模式呈现出更低的比值, 表明 FIFO 策略的优势随着负载的增大而逐渐明显. 具体而言, 当函数实例数分别为 50, 100, 150 和 200 时, 性能优化分别达到了 54%, 48%, 44% 和 41%. 在各个配置下, 平均性能提升分别为 39%, 37%, 34% 和 32%, 整体平均性能提升为 35%.

然而, FIFO 的优化效果会随着函数实例数量的增加而减少. 可以观察到相对于函数实例数配置为 200 的情况下, FIFO 在函数实例数配置为 50 时带来的优化比例更明显. 其主要原因是整个系统中可运行任务的数量会随着实体数量增多而显著增加. 这种情况下, FIFO 尽可能久地占用 CPU 资源的

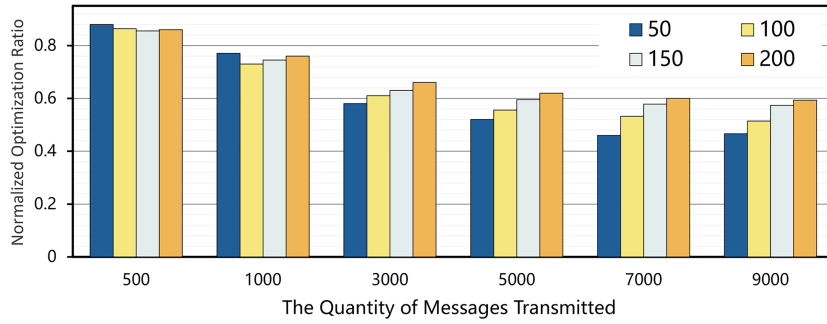


图 6 (网络版彩图) CPU 调度隔离 FIFO 策略与默认 CFS 策略对比的归一化优化结果图。

Figure 6 (Color online) Normalized optimization result diagram comparing CPU scheduling isolation FIFO strategy and default CFS strategy.

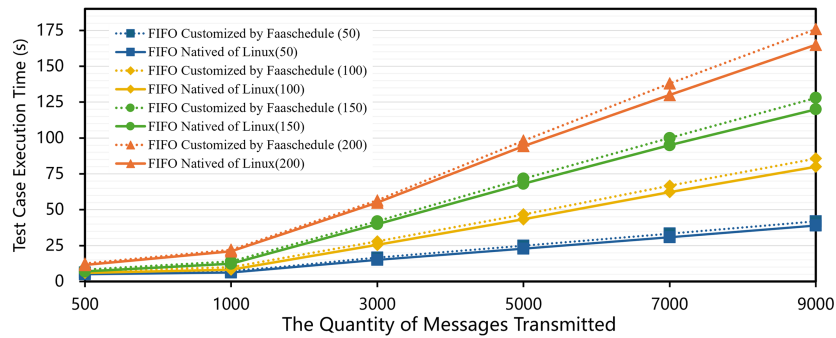


图 7 (网络版彩图) 局部调度器上 FaaSchedule 定制化的 FIFO 与 Linux 的实时调度类中 SCHED_FIFO 的性能对比。

Figure 7 (Color online) Performance comparison between FaaSchedule customized FIFO on the local scheduler and SCHED_FIFO in the Linux real-time scheduler.

调度特性会使得队列后部任务的等待时间过长, 导致 FIFO 的性能优势在可运行任务较多时降低. 这样的结果证明了, 单一的调度策略无法满足所有特定场景需求, 为每个应用提供独立的调度策略是必要的。

6.3 调度隔离的性能开销

为了评估 CPU 调度隔离的性能开销, 我们使用了 FaaSchedule 调度器的 FCFS 变体作为局部调度器的调度策略, 并将其与 Linux 原生的 FIFO 调度策略进行比较. 测试结果如图 7 所示. Linux 原生的 FIFO 调度策略属于实时调度类, 其设计初衷是为了支持严格的实时性需求. 然而, 这种设计也带来了一个显著问题, 即实时调度策略在全局范围内具有较高的优先级, 可能导致普通任务无法及时获得 CPU 资源, 进而影响整个系统的公平性和响应时间. 因此, FIFO 并不适合在复杂的、多任务运行环境中进行定制, 因为它会干扰全局调度, 造成不可控的系统行为。

