计 算 机 研 究 与 发 展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202221043
61(9): 2384−2399, 2024

面向高密度混部的动态资源分配方法

郭 静 胡存琛 包云岗
(处理器芯片全国重点实验室(中国科学院计算技术研究所) 北京 100190)
(中国科学院大学 北京 100190)
(guojing171@mails.ucas.ac.cn)

A Dynamic Resource Allocation Method for High-Density Colocation Scenario

Guo Jing, Hu Cunchen, and Bao Yungang
(*State Key Lab of Processors*(*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)
(*University of Chinese Academy of Sciences*, *Beijing* 100190)

Abstract Current serverless computing providers use a coupled resource allocation strategy with low flexibility and a fixed CPU-to-memory allocation ratio. As more types of functions are deployed to the serverless computing platform, the coupled strategy can not satisfy the wide range of resource requirements for these functions. Due to the small granularity of resource allocation and high deployment density in serverless functions, if CPU and memory resource allocation are decoupled, the problem of resource configuration space explosion needs to be solved. In this paper, we present Semi-Share, a decoupled resource manager for serverless functions, which can find the optimal resource configurations for functions while reducing the interference between co-located functions. To solve the resource configuration space explosion problem, Semi-Share builds a two-layer resource allocation architecture, which divides the resource configuration space into multiple subspaces to reduce problem complexity. The first layer is the function cluster, which is based on the resource preference and historical load information of the functions. The resource configuration space is divided according to these clusters. The second layer is resource allocation, which leverages the Bayesian optimization and weighted scoring function to guide Semi-Share to search in the right direction in the configuration space and reduce the time overhead. The experimental results show that Semi-Share greatly reduces the search time of searching the optimal resource configuration by using the two-layer architecture, reduces the average configurations sample by 85.77% compared with the widely used gradient descent search method, and improves the function performance by 42.72% on average. Compared with COSE, a coupled resource allocation system that also uses Bayesian optimization, Semi-Share can improve the function performance by 32.25% on average.

Key words serverless computing; colocation; performance guarantee; quality of service; resource allocation; high deployment density

摘 要 当前的无服务计算提供商采用了一种灵活度低、固定CPU和内存分配比例的耦合式资源分配策略.随着更多类型应用被部署在无服务计算平台中,该策略已无法满足函数应用的多样化资源需求.由于函数应用的资源分配粒度小、部署密度高,若将CPU与内存资源的分配进行解耦,需解决资源配置空间爆炸问题.提出Semi-Share,一个面向无服务计算的解耦式资源管理系统,为函数寻找最优资源配置的同时降低混部函数之间的干扰.为解决资源配置空间爆炸问题,Semi-Share构建了一个2层资源分配架构,

收稿日期: 2022-12-30; **修回日期**: 2023-12-20
基金项目: 广东省重点研发计划项目(2020B010164003)
This work was supported by the Key Research and Development Program of Guangdong Province (2020B010164003).
通信作者: 包云岗(baoyg@ict.ac.cn)

将资源配置空间划分为多个子空间来降低问题复杂度. 第 1 层是函数分组，基于函数的资源使用特征和历史负载信息进行函数分组，根据分组将资源配置空间划分为多个子空间. 第 2 层是资源分配，利用贝叶斯优化和加权打分函数来指导模型在资源配置空间中朝正确的方向搜索，降低时间开销. 实验结果显示，Semi-Share 相较于被广泛使用的梯度下降搜索法降低了平均 85.77% 的时间开销，并为函数带来平均 42.72% 的性能提升；与同样使用贝叶斯优化的耦合式资源分配系统 COSE 相比，Semi-Share 能带来平均 32.25% 的性能提升.

关键词 无服务计算；混部；性能保障；服务质量；资源分配；高部署密度

中图法分类号 TP316

无服务计算因其可扩展性高、成本效率高且无需运维等优势吸引了越来越多的用户，各种类型的函数应用被部署在无服务计算平台上，例如 Web 服务、文本处理、机器学习推理、大数据处理等函数应用. 这些函数应用以容器、虚拟机或轻量级沙盒的形式在物理节点上混部运行，共享物理资源.

在资源分配方面，当前的无服务提供商采用了一种耦合式资源分配策略. 该策略将函数应用的 CPU 资源分配与内存资源按固定比例绑定在一起，无服务平台根据用户设置的内存大小按比例为函数分配 CPU 资源，而不需用户指定具体的 CPU 资源大小. 各无服务计算提供商的资源分配策略如表 1 所示，例如，在 AWS Lambda 中，当用户给函数配置 1 769 MB 内存时，相当于为函数分配了 1 vCPU 资源，其 CPU 资源与内存资源的分配比例为 1∶1 769. 耦合式资源分配策略的优点在于能简化资源分配过程和计费模型，然而，这种将 CPU 资源和内存资源耦合在一起的资源分配策略并不适合于计算密集型函数应用[6]，用户为给该类函数分配足够的 CPU 资源，只能选择为函数分配过量的内存资源，这导致了资源使用效率低下. 此外，为函数过量分配内存资源会限制进一步提升 CPU 资源使用效率[7]. 因此，随着更多类型函数应用被部署在无服务计算平台，有必要将 CPU 资源与内存资源的分配进行解耦，以适应函数更广泛的资源需求.

Table 1 Resource Allocation Strategy of the Commercial Serverless Platform

表 1 商业无服务计算平台的资源分配策略

服务提供商	资源分配策略	资源分配范围
AWS Lambda[1]	根据内存配置大小按比例分配 CPU 资源	1 769 MB = 1 vCPU，128~10 240 MB
Google Function[2]	9 种固定的资源配置选项	128 MB/0.083 vCPU~32 768 MB/8 vCPU
IBM Cloud Function[3]	根据内存配置大小按比例分配 CPU 资源	128~2 048 MB
Azure Function[4]	CPU 资源数量固定，内存自行配置	100 ACU，最大内存为 1.5 GB
腾讯云函数[5]	根据内存配置大小按比例分配 CPU 资源	1 280 MB = 1 vCPU，64 MB ~ 3 GB

若将 CPU 资源的分配从内存资源中解耦出来，并根据函数的真实资源需求为其分配资源，需满足函数应用对资源的 2 点需求：合适的资源数量和较高的资源质量，即要决定为函数分配多少资源，并且共享同一资源的函数间的干扰要尽可能小(资源质量). 由于无服务函数直接面向用户，且根据函数的执行时间进行计费，因此在进行资源分配时还需考虑资源配置对函数性能的影响. 其中，性能指的是函数应用处理请求的延迟[8-12]. 基于函数应用的部署密度高、资源需求粒度小和种类多样等特性，设计灵活度更高的资源分配策略并提供性能保障，需解决 2 个挑战.

第一，要解决将 CPU 与内存资源解耦后资源配置空间爆炸问题. 在无服务计算场景下，函数应用的资源分配粒度变得更小. 例如，CPU 资源的分配粒度从以 CPU 核为单位进行分配，转变为以 CPU 时间片为单位进行分配[8]；内存资源的分配粒度从以 GB 为单位，转变为以 MB 为单位进行分配. 因此，以 CPU 和内存资源为主的资源配置空间变得更大，在设计解耦式资源分配策略时需考虑 2 个问题：1)将 CPU 资源的分配与内存资源解耦后，应为函数分配多少 CPU 资源. 2)如何降低搜索最优资源配置的时间开销. 当前的资源管理系统[9-11]均是面向单体式应用或微服务的低密度部署场景，且资源分配粒度较大，若将其直接应用到无服务计算场景，会导致资源配置搜索时间和模型训练的时间开销增大.

第二，要解决高密度部署下函数应用的性能保障问题. 在无服务计算场景下，函数应用的部署密度变得更高. 部署密度是指单位资源上部署的应用个数，在无服务计算场景下，1 个 CPU 核上会部署多个函数应用[13-14]，应用间处于高密度共享状态. 这种高密度部署使得函数应用数量与资源数量出现了不匹配现象. 例如，函数应用的数量远大于物理机上 CPU

核和末级缓存(last level cache, LLC)的数量，这使得多个函数不得不去共享有限的资源. 虽然这种更高的部署密度和更细粒度的资源共享带来了较高的经济效益，但却使函数对共享资源的竞争变得更复杂. 因此，为保障混部函数的性能，在进行资源分配时还需考虑哪些函数应用共享同一核内资源.

本文的目标是在无服务计算的高密度混部场景下，为函数提供灵活度更高的解耦式资源分配策略，在保障函数应用性能的同时提升资源使用效率. 为实现该目标，需将 CPU 资源的分配从内存资源中解耦，并解决 2 个挑战：第一，降低时间开销，在资源配置空间中快速搜索最优资源配置；第二，降低函数间干扰，在搜索资源配置的同时，降低函数对共享资源的竞争.