我们的调度隔离方案允许在同一主机上并行运行多种调度策略, 同时确保这些策略在隔离的环境中运行, 不会像 Linux 的实时调度类那样对全局调度造成影响. 这样一来, 我们既可以灵活地满足特定任务的需求, 又能保障系统的整体性能和稳定性. 经过对比实验, 我们发现即使在引入调度隔离的情况下, FaaSchedule 的 FCFS 变体与 Linux 原生的 FIFO 调度策略在性能表现上几乎没有差别. 在函数实例数配置为 50, 100, 150, 200 的情况下, 局部 FIFO 策略的性能开销分别为 0.5%, 0.6%, 0.7% 和 0.8%, 平均性能开销仅为 0.65%. 这些数据表明, 调度隔离对性能的影响可以忽略不计, 同时还能能为不同类型的任务提供更灵活、优化的调度服务。

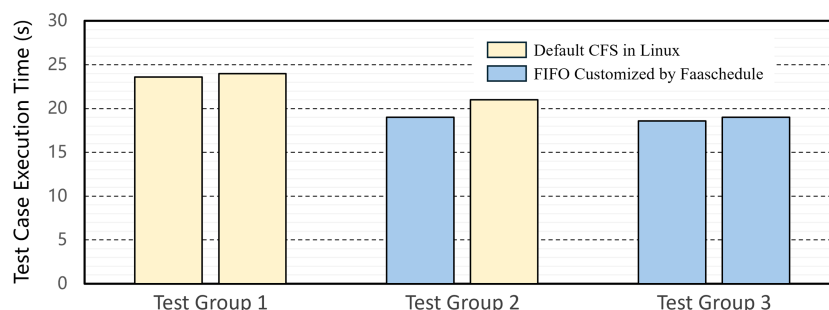


图 8 (网络版彩图) 验证调度隔离是否能有效地隔离两个不同局部调度器。

Figure 8 (Color online) Verify that scheduling isolation can effectively isolate two different local schedulers.

6.4 调度隔离的效果验证

本实验旨在验证调度隔离是否能有效地隔离两个不同局部调度器。我们使用同时运行的两个局部调度器进行测试,并根据每个调度器的调度策略分为如下 3 组: (1) 两个调度器均使用 Linux 原生的 CFS 调度策略; (2) 一个调度器使用 FaaSchedule 定制的 FIFO 调度策略,另一个调度器使用 Linux 原生的 CFS 调度策略; (3) 两个调度器均使用 FaaSchedule 定制的 FIFO 调度策略。每个测试组中的两个调度器分别运行相同的函数实例,但是每个函数实例的处理参数不同。测试结果如图 8 所示。我们可以看到,当两个调度器均使用 FaaSchedule 定制的 FIFO 调度策略时,两个调度器之间的性能表现几乎没有差别。将一个(测试组 2)或两个(测试组 3)局部调度器配置为 FIFO 能够提高容器的运行效率。其中测试组 3 对比测试组 1 (默认的 CFS 策略) 能够获得约 20% 的性能提升。在 3 组对比实验中,我们对两个局部调度器的策略实现了差异化配置,并获得了不同的运行结果。这验证了 CPU 隔离方案能够实现并发容器的独立任务调度。

6.5 FaaSchedule 对服务器无感知应用的性能提升

为了评估 FaaSchedule 在服务器无感知计算负载中的性能,我们使用了 Azure Functions Trace 数据集,并与多种调度算法进行了对比,包括 Linux 内核中默认的 CFS 调度算法、先入先出 (FIFO) 调度算法以及最短作业优先 (SJF) 算法。为展示 FaaSchedule 的可定制性,我们将 ε 参数设置为 0~1 的不同值,其中 $\varepsilon = 0$ 表示完全追求公平性,而 $\varepsilon = 1$ 表示完全追求性能。不同的 ε 值帮助用户在公平性和性能之间找到最佳平衡。实验结果如图 9 所示。

在完全暖启动的情况下, SJF 和 FaaSchedule ($\varepsilon = 1$) 的执行时间相同,均达到最高性能。这是因为在完全暖启动环境下, SJF 能精确选择最短任务,而 FaaSchedule ($\varepsilon = 1$) 的行为与 SJF 相同。当 FaaSchedule 更注重公平性 (ε 趋近于 0) 时,其性能有所下降。然而,即便在 $\varepsilon = 0$ 时, FaaSchedule 的性能依然优于其他调度算法。与 CFS 和 FIFO 相比, FaaSchedule ($\varepsilon = 0$) 分别减少了 13% 和 23% 的执行时间。这是因为 FaaSchedule 在不影响长任务完成时间的前提下优先调度短任务,从而降低了平均响应时间。

完全暖启动是一种理想情况,但在实际的服务器无感知平台中,冷启动不可避免。为此,我们将暖实例的缓存命中率设置为 90%,并在该环境下评估 FaaSchedule 的性能。结果如图 9(b) 所示。即使在 $\varepsilon = 0$ 的情况下, FaaSchedule 的性能仍然超过了理论最优的 SJF。与 CFS, FIFO 和 SJF 相比, FaaSchedule ($\varepsilon = 0$) 分别减少了 75%, 75% 和 56% 的执行时间。这是因为 FaaSchedule 在调度时充分考虑了冷启动开销对任务执行时间的影响。

此外,无论在何种情况下,随着 ε 减小, FaaSchedule 的性能均有所提升。这进一步证明了其可定制性和灵活性,使用户能够根据实际需求,在提升整体性能的同时,平衡对长任务的公平性。

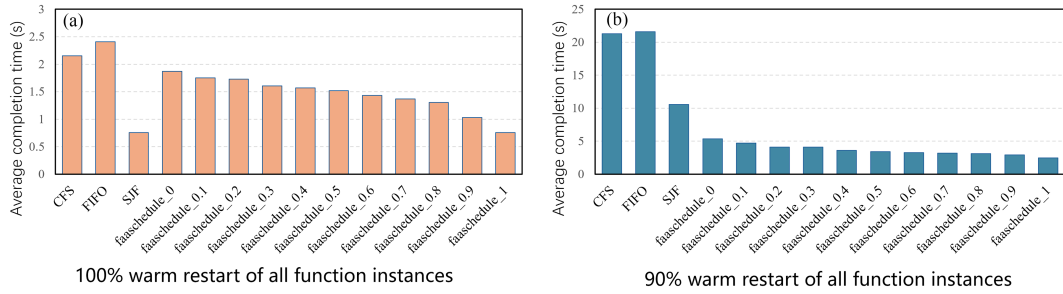


图 9 (网络版彩图) 在完全暖启动 (a) 和暖实例命中率为 90% (b) 的情况下, FaaSchedule 与其他调度算法的性能对比.

Figure 9 (Color online) Performance comparison of FaaSchedule with other scheduling algorithms in the case of (a) full warm start and (b) warm instance hit rate of 90%.

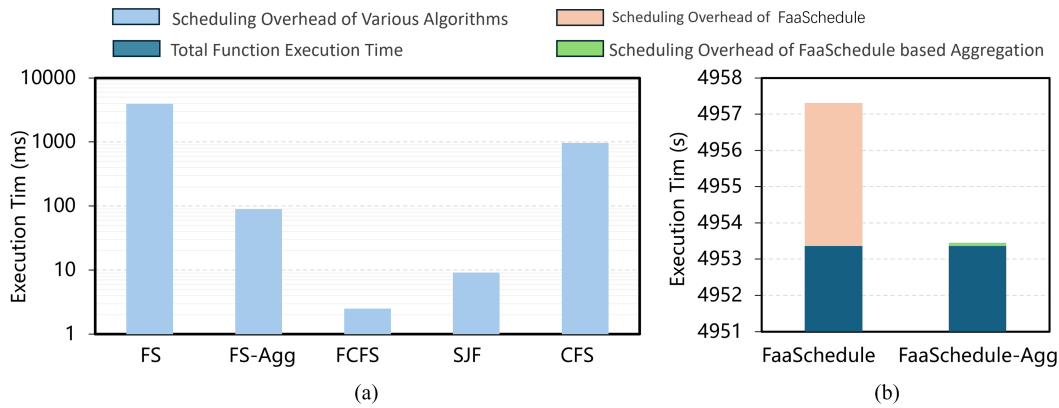


图 10 (网络版彩图) FaaSchedule 调度开销. (a) 不同调度算法的开销对比; (b) 调度开销与执行时间的堆叠图.

Figure 10 (Color online) FaaSchedule scheduling overhead. (a) Comparison of overheads among various scheduling algorithms; (b) stacked graph of scheduling overhead and execution time.