本文提出 Semi-Share，其含义为混部函数之间对于资源处于一种半共享的状态，它是一个面向无服务计算的解耦式资源管理系统，为函数搜索最优资源配置的同时降低混部函数之间的干扰. 为解决高密度混部场景下资源配置空间爆炸的问题，Semi-Share 采用分而治之的思想. 它没有直接在整个资源配置空间中搜索，而是将多维资源划分问题分解为多个子问题来降低问题复杂度. 由于函数应用的生命周期较短，为降低搜索全局最优解的时间开销，Semi-Share 采用了一种折中的方式，通过仅为每个子问题求解最优解，来获取全局的近似最优解.

为划分资源配置空间，Semi-Share 构建了一个 2 级资源分配架构：第 1 层是函数分组，基于函数的资源使用特征和历史负载信息进行函数分组，分组个数为节点上可运行函数应用的 CPU 核数量，根据分组将资源配置空间划分为多个子空间. 第 2 层是资源分配，利用贝叶斯优化模型和加权打分函数来指导模型在资源配置空间中朝正确的方向搜索，降低时间开销. 在为每个函数搜索到最优资源配置后，Semi-Share 启动一个资源反馈调节程序持续为函数提供性能保障.

本文的主要贡献有 3 个方面：

1)构建了一个 2 级资源分配架构，解决 CPU-内存资源解耦后的资源配置空间爆炸问题. 本文基于函数的特征信息，将高密度共享下多资源分配的优化问题分解为多个子问题，通过分别求解子问题的最优解，确保获得全局的近似最优解.

2)构建了一个基于函数应用资源使用特征以及负载调用特征的函数分组模型，解决资源分配数量与函数应用数量的不匹配问题. 通过函数分组，组内的函数共享片上资源，不同组的函数使用隔离的资源，以此缓解由于共享资源的竞争而造成的性能干扰.

3)构建了一个基于贝叶斯优化的资源分配模型和分段加权打分函数，以降低模型的训练开销以及搜索最优资源配置的时间开销. 分段加权打分函数不仅能涵盖多种混部场景，形式化表示 2 类函数应用的优化目标，而且能使模型在资源配置空间中朝正确的方向进行搜索，减少搜索过程的采样次数.

1 背景及动机

耦合式资源分配策略的优点是实现简单、方便构建计费模型，但因其 CPU 资源的分配与内存资源进行绑定，资源分配的灵活性较低. 本节从函数应用的混部特征和资源使用特征出发，讨论 3 个问题：

1)现阶段的耦合式资源分配策略能否满足函数应用的资源需求；

2)设计解耦式资源分配策略时会面临哪些挑战；

3)相关工作能否适用于高密度混部场景.

1.1 资源配置的灵活性

函数应用的种类和业务逻辑多样，其调用特征和资源使用偏好也各不相同. 函数应用对资源的使用偏好主要由 2 个因素决定：1)业务逻辑. 函数应用的逻辑功能和实现方式决定了应用的行为和资源使用偏好，如 CPU 密集或访存密集等. 2)负载强度. 函数应用的负载强度可用每秒请求数(request per second, RPS)表示，RPS 的高低决定了函数应用在单位时间内处理请求时对资源的需求度，且一般情况下应用的 RPS 越高其对资源的需求度越大. 因此，应根据函数应用的特征和负载情况，对其真实资源需求进行分析判断后再进行资源分配.

耦合式资源分配策略假设函数应用的 CPU 与内存资源需求比相同，将这 2 种资源的分配按固定比例绑定. AWS Lambda 的资源分配比例(简称：AWS 比例)如图 1 中斜线所示，在 2 维的资源配置空间中为一条固定斜率的斜线，斜率为 CPU 和内存的分配比例，这些资源配置只占整个 2 维资源配置空间的很小一部分. 并且，函数应用对 CPU 资源和内存资源的需求比例随其负载强度的变化而不同，当函数的 RPS 增加时，其需要更多的 CPU 资源来保障其性能. 图 1 中斜线上方的空间代表函数需要更多的 CPU 资源；当负载强度降低时，函数对 CPU 资源的需求也随之减少，斜线下方的空间代表函数需要较少的 CPU 资源. 因此，耦合式资源分配策略因其灵活度较低，

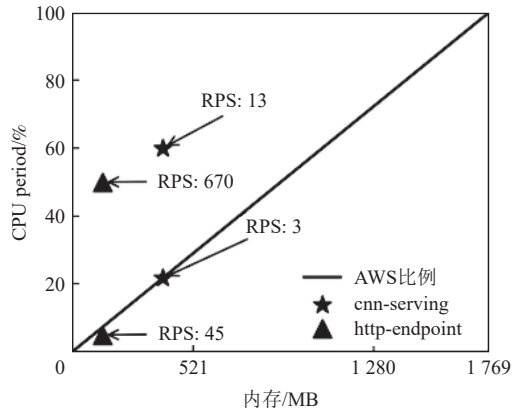


Fig. 1 CPU-memory resource allocation space

图 1 CPU-内存资源配置空间

已无法满足不同类型函数在不同负载强度下的资源需求.

耦合式资源分配策略会导致内存资源的浪费与执行费用的增加. 当用户希望为函数配置更多 CPU 资源时, 耦合式策略只能通过调整内存资源来间接调整 CPU 资源, 这不仅造成了内存资源的浪费, 还会增加执行函数过程中所产生的费用. 例如, 在耦合式和解耦式资源分配策略下, 函数 1 的资源配置分别为⟨60% CPU period, 1 061 MB⟩, ⟨60% CPU period, 384 MB⟩, 函数 2 的资源配置分别为⟨30% CPU period, 531 MB⟩, ⟨30% CPU period, 128 MB⟩. 如图 2 所示, 2 个函数在这 2 种资源分配策略下能获得近似的请求处理延迟, 但成本却相差至少 2 倍, 原因是当前一些函数服务的计费模型是根据函数分配的内存量、函数执行时间等参数制定的[15-18], 为函数分配的内存资源越多, 资源价格也就越高. 此外, 被过度分配的内存资源会限制更多的函数调度到物理机上运行以进一步提升 CPU 资源的使用效率[7].

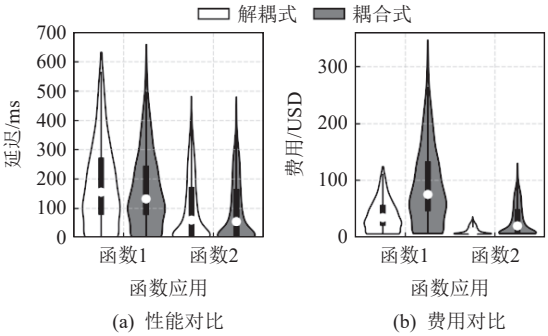


Fig. 2 Comparison of coupled and decoupled resource allocation strategies in performance and cost

图 2 耦合式和解耦式资源分配策略在性能和费用上的对比

因此, 在无服务计算场景下, 耦合式资源分配策略已无法满足函数应用的资源需求. 为提供灵活度更高的资源分配策略, 需要将 CPU 资源的分配与内存资源进行解耦.

1.2 资源分配粒度的变化

函数应用资源分配粒度的变小, 使资源解耦后出现资源配置空间爆炸的问题. 如表 1 所示, 在无服务计算场景下, 函数应用的内存资源分配单位从 GB 变为 MB, 资源分配空间为 64 MB~32 GB 不等, 资源增长幅度变为 1~128 MB 不等; CPU 资源分配单位从 CPU 核变为 CPU Quota, 资源分配空间为 0.072 vCPU~8 vCPU 不等, 资源增长幅度最小为 0.000 57 vCPU. 更小的资源分配单位和较大的资源分配空间, 使得 2 维的 CPU-内存资源配置空间变得更大. 因此, 将资源解耦后, 需考虑为函数搜索最优资源配置的时间开销问题.

1.3 资源分配的不匹配

函数应用大多运行在单个 CPU 核上[14,19], 一些研究专注于在单个 CPU 核上部署更多的函数应用以获得更高的部署密度和经济效益[6,13]. 当多个函数在同一 CPU 核上共享资源时, 会不可避免地产生资源竞争, 造成函数性能的不可预测性. Schall 等人[19] 发现运行在同一 CPU 核上的多个函数由于交替使用 CPU 时间片, 导致了函数在 CPU 核上的微结构数据反复被其他函数替换掉, 从而造成性能损失. Qiu 等人[20] 发现一些函数对于 LLC 和内存带宽资源的竞争很敏感, 能引起 2~20 倍的性能下降. 若为每个函数分配隔离的 CPU 与 LLC 资源, 则会出现应用数量远大于资源数量的情况, 即资源分配的不匹配现象. 由于资源分配的不匹配, 若以 CPU Quota 和 LLC 路为资源分配单位, 需考虑哪些函数应用共享同一 CPU 核内资源和 LLC 资源.

1.4 相关研究工作

一些研究工作聚焦于如何快速地为应用搜索最优资源配置. 根据资源搜索方法不同可将其分为 2 类: 1)基于采样的资源调整法; 2)基于应用性能模型的资源分配法.

基于采样的资源调整法是在资源配置空间中对可能的资源配置逐一进行采样分析, 逐步搜索出最优资源配置的方法. 例如, Heracles[9], PARTIES[10], Rhythm[21] 等利用梯度下降搜索法为应用搜索最优配置. 这些方法类似于穷举搜索法, 适用于资源配置空间较小、资源数量与应用性能是凸函数的场景, 当资源配置与函数性能关系较为复杂时, 这些方法易落入局部最优解.

基于应用性能模型的资源分配法是指, 利用机

器学习等方法构建应用资源配置与性能关系模型，通过模型预测应用在某资源配置下的性能，以此来寻找最优的资源配置. 例如，FIRM[8] 利用强化学习和迁移学习为每个函数训练一个性能模型，当函数出现性能干扰时用模型来预测应为函数分配多少资源. 为每个函数单独训练模型的优点在于模型准确度高、预测结果更精确. 由于 FIRM 需要对每个函数采集运行时数据并训练性能模型，其适用于长时间运行的函数应用. 相较于强化模型和深度神经网络等优化方法需要大量训练数据来提高预测准确度，贝叶斯优化方法仅需有限的采样次数即可找到接近最优的资源配置，并适用于多维资源配置场景. 例如，针对在线应用和大数据处理类应用的资源配置问题，CLITE[11], SATORI[22], CherryPick[23] 根据优化目标，有针对性地构建打分函数，以指导模型的搜索方向. 针对无服务函数应用，COSE[24] 利用贝叶斯优化模型学习函数的预期成本、函数执行时间和资源配置之间的关系，在满足函数的性能或成本约束的前提下为函数选择最合适的内存资源配置和云服务提供商，它适用的场景为单个函数和函数链. COSE 没有将 CPU 资源的分配与内存资源解耦，而是通过指定内存资源大小，并根据云服务提供商的资源分配策略对函数分配 CPU 资源.

随着大型应用被拆分为函数应用，并被部署到无服务计算平台，函数的资源需求更加多样化. 一些研究聚焦于将函数应用的 CPU 和内存资源的分配解耦，根据函数真实的资源需求分配资源. Bilal 等人[25] 发现，将 CPU 和内存资源解耦能降低近 50% 的函数执行开销. Guo 等人[26] 提出一个基于资源解耦的无服务计算框架 Scad，指出无服务计算应以函数实际的资源需求为中心来分配资源，它从数据中心的资源解耦架构出发，根据资源利用率和应用行为动态地分配 CPU 和内存资源，通过资源解耦提升了资源使用效率和函数性能.

1.5 小 结

在高密度混部场景下，通过资源解耦为函数提供灵活度更高的资源配置策略，不仅能保障函数的运行时性能，还能降低内存资源的浪费. 为实现该目标，本文提出资源管理系统 Semi-Share，该系统能：1)支持低时间开销的解耦式资源分配；2)支持干扰感知的混部策略；3)保障函数的运行时性能.

设计这样一个解耦式资源管理系统需解决 2 点挑战：1)高密度部署和细粒度的资源分配单位进一步扩大了资源配置空间，基于采样的资源搜索法和利用强化学习构建应用性能模型等方法，因其时间开销大而无法应用于该场景；2)高密度共享下的资源分配不匹配现象，使得应用不得不共享同一资源，在进行资源分配时不仅要考虑如何分配资源，还需考虑应用间的干扰问题.

2 Semi-Share 动态资源分配方法

Semi-Share 是一个针对无服务计算的解耦式资源管理系统. 它面向延迟关键型(latency critical, LC)函数与尽最大努力型(best effort, BE)函数高密度混部的场景，在资源分配方面将 CPU 资源的分配与内存资源进行解耦，自动为函数应用配置合适的 CPU 资源和 LLC 资源. Semi-Share 的关键思想在于分而治之，为解决资源解耦后的资源配置空间爆炸问题，它基于函数应用的特征，将资源分配问题分解为多个子问题，通过分别求解子问题的最优解来确保获得全局的近似最优解，并降低搜索资源配置的时间开销.

为划分资源配置空间，Semi-Share 构建了一个 2 级资源分配架构：第 1 层是函数分组，基于函数的资源使用特征和历史负载信息进行函数分组，根据分组将资源配置空间划分为多个子空间；第 2 层是资源分配，利用贝叶斯优化和加权打分函数构建资源分配模型，并利用打分函数指导模型在资源配置空间中朝正确的方向搜索，降低搜索时间开销. 通过 2 级资源分配架构，Semi-Share 不仅将资源配置问题分解为多个子问题，还能根据优化目标不同分别优化每个层级，例如，降低混部应用间干扰和搜索最优资源配置.

为解决资源分配不匹配问题，Semi-Share 构建了一个基于函数资源使用特征的分组模型. 基于函数应用的分组结果以及资源的分配粒度，区别资源的分配对象，分别为函数应用分配 CPU 时间片资源，为函数分组分配 LLC 资源. 通过区别资源分配对象，不仅解决资源数量与函数数量不匹配的问题，同时能缓解由于共享资源的竞争而造成的性能下降.

Semi-Share 包含 3 个模块：函数特征分析模块、基于函数资源使用偏好的函数分组模块 RPCluster(resource preference based cluster)和资源划分(resource partition, ResPart)模块. 图 3 展示了 Semi-Share 的架构，特征分析模块负责分析函数的资源使用偏好特征，形成画像信息，该过程在用户上传函数时进行 1 次，只有当函数更新时才会更新画像信息. 函数分组模

块 RPCluster 根据函数的画像信息进行函数分组，分组个数为节点上可运行函数应用的 CPU 核数量. 资源划分模块 ResPart 根据函数的运行时数据为每个函数配置合适数量的资源. 当 LC 函数对 CPU 资源需求度较低时，ResPart 可回收部分 CPU 时间片资源给 BE 函数，以充分利用空闲资源，提升资源利用率.

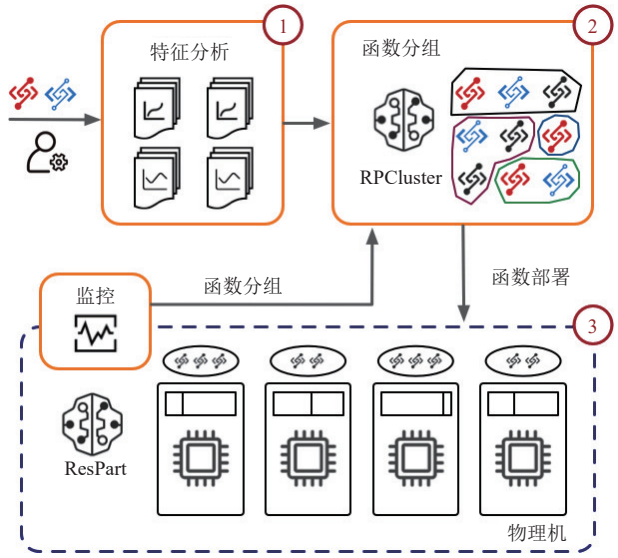


Fig. 3 Framework diagram of Semi-Share

图 3 Semi-Share 架构图

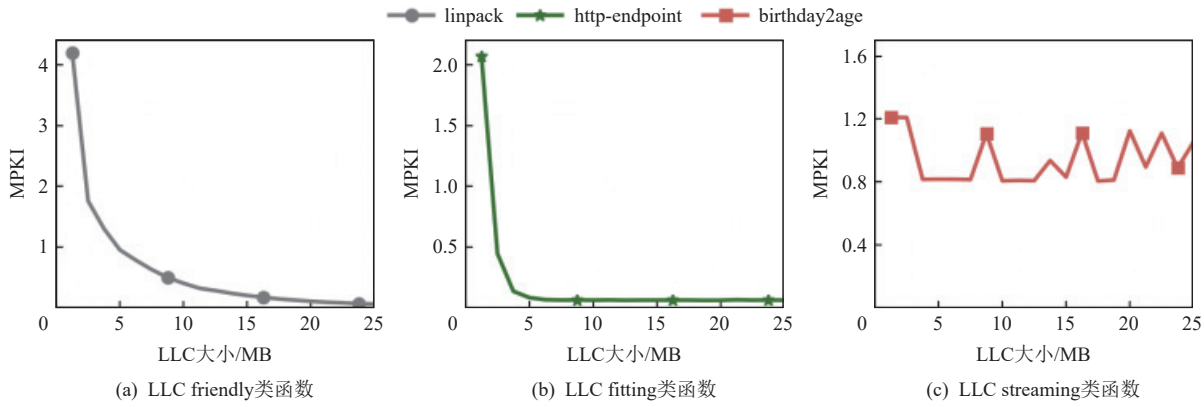
函数分组不仅将 CPU 和 LLC 资源配置空间分解为多个子空间，还将高密度共享下应用性能保障问题分解为 2 个子问题：1）混部应用组合问题；2）多维度资源划分问题. 前者解决高密度共享下哪些函数应用共享同一资源的问题；后者解决资源分配数量不匹配现象下的资源划分问题.

2.1 函数特征分析

Semi-Share 利用函数应用的 LLC 资源使用特征和历史负载信息指导函数分组. 本文采取离线分析法对函数应用进行特征分析，并且只需 1 次离线分析，直到用户更新函数代码时才需更新特征信息. 此外，离线分析法可避免函数应用在自动扩展时由于重新分析所带来的额外开销.