6.6 FaaSchedule 的调度开销

为了评估 FaaSchedule 的调度开销, 我们测试了在不同调度算法下处理完全相同工作负载的用时. 测试的算法包括 FaaSchedule, 使用 5.4 小节中提到的聚合优化方式改进的 FaaSchedule, CFS, FCFS 和 SJF 等. 测试结果如图 10 所示. 从图中可以看出, FCFS 算法的调度开销最小, 这是因为它仅需要一个简单的先进先出队列, 而无需进行其他操作. SJF 的性能仅次于 FCFS, 因为它需要在每次调度时选择最短的任务, 进而对所有任务进行排序. FaaSchedule 的调度开销最大, 这是因为每当有新任务到达时, 系统需要更新所有队列中的任务优先级并重新排序, 且优先级的计算涉及浮点运算, 造成了较高的开销. 幸运的是, 经过聚合优化的 FaaSchedule 调度开销大幅降低, 这是由于优化后的 FaaSchedule 只需要在函数级别维护优先级排序队列, 而不再在每个请求级别进行维护, 从而显著减少了维护的复杂度. 与原始 FaaSchedule 相比, 聚合优化后的 FaaSchedule 调度开销减少了 97.74%. 值得注意的是, 优化后的 FaaSchedule 的开销远低于 CFS, 这主要是因为 CFS 每个时钟周期 (通常为 1 ms) 都会执行一次调度, 而其他调度算法仅在任务到达或任务完成时进行调度.

尽管原生 FaaSchedule 引入了最高的调度开销, 但相较于工作负载的总执行时间, 这一开销仍是可以接受的. 如图 10(b) 所示, 原生 FaaSchedule 的调度开销仅占工作负载总执行时间的 0.08%, 而聚合优化后的 FaaSchedule 的开销更为微小, 仅占不到 0.01%. 这表明, FaaSchedule 的调度开销在整体上是接受的, 尤其是在经过优化后, FaaSchedule 的调度开销几乎可以忽略不计.

7 总结与展望

服务器无感知计算凭借其弹性、高效性和低成本已成为云计算的关键组成部分. 然而, 现有的统一调度策略难以满足多样化应用的需求, 尤其是在任务执行时间差异和冷启动影响下. 为解决这一问题, 本文提出了 FaaSchedule 调度策略, 通过引入调度隔离机制和策略定制, 显著降低了短任务的完成时间, 并能够根据不同应用需求动态调整调度策略. 实验结果表明, FaaSchedule 有效地支持了任务调度的隔离性与定制性. 展望未来, 调度策略的定制将不再仅限于任务的长短, 而是将更加灵活地考虑多维度的应用需求, 以进一步优化云计算平台的整体性能与公平性.