2.1.1 LLC 资源使用特征

遵循先前研究的分类方法[27-28]，本文将函数应用分为 3 类. 如图 4 所示，LLC friendly 类函数的缓存未命中次数 MPKI 会随 LLC 资源的增多而减少；LLC fitting 类函数的缓存未命中次数会随 LLC 资源的增加而突然降低，当为其分配的资源数满足其需求后，再多的 LLC 资源也不会使其收益；LLC streaming 类函数无论分配多少 LLC 资源，其缓存未命中次数均无明显变化，该类函数的时间局部性很差，当数据被缓存进 LLC 后，已缓存的数据在相对较长的时间内将不会被再次访问，因此对 LLC 资源的重利用率很低，并且密集访问访存的行为特征通常会引发 LLC 的频繁替换[29].



(a) LLC friendly类函数 (b) LLC fitting类函数 (c) LLC streaming类函数

Fig. 4 Function's resource preference to LLC resource

图 4 函数应用对 LLC 资源的资源使用偏好

Semi-Share 根据函数应用对 LLC 资源的使用特征进行资源分配. 若让混部函数无序共享 LLC 资源，streaming 类函数会因频繁访问访存而导致另外 2 类函数的 LLC 未命中次数增高，进而影响混部函数的请求处理延迟. 实验发现，若为 streaming 类函数分配隔离的 LLC 资源，能为其他类型函数带来 37.1%~75.1% 的尾延迟性能提升，同时 streaming 类函数的性能也没有受到较大影响. 因此，虽然函数应用对 LLC 资源的无序共享会导致性能的不可预测性，但利用其资源特征进行资源分配，不仅能缓解由于资源竞争导致的性能下降，而且能解决 LLC 资源数量与函数应用数量不匹配的问题.

2.1.2 负载调用特征

本文发现函数应用不仅遵循昼夜调用模式，而

且它与自身历史调用之间也存在较高的相似性. 以函数默认的生命周期为基础, 我们分析了微软函数集群的数据集, 发现约 54% 的函数的当前调用与历史调用的平均调用相似度大于 0.9, 77% 函数的平均调用相似度大于 0.5, 如图 5(a)所示. 本文使用皮尔森相似度来度量函数每个生命周期之间的调用相似度, 数值越大代表调用相似度越高, 函数负载的可预测性就越高. 并且函数调用的可预测性与函数的触发器类型有较大相关性. 如图 5(b)所示, 触发器类型为 Timer 的函数其调用模式与历史调用的相似度最高, 平均值达到了 0.91, 触发器类型为 Http 的函数其平均调用相似度达到了 0.68. 函数负载调用的可预测性特征可用于指导函数的分组, 并指导资源分配模型中初始化采样点的选择, 以加速资源配置的搜索过程. 本文利用一个滑动窗口来使用函数的历史调用数据, 使用窗口内 1 min 级别的调用平均值来量化函数的历史负载量.

2.2 函数分组

本文基于函数的资源使用与负载调用特征设计分组算法 RPCluster. 分组算法分为 2 个阶段: 第 1 阶段根据函数对 LLC 资源的使用特征将函数分为 2 组, streaming 类函数为一组, fitting 和 friendly 类函数为另一组, ResPart 以第 1 阶段的分组结果进行 LLC 资源的分配. 第 2 阶段根据函数的历史负载信息进一步分组, 属于同一组的函数应用运行在同一 CPU 核上, 最终使各个 CPU 核之间的负载均衡分布. 函数分组不仅利用函数的特征信息解决了哪些函数共享同一资源的问题, 为 Semi-Share 构建了干扰感知的能力, 而且以 CPU 核为边界将 CPU 资源和 LLC 资源的配置空间划分为多个子空间.

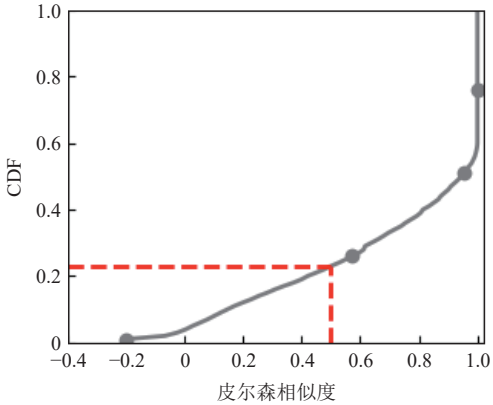
分组算法介绍如下. 给定 N 个函数, RPCluster 的目标是将 N 个函数分为至多 M 个组. 其中, M 是可运行函数的 CPU 核数量. 当 $N \leqslant M$ 时, 无需分组, N 个函数分别运行在单独的 CPU 核上, 函数应用间不共享核内资源. 当 $N>M$ 时, 对函数进行分组. 首先, 依据函数的 LLC 资源使用特征分为 2 组: LLC streaming 和 others, 根据函数的历史负载信息计算这 2 组函数分别所需的 CPU 核数. 其次, 将 2 组函数按历史负载量从高到低排序, 并依次放置在 M 个 CPU 核上, 直至 M 个核上均有 1 个函数应用. 之后, 利用负载均衡算法对剩余函数进行分组, 根据函数和 CPU 核的负载情况将待分组函数放入各个组内, 以达到各 CPU 核之间负载均衡的目的. 函数分组的伪代码如算法 1 所示.

算法 1. 函数应用分组算法.

输入: N 个函数的 LLC 资源使用特征 *res_type*, N 个函数的历史负载信息 *func_load*, 可运行函数的 CPU 核数量 M;

输出: 函数的分组个数, 每个分组内的负载信息, 基于资源使用特征的分组结果.

① 基于 *res_type*, 将函数分为 2 组, 分别为 *cluster_streaming* 与 *cluster_others*;
② if $N \leqslant M$ then
③ return N, *func_load*, *cluster_streaming*, *cluster_others*;
④ end if
⑤ 基于 *func_load*, 分别计算 2 个分组内函数的负载量总和为 *load_streaming*, *load_others*;
⑥ 基于 M, 计算每个 CPU 核上的平均负载量为 *load_avg*;
⑦ 计算 2 个分组所需的 CPU 核数分别为 *cpus_streaming=load_streaming/load_avg*, *cpus_others=load_others/load_avg*;



(a) 调用相似度分布

触发器	函数比例/%	平均调用相似度
Http	35.31	0.68
Timer	34.77	0.91
Queue	15.41	0.59
Orchestration	6.95	0.69
Event	3.26	0.44
Storage	2.11	0.61
其他	2.19	0.88

(b) 触发器类型

Fig. 5 Similarity distribution of the function's invocation pattern

图 5 函数调用模式相似度分布

⑧ 初始化所有分组 *clusters* 的负载信息为 0;
⑨ 根据 *func_load* 将函数按负载量从大至小排序，并从 2 个分组中共取出前 *M* 个函数放入每个分组中，更新 *clusters* 中的负载信息;
⑩ while 存在未分组的函数 *f* do
⑪ if *f* ∈ *cluster_streaming* then
⑫ 在 *cpus_streaming* 中选择负载最低的分组 *c*，并将 *f* 放入组 *c* 中;
⑬ 更新 *clusters* 中的负载信息;
⑭ end if
⑮ if *f* ∈ *cluster_others* then
⑯ 在 *cpus_others* 中选择负载最低的分组 *c*，并将 *f* 放入组 *c* 中;
⑰ 更新 *clusters* 中的负载信息;
⑱ end if
⑲ end while
⑳ return*M*,*clusters*,*cluster_streaming*,*cluster_others*.

2.3 资源划分

当函数分组完成后会启动资源划分模块 ResPart，为每个函数分配资源. 为解决高密度混部下资源分配不匹配问题，ResPart 将 CPU 资源与 LLC 资源的分配分开. 对于 CPU 资源，ResPart 构建了一个基于贝叶斯优化的资源分配模型和加权打分函数，对每个函数分组中的函数，在 CPU 资源配置空间中快速搜索最优配置；对于 LLC 资源，根据 RPCluster 第 1 阶段的函数分组结果，ResPart 利用二分搜索法为 2 组函数分配 LLC 资源.

2.3.1 贝叶斯优化模型

ResPart 采用贝叶斯优化为每个函数分组构建一个 CPU 资源分配模型. 模型的输入是分组内所有函数的 CPU 资源配置，输出是对组内函数在该资源配置下是否达到优化目标的评估分数. ResPart 采用贝叶斯优化的优势有 3 点：1）贝叶斯优化无需知道函数应用的资源配置和目标函数间的关系，它通过建立一个未知目标函数的随机模型，在搜索过程中迭代更新，以此决定在 CPU 资源配置空间中的搜索方向，并不断接近该子问题的最优点；2）能通过有限次数的采样评估快速找到最优解，而无需大量的训练数据和训练开销；3）贝叶斯优化能同时分配多个维度的资源，并且适用资源配置与应用性能为非凸函数的场景，可扩展性强.

与传统贝叶斯优化相比，在高密度混部场景下构建基于贝叶斯优化的资源分配模型，需解决 2 个问题：1）需构建一个平滑的目标函数，不仅能衡量不同类型应用在不同混部组合下的性能变化，还要满足目标函数的平滑性约束，以保证模型能在资源配置空间中朝正确的方向搜索；2）由于初始化采样点决定了模型的收敛速度和搜索质量，需根据函数特征选择合适的初始化采样点，降低搜索最优配置的时间开销.

为解决上述 2 个问题，本文采取 4 个措施：1）本文针对无服务函数的混部场景，构建了一个分段加权打分函数，打分函数的分值表示了函数在某资源配置下是否满足了性能优化目标. 2）不同于传统贝叶斯方法为每个应用均构建一个资源分配模型，Semi-Share 根据函数分组的个数构建 *M* 个模型，每个模型同时为组内所有函数分配 CPU 资源. 该设计的优点在于能从资源分配子空间的角度寻找子问题的最优解，并能降低为每个函数构建模型的时间与资源开销. 3）根据 CPU 资源配置空间的大小以及函数的特征，有针对性地设置资源分配模型的探索与开发比例，避免模型陷入局部最优解. 4）利用函数的负载特征构建初始化采样点，加速模型的搜索过程.

ResPart 选择带有 Matérn 内核的高斯过程作为代理模型. 研究显示，高斯过程对资源分配问题的适配度较高[25]，并且 Matérn 协方差不限制目标函数的强平滑性[30-31]，适用于性能目标多样的多类型应用混部场景. ResPart 选择 EI（expected improvement）作为采集函数，它能平衡搜索过程中开发与利用的关系[30]，避免搜索陷入局部最优解中.

本文构建了 2 种初始 CPU 资源配置：1）平均分配，根据组内函数应用的个数平均分配 CPU 时间片资源；2）基于历史负载信息的资源划分，根据函数应用间的历史负载信息按比例分配 CPU 时间片资源. 这 2 种初始化资源配置：可用于加速资源分配模型在 CPU 资源配置空间中的搜索过程.

2.3.2 目标函数

ResPart 的优化目标是，在保障 LC 函数应用的性能满足服务质量（quality of service, QoS）目标的前提下，提升 BE 函数应用的执行性能. LC 函数应用的性能是指函数处理每个请求的执行时间，即请求延迟[8-11,21].其 QoS 目标是指函数应用在最大负载下的 99 分位请求延迟[10]，该 QoS 目标通过函数应用的离线分析获得. 本文将 0.8 倍的 QoS 目标设置为 LC 函数的优化目标，若 LC 函数的请求延迟大于该阈值，表明函数将要发生或已发生 QoS 违例. LC 函数的优化目标如式（1）所示：

$$\forall f_{\mathrm{LC}} \in LCFs, Latency_{\mathrm{current}} \leqslant 0.8 \times QoS_{\mathrm{target}}, \quad (1)$$

其中 *LCFs* 是指 LC 函数集合，$Latency_{\mathrm{current}}$ 是指函数

f_{LC} 在当前的资源配置下请求的处理延迟, QoS_{target} 是指函数 f_{LC} 的 QoS 目标, 本文将优化目标设置为 0.8 倍的 QoS_{target}, 以便在函数即将发生 QoS 违例时提前进行资源调整.

BE 函数应用的执行性能是指函数在运行过程中的 IPC(instruction per cycles)值[32], 其最优性能是指在单独运行时的 IPC 值. BE 函数的优化目标为, 在所有 LC 函数均达到优化目标的前提下, 最大化 BE 函数的执行性能, 如式(2)所示:

$$\text{if } \forall f_{LC} \in LCFs,\ Latency_{current} \leqslant 0.8 \times QoS_{target} \text{ then}$$
$$\text{if } \forall f_{BE} \in BEFs, \max\left(\frac{Colocation_{IPC}}{Solo_{IPC}}\right), \tag{2}$$

其中 $BEFs$ 是指 BE 函数集合, $Colocation_{IPC}$ 是指函数 f_{BE} 在当前的资源配置下混部运行时的执行性能, $Solo_{IPC}$ 是指函数 f_{BE} 单独运行时的执行性能.

目标函数用于评估函数应用在当前资源配置下是否达到函数的优化目标, 以判断该资源配置与未知目标函数最优值的接近程度. 在高密度混部场景下, 目标函数的设计需满足 4 点要求: 1)能涵盖不同类型的混部场景; 2)能衡量 2 种函数应用的性能是否满足约束条件; 3)能同时对分组内的函数分配 CPU 资源; 4)目标函数足够平滑, 保证搜索的有效性.

本文将无服务计算的混部场景归纳为 2 类: 第 1 类为 LC 函数间混部, 该场景的优化目标是保障每个 LC 函数的性能; 第 2 类为 LC 函数与 BE 函数混部, 该场景的优化目标是在保障 LC 函数服务质量的前提下, 最大化 BE 函数的执行性能. 为同时涵盖这 2 类混部场景, 本文构建了一个分段加权打分函数作为目标函数. 目标函数的分值在 [0, 1] 之间, 分值越接近 1 代表函数性能越好, 0 代表在该资源配置下所有 LC 函数均发生了 QoS 违例. 目标函数如式(3)(4)所示:

$$Score_1 = \frac{1}{N_{LC}} \sum_{n=1}^{N_{LC}} \min\left(1.0, \frac{0.8 \times QoS_{target}}{Latency_{current}}\right), \tag{3}$$

$$Score_2 = \begin{cases} \dfrac{1}{2}\dfrac{1}{N_{LC}} \displaystyle\sum_{n=1}^{N_{LC}} \min\left(1.0, \dfrac{0.8 \times QoS_{target}}{Latency_{current}}\right), \\ \quad \text{发生} QoS \text{违例}, \\ \dfrac{1}{2} + \dfrac{1}{2}\dfrac{1}{N_{BE}} \displaystyle\sum_{n=1}^{N_{BE}} \dfrac{Colocation_{IPC}}{Solo_{IPC}}, \text{其他}, \end{cases} \tag{4}$$

其中 N_{LC} 代表当前分组内 LC 函数的个数, N_{BE} 代表当前分组内 BE 函数的个数, QoS_{target} 表示 LC 函数的服务质量目标, $Latency_{current}$ 代表 LC 函数在当前资源配置下的请求处理延迟, $Colocation_{IPC}$ 代表 BE 函数在当前资源配置下的 IPC 值, $Solo_{IPC}$ 代表 BE 函数在独自运行时的 IPC 值. $Score_1$ 为在当前资源配置下第 1 类混部场景的分值, $Score_2$ 为在当前资源配置下第 2 类混部场景的分值.

在第 1 类混部场景下, 当前分组的分值仅由 LC 函数的性能计算得到, 若分组内所有 LC 函数的延迟均低于 0.8 倍的 QoS_{target}, 即函数性能在安全范围内且达到了优化目标, 当前函数的分值为 1. 该函数分组的分值是所有函数分数的平均值.

在第 2 类混部场景下, 目标函数的分值根据 LC 函数是否满足优化目标进行分段计算. 当存在 LC 函数已发生或将要发生 QoS 违例时, 只计算 LC 函数的分数, 该设计的目标是优先保障 LC 函数的性能. 当该分组中所有 LC 函数的性能均达到了优化目标时, 计算 BE 函数的性能分数. 该函数分组的分值是所有函数分数的平均值, LC 函数和 BE 函数的分值权重均为 0.5.

为分组内所有函数同时分配资源时, 资源分配模型需满足 2 点约束: 1)组内所有函数分配的 CPU 资源总量应不大于 CPU 核的时间片总量, 其形式化表述如式(5)所示; 2)组内每个函数分配的 CPU 资源应不低于操作系统所规定的最低资源量, 其形式化表述如式(6)所示.

$$\sum_{f \in Funcs} Res_f \leqslant Res_{CPU}, \tag{5}$$

$$\forall f, Res_f \geqslant \alpha, \tag{6}$$

其中 $Funcs$ 代表分组内的函数集合, Res_f 代表函数 f 的 CPU 资源配置, Res_{CPU} 代表该分组的 CPU 资源上限, α 代表 CPU 资源分配的最小值.

根据优化目标, 本文为资源分配模型设置了 3 条终止条件: 1)EI 低于阈值, 本文设置为 1%; 2)目标函数大于阈值, 针对纯 LC 应用和 LC/BE 应用混部的场景, 阈值分别设置为 0.9 和 0.99, 前者是为了保证所有 LC 函数均达到了优化目标, 后者是为了保证在所有 LC 函数均达到优化目标的基础上最大化 BE 函数的性能; 3)至少采样了 K 个资源配置点, 本文设置 K=2, 不包括初始资源配置. 当同时满足这 3 个条件时, ResPart 终止该分组的资源搜索过程. 这些参数可根据实际情况进行调整, 以确保 ResPart 不会过早地终止搜索过程, 并搜索到最优资源配置.

2.3.3 LLC 资源划分与持续性能保障

待每个分组的模型执行完毕时, ResPart 开始进行 LLC 资源划分. ResPart 将 LLC 资源划分为 2 部分,

一部分给 LLC streaming 类函数，一部分给其他类型函数. 为减少资源划分的时间开销，本文设计了一个二分资源划分算法，根据 2 个大组中函数运行时性能快速划分 LLC 资源.