参考文献

- Jonas E, Schleier-Smith J, Sreekanti V, et al. Cloud programming simplified: a Berkeley view on serverless computing. 2019. ArXiv:1902.03383
- Shahrad M, Balkind J, Wentzlaff D. Architectural implications of function-as-a-service computing. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, New York, 2019. 1063–1075
- Wang L, Li M, Zhang Y, et al. Peeking behind the curtains of serverless platforms. In: Proceedings of the USENIX Conference on Usenix Annual Technical Conference, Boston, 2018. 133–146
- Oakes E, Yang L, Zhou D, et al. SOCK: rapid task provisioning with serverless-optimized containers. In: Proceedings of the USENIX Conference on Usenix Annual Technical Conference, Boston, 2018. 57–70
- Akkus I E, Chen R, Rimac I, et al. SAND: towards high-performance serverless computing. In: Proceedings of the USENIX Conference on Usenix Annual Technical Conference, 2018. 923–935
- Hendrickson S, Sturdevant S, Harter T, et al. Serverless computation with OpenLambda. In: Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing, 2016
- Du D, Yu T, Xia Y, et al. Catalyzer: sub-millisecond startup for serverless computing with initialization-less booting. In: Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, 2020. 467–481
- Ustiugov D, Petrov P, Kogias M, et al. Benchmarking, analysis, and optimization of serverless function snapshots. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021. 559–572
- Werner S, Kuhlenskamp J, Klems M, et al. Serverless big data processing using matrix multiplication as example. In: Proceedings of IEEE International Conference on Big Data, 2018. 358–365
- Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: Proceedings of USENIX Annual Technical Conference, 2014. 305–319
- Jonas E, Pu Q, Venkataraman S, et al. Occupy the cloud: distributed computing for the 99. In: Proceedings of the Symposium on Cloud Computing, 2017. 445–451
- Carreira J, Fonseca P, Tumanov A, et al. A case for serverless machine learning. In: Proceedings of Workshop on Systems for ML and Open Source Software at NeurIPS, 2018
- Carreira J, Fonseca P, Tumanov A, et al. Cirrus: a serverless framework for end-to-end ML workflows. In: Proceedings of the ACM Symposium on Cloud Computing, 2019. 13–24
- Ishakian V, Muthusamy V, Slominski A. Serving deep learning models in a serverless platform. In: Proceedings of IEEE International Conference on Cloud Engineering (IC2E), 2018. 257–262
- Shankar V, Krauth K, Vodrahalli K, et al. Serverless linear algebra. In: Proceedings of the 11th ACM Symposium on Cloud Computing, 2020. 281–295
- Kaffes K, Yadwadkar N J, Kozyrakis C. Hermod: principled and practical scheduling for serverless functions. In: Proceedings of the 13th Symposium on Cloud Computing, 2022. 289–305
- Jin X, Wu B Y, Liu F Y, et al. Exploiting temporal-spatial characteristics for function scheduling in serverless computing. J Comput Res Dev, 2023, 60: 2000–2014 [金鑫, 吴秉阳, 刘方岳, 等. 服务器无感知计算场景下基于时空特征的函数调度. 计算机研究与发展, 2023, 60: 2000–2014]
- Banaei A, Sharifi M. ETAS: predictive scheduling of functions on worker nodes of Apache OpenWhisk platform. J Supercomput, 2022, 78: 5358–5393
- Kim J, Lee K. Functionbench: a suite of workloads for serverless cloud function service. In: Proceedings of the 12th International Conference on Cloud Computing (CLOUD), 2019. 502–504

- 20 Copik M, Kwasniewski G, Besta M, et al. SeBS: a serverless benchmark suite for function-as-a-service computing. In: Proceedings of the 22nd International Middleware Conference, 2021. 64–78
- 21 Ao L, Izhikevich L, Voelker G M, et al. Sprocket: a serverless video processing framework. In: Proceedings of the ACM Symposium on Cloud Computing, 2018. 263–274
- 22 Fuerst A, Sharma P. FaaS-Cache: keeping serverless computing alive with greedy-dual caching. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021. 386–400
- 23 Epema D H J. An analysis of decay-usage scheduling in multiprocessors. In: Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, 1995. 74–85
- 24 Bansal N, Harchol-Balter M. Analysis of SRPT scheduling. SIGMETRICS Perform Eval Rev, 2001, 29: 279–290
- 25 Harchol-Balter M, Bansal N, Schroeder B, et al. SRPT scheduling for web servers. In: Proceedings of the 7th International Workshop on Job Scheduling Strategies for Parallel Processing, 2001. 11–20
- 26 Al-Husainy M A F. Best-job-first CPU scheduling algorithm. Inf Tech J, 2007, 6: 288–293
- 27 Karthick A, Ramaraj E, Subramanian R G. An efficient multi queue job scheduling for cloud computing. In: Proceedings of World Congress on Computing and Communication Technologies, 2014. 164–166
- 28 Fu Y, Liu L, Wang H, et al. SFS: smart OS scheduling for serverless functions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2022
- 29 Asyabi E, Sharafzadeh E, SanaeeKohroudi S A, et al. CTS: an operating system CPU scheduler to mitigate tail latency for latency-sensitive multi-threaded applications. J Parall Distrib Comput, 2019, 133: 232–243
- 30 Qian J, Srisa-An W, Seth S, et al. Exploiting FIFO scheduler to improve parallel garbage collection performance. In: Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2016. 109–121
- 31 Leverich J, Kozyrakis C. Reconciling high server utilization and sub-millisecond quality-of-service. In: Proceedings of the 9th European Conference on Computer Systems, 2014. 1–14
- 32 Ousterhout A, Fried J, Behrens J, et al. Shenango: achieving high CPU efficiency for latency-sensitive datacenter workloads. In: Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, 2019. 361–378
- 33 Kaffes K, Chong T, Humphries J T, et al. Shinjuku: preemptive scheduling for usecond-scale tail latency. In: Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, 2019. 345–360
- 34 Prekas G, Kogias M, Bugnion E. ZygOS: achieving low tail latency for microsecond-scale networked tasks. In: Proceedings of the 26th Symposium on Operating Systems Principles, 2017. 325–341
- 35 Zhao S, Hao C L, Zhai J, et al. S-Bridge: CPU load balancing agent for performance asymmetric multicore processors. J Softw, 2020, 31: 2965–2979 [赵姗, 郝春亮, 翟健, 等. S-Bridge: 性能非对称多核处理器下负载均衡代理机制. 软件学报, 2020, 31: 2965–2979]
- 36 Suresh A, Somashekar G, Varadarajan A, et al. ENSURE: efficient scheduling and autonomous resource management in serverless environments. In: Proceedings of IEEE International Conference on Autonomic Computing and Self-Organizing Systems, 2020. 1–10
- 37 Suresh A, Gandhi A. FnSched: an efficient scheduler for serverless functions. In: Proceedings of the 5th International Workshop on Serverless Computing, 2019. 19–24
- 38 Yu H, Irissappane A A, Wang H, et al. FaaSRank: learning to schedule functions in serverless platforms. In: Proceedings of IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), 2021. 31–40
- 39 Carver B, Zhang J, Wang A, et al. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In: Proceedings of the 11th ACM Symposium on Cloud Computing, 2020. 1–15
- 40 Tariq A, Pahl A, Nimmagadda S, et al. Sequoia: enabling quality-of-service in serverless computing. In: Proceedings of the 11th ACM Symposium on Cloud Computing, 2020. 311–327
- 41 Singhvi A, Balasubramanian A, Houck K, et al. Atoll: a scalable low-latency serverless platform. In: Proceedings of the ACM Symposium on Cloud Computing, 2021. 138–152
- 42 Mahgoub A, Yi E B, Shankar K, et al. ORION and the three rights: sizing, bundling, and prewarming for serverless DAGs. In: Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation, 2022. 303–320
- 43 Schroeder B, Harchol-Balter M. Web servers under overload: how scheduling can help. Teletraffic Sci Eng, 2003, 5: 171–180
- 44 Wierman A, Zwart B. Is tail-optimal scheduling possible? Oper Res, 2012, 60: 1249–1257