当 ResPart 完成每个分组的 CPU 资源分配并完成 LLC 资源分配后，随即启动资源动态反馈调节程序，以持续保障函数应用的运行时性能. 该机制为函数运行提供了 2 点保障：1)避免由于 ResPart 找到次优解而导致函数性能下降；2)适应函数调用的波动性，在这种波动性下持续为函数提供性能保障.

3 实验评估

3.1 实验配置

为评估 Semi-Share，本文从无服务函数标准测试集 FunctionBench[33]，FaaSProfiler[34]，AWS 函数仓库中选取了若干函数应用. 每个 LC 函数的最大负载和 QoS 目标是根据函数在单独运行时能处理的最大请求量与对应的 99 分位延迟确定的. 其中，BE 函数来自大数据和 AI 负载基准测试集[35]，我们将其进行了改动以适应函数平台. 本文使用 wrk2 负载生成器为函数生成持续的负载.

我们在拥有 20 核的 Intel Xeon E5-2630 CPU(2 个 socket)上进行实验，物理机的内存层次结构包括 25 MB 的共享 LLC 缓存和 64 GB 的内存. 该处理器支持英特尔的 RDT(resource director technology)技术，我们利用其缓存分配技术来为函数分配 LLC. 实验平台的配置如表 2 所示，所有的实验都在 Ubuntu 16.04 中进行，使用的 Linux 内核版本为 4.15.0、Docker 版本为 20.10.2.

Table 2 Experimental Platform Configuration

表 2 实验平台配置

配置	参数
处理器	Intel® Xeon® CPU E5-2630 v4
操作系统	Ubuntu 16.04.1(kernel 4.15.0)
内存	32 GB×2, 2667 MHz
Socket 数量	2
每个 Socket 的核数	10
每个 Socket 的 LLC 数	25 600 KB(25 MB, 20 ways)

3.2 对比方法与对比指标

在 Semi-Share 的函数分组下，资源分配是以组为单位进行的，每个分组(CPU 核)上均运行一个资源分配模型，为组内的函数分配 CPU 资源. 待每个分组完成 CPU 资源分配后，再为函数分组分配 LLC 资源. 本文使用下述资源分配策略与 Semi-Share 进行对比.

1)耦合式资源分配. 该方法是当前无服务计算平台所采取的资源分配方法，它根据函数应用设置的内存大小按固定比例为函数分配 CPU 资源. 本文使用 AWS Lambda 的资源分配比例进行实验.

2)COSE[24]. COSE 是一个利用贝叶斯优化为函数搜索最优内存配置和云服务提供商的资源分配系统. 它没有将 CPU 和内存资源的分配进行解耦，而是使用云服务提供商的资源分配策略为函数分配 CPU 资源. 针对无服务计算中的函数应用性能保障和用户成本问题，它分别构建了函数的执行模型和成本模型，本文利用执行模型与 Semi-Share 进行对比，资源分配比例采用 AWS Lambda 的比例.

3)随机搜索法(Random). Random 使用均匀分布随机地从资源配置空间中选择一个配置进行采样分析，并且不重复采样.

4)梯度下降法(gradient descent method, GDM). GDM 为发生 QoS 违例的函数一次增加一个单位的资源，当系统中没有空闲资源时，通过从拥有最高性能松弛度的函数回收资源来进行重分配. 本文以 PARTIES[10] 的主逻辑为模板重新实现了 GDM，它的初始配置是资源平均分配，采样的间隔设置为 1 次/s，资源调整的步伐和 Semi-Share 一致.

5)最优配置(Oracle). 本文通过穷举所有资源配置来找到最优的资源分配方案，并使用它来比较上述 4 种资源搜索方法与最优配置间的差距.

本文使用下述指标对比相关工作.

1)函数性能. 对于 LC 函数应用，性能是指函数请求的处理延迟. 在实验中，使用 2 个指标评估 LC 函数的性能：①使用函数请求的 99 分位延迟是否满足 QoS 目标，来判断函数是否满足性能约束，由于该指标能直接体现函数的性能变化，因此被广泛应用于性能评估[1,8-11,36]；②使用函数在最优资源配置(Oracle)下的平均延迟与在某资源配置下的平均延迟的比值[10-11,37]，来判断该资源配置与最优资源配置间的差距，由于函数以执行时间为主要计费依据，因此这里使用平均延迟而未使用 99 分位延迟. 对于大数据处理类的 BE 函数，性能是指函数在 CPU 上的执行效率，可用函数的 IPC 值或单位时间内处理的请求个数表示. 在实验中，使用 BE 函数在某资源配置下单位时间内处理的请求个数与在最优资源配置下处理的请求个数[10,22,32,37]，来判断该资源配置与最优资源配置间的差距.

2)采样次数. 实验采用搜索资源配置的采样次数来衡量时间开销. Semi-Share 与上述资源分配策略的采样时间间隔均为 1 s.

3.3 资源配置的灵活度

相比于耦合式资源分配策略, Semi-Share 的解耦式资源分配策略具有更高的灵活性. 图 6 展示了当 2 个 LC 函数在同一 CPU 核内混部时, 3 种资源分配策略在不同负载强度下的性能对比. cnn-serving 函数和 markdown2html 函数运行在同一个 CPU 核上, 分别为这 2 个函数注入不同强度的负载. Semi-Share 分别为 cnn-serving 函数和 markdown2html 函数配置了固定数量的内存资源: 384 MB 和 128 MB, 并根据函数运行时的负载情况为其分配 CPU 资源; 在 AWS Lambda 的耦合式资源分配比例下, 分别为 cnn-serving 函数和 markdown2html 函数配置了同等数量的内存资源: ⟨0.217 CPU, 384 MB⟩和⟨0.072 4 CPU, 128 MB⟩; COSE 根据 2 个函数的执行模型, 为其配置合适的内存资源和相应比例的 CPU 资源. 图 6 中灰色方块代表 2 个 LC 函数在相应负载强度下混部时, 均能满足性能约束; ×代表至少 1 个 LC 函数无法满足性能约束; NC(no colocation)代表 CPU 核上仅运行了 1 个函数, 没有混部.

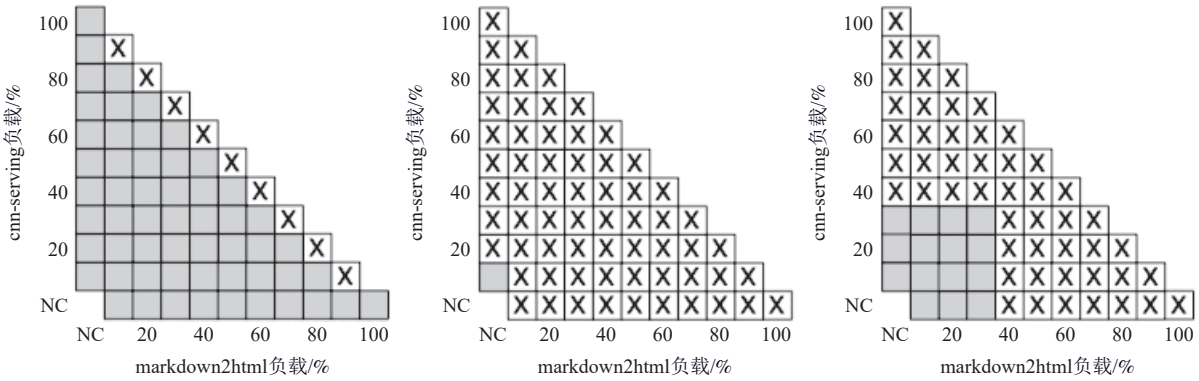


Fig. 6 Function performance under different loads and resource allocation strategies

图 6 函数在不同负载和资源分配策略下的函数性能

在耦合式资源分配策略下, 有限的资源只能支持函数在较小的负载场景下正常运行, 而解耦式资源分配策略的灵活度更高, 能支持更大范围的负载场景. 尽管 CPU 核上只运行了 2 个函数并剩余大量 CPU 资源, 但固定的资源分配比例限制了函数能使用的 CPU 资源上限, 当函数负载增高时, 只能为函数分配更多的内存或扩展更多的函数实例, 这不可避免地造成了内存资源浪费并引入了额外的冷启动开销. COSE 虽然利用贝叶斯优化为函数分配最优的内存配置, 但其执行模型在使用函数的请求处理延迟作性能指标时, 不仅没有进行归一化处理, 而且没有形式化表示函数性能应满足怎样的约束条件, 仅以 EI 作为模型运行的结束条件, 因此在某些负载场景下会出现资源配置不足的情况. 而 Semi-Share 能利用加权打分函数更精确地表示混部函数当前是否达到优化目标或与之的差距, 从而指导模型在资源配置空间中朝正确的方向搜索, 自动为函数分配合适的 CPU 资源. 因此, 在这 3 种资源分配策略下, Semi-Share 能支持更大范围的负载场景, 具有更高的灵活性.

3.4 资源分配质量

Semi-Share 能够高效地支持多个 LC 函数在同一 CPU 核上混部, 并保障混部函数的性能. 图 7 展示了 2 组混部组合在变化的负载强度下函数的平均性能对比. 平均性能是指, 多个混部函数在最优资源配置下的平均延迟与在某资源配置下的平均延迟之比的平均数. 在 3 个 LC 函数混部的实验中, 其中一个函数的负载强度从 10% 增加到 70%, 另外 2 个函数的负载均保持在 10%. 由实验结果可知, GDM 在函数负载强度较低时还能提供较优的性能, 当函数的负载强度不断增强时, 函数的整体性能不断下降, 这是由于当负载增大时能分配给每个函数的冗余资源随之减小, 并且 GDM 在为其中 2 个函数分配刚刚好的资源后就停止搜索, 加之调用的波动性, 函数在运行时会产生性能波动. Random 策略在某些场景下能获得比 GDM 高的性能分数, 这是由于平均性能比值是取 3 个混部函数性能比值的平均值, 当把更多的资源都分配给其中 2 个函数时会出现该情况, 但因其资源分配的随机性, 在不同的混部组合下其平均性能差距较大. COSE 虽然也利用贝叶斯优化模型来自动寻

找最优资源配置，但它在不同负载下性能波动较大，这是由于 COSE 没有为函数的执行模型设定具体的优化目标，均是以 EI 作为模型的终止条件，而没有考虑资源配置是否满足函数的性能约束. Semi-Share 在不同的混部组合下都能获得最接近 Oracle 方案的性能. 这得益于其资源分配模型和目标函数，Semi-Share 的资源分配模型同时为分组内的函数搜索资源配置，这使得函数能充分利用 CPU 核上的空闲资源. 其目标函数的优化目标是保障每个 LC 函数的性能，因此资源分配模型仅在达到该优化目标时才停止搜索过程. 此外，在资源分配完成后，Semi-Share 还会利用动态反馈调节程序持续保障函数性能. 因此，Semi-Share 在函数运行的整个生命周期中能保证较高的性能，平均能达到 Oracle 90% 左右的性能.

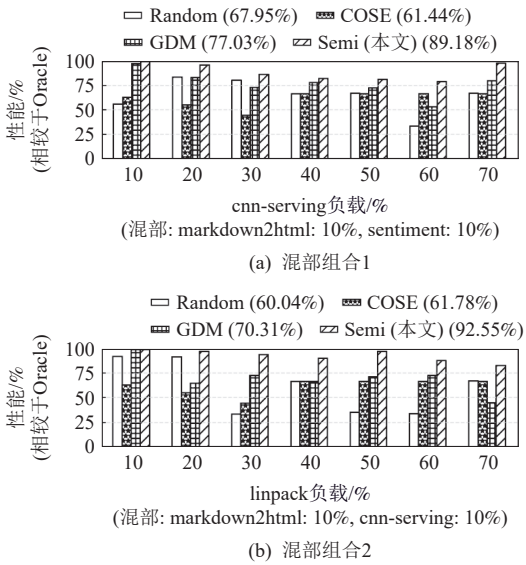


Fig. 7 Performance comparison of three LC functions colocation scenario

图 7 3 个 LC 函数混部的性能对比

当 LC 函数和 BE 函数在同一 CPU 核上混部时，Semi-Share 不仅能保障 LC 函数的性能，并且能将空闲的资源分配给 BE 函数，以最大化资源使用效率. 图 8 展示了 2 个 LC 函数和 1 个 BE 函数混部时 3 个函数的平均性能对比. 其中，BE 函数的性能是指在混部时单位时间内处理完成的请求个数与在单独运行时单位时间内处理完成的请求个数的比值. GDM 和 Random 在不同的混部组合下的性能差异较大，这是由于当 LC 函数和 BE 函数混部时，GDM 的实现逻辑会倾向于将空闲资源都分配给 BE 函数，以提升资源利用率，只给 LC 函数分配刚刚好的资源. 与前述实验相同，由于 Random 的随机特性，在多种混部场景下都无法同时保障 3 个函数的性能. COSE 的执行模型为 2 个负载较低的 LC 函数都分配了足量的资源，保证了 2 个 LC 函数的性能，但因其执行模型对函数是否满足性能约束无感知，为 LC 函数分配过多资源导致 BE 函数分配的资源较少，导致平均性能逊色于 Semi-Share. Semi-Share 的分段加权打分函数保证了在 2 类混部场景下，都能为每个函数搜索到接近最优的资源配置，获得较高的平均性能.

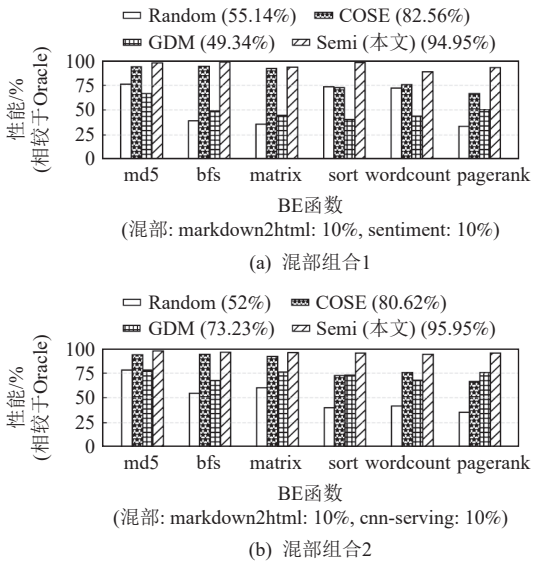
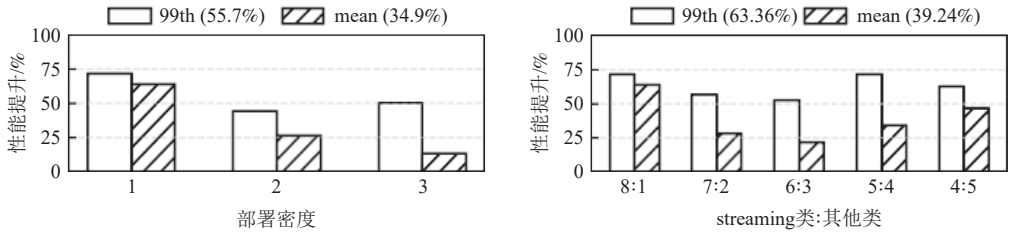


Fig. 8 Performance comparison of two LC functions and one BE function colocation scenario

图 8 2 个 LC 函数与 1 个 BE 函数混部性能对比

Semi-Share 通过将 LLC 资源分为 2 部分，不仅降低了函数应用的 99 分位延迟，相比于不划分 LLC 资源而且能带来平均 34.9% 的性能提升. 但 LLC 资源划分带来的性能提升会受部署密度和应用类型比例的影响. 图 9(a)展示了当 9~27 个函数部署在 9 个 CPU 核上时，划分 LLC 资源所带来的性能提升. 随着部署密度的提升，部署密度为 3、函数个数为 27 的混部组合与部署密度为 1、函数个数为 9 的混部组合相比，平均性能提升相差了 4.6 倍，这是由于当 CPU 核上部署的函数数量增加时，函数之间对片上资源的竞争加剧所致[11]. 此外，随着 streaming 类函数的占比减少，LLC 资源划分所带来的性能提升有所下降，这是由于 LLC friendly 和 fitting 类函数间也会竞争有限的 LLC 资源，但依旧能带来平均约 39% 的性能提升，如图 9(b)所示. 实验发现，若为 LLC friendly 和 fitting 类函数单独划分 LLC 资源，不但不能带来更进一步的性能提升，反而会降低 LLC friendly 类函数的性能，这是由于函数对 LLC 资源的需求粒度与当前 LLC 资源的分配粒度不匹配所致.



(a) 部署密度 (b) streaming类型占比

Fig. 9 Performance improvement caused by the LLC partition

图 9 LLC 资源划分所带来的性能提升

3.5 资源搜索速度

Semi-Share 不仅时间开销低，而且函数的部署密度和负载强度对时间开销的影响也很小. 图 10、图 11 展示了 Semi-Share 和 GDM 在不同混部组合下找到最优资源分配方案的采样次数. GDM 的采样次数受部署密度、混部组合和负载强度等因素的影响较大，尤其是当 LC 函数和 BE 函数混部时，其采样次数明显高于多个 LC 函数混部的场景，并且平均采样次数是 Semi-Share 的 7 倍左右. 采样次数代表了搜索资源配置的时间开销. GDM 的时间开销远大于 Semi-Share 的原因如下：1)函数应用的资源分配粒度小以及 GDM 每步调整的资源数量小，导致小步搜索的收敛速度较慢；2)GDM 以资源平均分配为搜索的起始点，当 LC 函数的负载强度增高时，初始资源量的不足会导致请求排队积压，在资源搜索过程中，随着 LC 函数被分配的资源量越来越多，请求的排队积压情况不断缓解，但该过程需要较多的采样次数；3)在 LC 函数与 BE 函数混部时，GDM 的实现逻辑倾向于将空闲资源都分配给 BE 函数，而只给 LC 函数分配刚刚好的资源数量，因此在资源划分的边界搜索时，需较多的采样次数来判断最优分配点.