Customizable function scheduling for serverless computing

Xinmin ZHANG^{1,2,3,4}, Xingru LI^{1,2,3,4}, Hao FAN^{1,2,3,4*}, Zhuo HUANG^{1,2,3,4}, Song WU^{1,2,3,4},
Dezhong YAO^{1,2,3,4}, Hai JIN^{1,2,3,4} & Chen YU^{1,2,3,4}

1. *National Engineering Research Center for Big Data Technology and System, Huazhong University of Science and Technology, Wuhan 430074, China*

2. *Key Laboratory of Services Computing Technology and System, Ministry of Education, Huazhong University of Science and Technology, Wuhan 430074, China*

3. *Key Laboratory of Cluster and Grid Computing, Hubei Province, Huazhong University of Science and Technology, Wuhan 430074, China*

4. *School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China*

* Corresponding author. E-mail: haofan@hust.edu.cn

Abstract Serverless computing, with its advantages of elasticity, efficiency, and low cost, has become a key component of next-generation cloud computing. As the complexity and diversity of serverless applications increase, existing serverless platforms typically deploy function instances from different applications on the same server and apply a unified CPU scheduling policy, such as Linux's completely fair scheduler (CFS). However, due to the varying requirements of different applications in terms of performance, latency, and scheduling fairness, a single global scheduling policy struggles to meet diverse scheduling demands. Moreover, our research reveals that function instances exhibit a wide range of execution times, with a high proportion of short tasks, and that cold starts significantly impact task completion times. These characteristics make current scheduling policies ineffective in balancing efficiency and fairness, especially increasing the scheduling latency for short tasks. To address these issues, we propose and implement a scheduling isolation mechanism that allows single or multiple applications with similar requirements to adopt independent CPU scheduling policies on the same server. Additionally, we design FaaSchedule, a novel customizable scheduling policy that takes into account function execution time, startup time, and waiting time, and supports dynamic customization of scheduling strategies based on the needs of different applications. Experiments on the OpenWhisk platform show that FaaSchedule significantly reduces the average completion time of short tasks with minimal impact on the performance of long tasks, while effectively supporting scheduling isolation and policy customization.

Keywords CPU scheduling, serverless computing, cold start, scheduling isolation, customized kernel