Semi-Share 的采样次数受混部密度、负载强度和混部组合的影响都很小，平均仅需 6.41 次采样即可找到最优的资源配置. 这得益于 3 点设计：1)Semi-Share 通过平衡资源配置空间中采样点的开发和利用来保证快速在整个资源配置空间中进行搜索，根据经验值，本文设置的探索和开发比例为 1∶4；2)分段加权打分函数保证了资源分配模型在资源配置空间中朝正确的方向进行搜索；3)利用函数的历史负载信息加速搜索过程. 这保证了 Semi-Share 能在较大的资源配置空间中快速搜索最优的资源配置. 图 12 展示了 Semi-Share 和 GDM 的搜索过程，GDM 从资源分配的起始点开始逐步地进行采样分析，由于其步子迈得小、资源空间大导致搜索过程较慢，而 Semi-Share 能直接在整个空间中进行搜索，在 10 次采样以内就

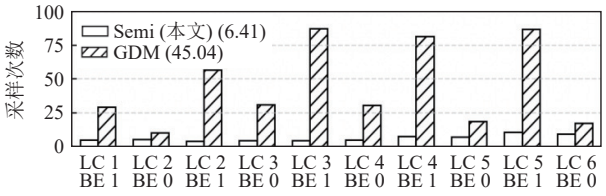
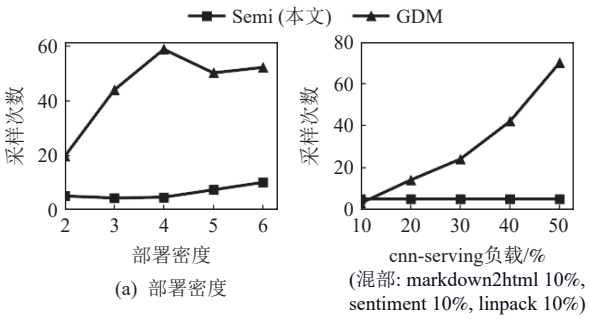


Fig. 10 Sampling numbers under different colocation combination

图 10 不同混部组合下的采样次数



(a) 部署密度 (b) 负载强度

Fig. 11 Sampling numbers vary with different colocation densities and intensity of load

图 11 采样次数在不同混部密度和负载强度下的变化

能找到最优资源配置. Semi-Share 在完成资源分配后会启动一个动态资源调整程序，该程序根据函数的运行时状态为函数小步调整资源配置，为函数提供持续的性能和 QoS 保障.

函数的历史调用信息加速了资源配置的搜索过程，当历史调用信息不准确时，Semi-Share 依旧能快速找到最优资源配置. 函数的调用信息不准确可分为 2 类：第 1 类是实际负载强度低于历史负载强度；第 2 类是实际负载强度高于历史负载强度. 由于函数的历史调用信息应用于初始采样点的选择，第 1 种情况对于资源搜索的过程影响较小，而第 2 种情况会对资源搜索过程造成较大影响. 图 13 展示了在 2 种混部组合下，函数的实际负载强度高于历史调用强度时的搜索过程. 对于 LC 函数和 BE 函数混部的

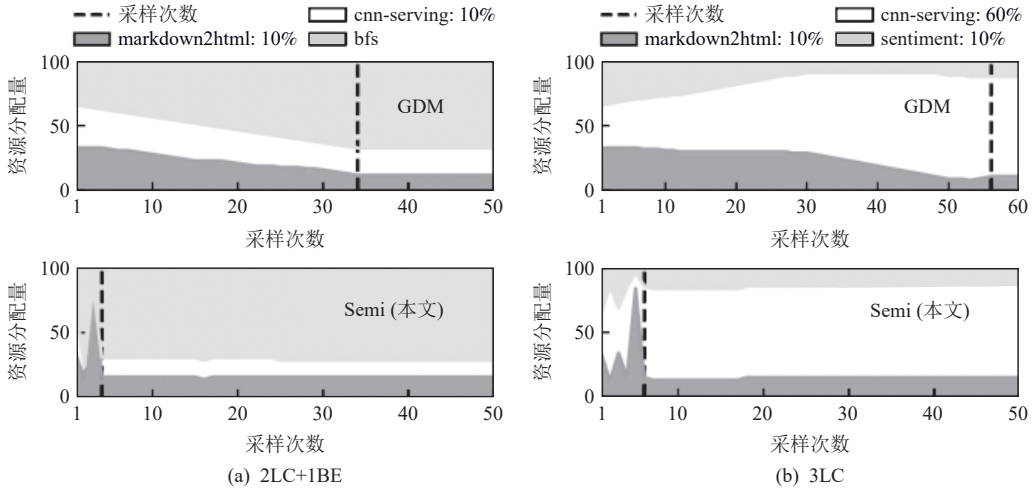


Fig. 12 Resource search process under different colocation combination

图 12 不同混部组合下资源搜索过程

场景如图 13(a)所示，markdown2html 函数和 cnn-serving 函数的真实负载为 50% 和 10%，但根据历史负载预测为 10% 和 15%，在构建初始资源配置时 markdown2html 函数分配的资源较少，cnn-serving 函数分配的资源较多. 打分函数的设计以保障 LC 函数的性能为首要目标，虽然初始配置为 markdown2html 函数分配的资源较少，但目标函数能感知函数的性能变化，并指引模型朝正确的方向进行搜索. 对于 3 个 LC 函数混部的场景如图 13(b)所示，cnn-serving 函数的真实负载为 60%，但根据历史负载预测为 25%，在构建初始资源配置时为其分配的 CPU 资源较少. 由于搜索的过程要同时保障 3 个 LC 函数的性能，实际的采样次数明显变多. 如图 13(b)所示，Semi-Share 在整个资源空间中广泛地搜索资源配置，并在第 28 次采样时停止了搜索过程. 虽然 Semi-Share 在该场景下的采样次数远大于其平均采样次数，但依旧优于 GDM. 因此，当函数历史负载信息不准确时，Semi-Share 也能快速找到最优资源配置方案.

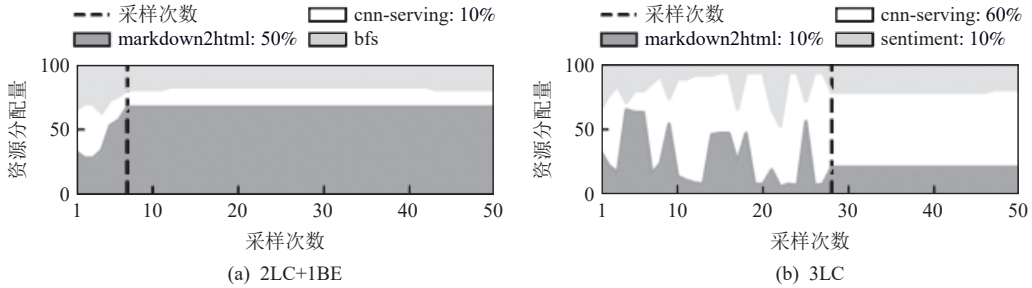


Fig. 13 Resource search process under inaccurate function load prediction

图 13 函数负载预测不准时资源搜索过程

4 总 结

本文对无服务计算的高密度混部场景进行了深入分析，发现当前的耦合式资源分配策略灵活度低，无法适用于更多的函数类型，并且会迫使用户过量分配内存资源，造成内存资源浪费. 本文提出 Semi-Share，一个面向无服务计算的解耦式资源管理系统，在为函数分配最优资源配置的同时降低混部函数间的干扰. 为解决因资源分配单位变小和混部密度增加而导致的资源配置空间爆炸问题，本文通过构建 2 级资源分配架构，将资源配置空间拆分为多个子空间分而治之. 为保证资源分配质量并降低搜索资源配置的时间开销，本文利用贝叶斯优化构建资源分配模型，为函数自动搜索最优资源配置，并利用分段加权打分函数指导模型在资源空间中朝正确方向搜索. 为持续保障函数的运行时性能，在完成资源分配后 Semi-Share 会启动一个动态资源调整程序. 实验表明，Semi-Share 具有较高的灵活性，与广泛使用的梯度下降方法相比，搜索速度快、质量高，与同样使用贝叶斯优化的耦合式策略相比，能为函数带来平均 32.25% 的性能提升.

作者贡献声明:郭静提出本文的主要工作思路,并负责文章的设计、实现与撰写,完成相关实验的设计和实验评估;胡存琛协助完成实验部分的设计;包云岗负责文章整体思路的指导和逻辑的调整,提出指导意见并修改论文.

参 考 文 献

[1] AWS. Configuring Lambda function options[EB/OL]. 2022 [2022-11-30].https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html

[2] Google. Google functions pricing[EB/OL]. 2022 [2022-11-30]. https://cloud.google.com/functions/pricing

[3] IBM. System details and limits[EB/OL]. 2022 [2022-11-30].https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-limits

[4] Azure. Azure functions hosting options[EB/OL]. 2022 [2022-11-30].https://learn.microsoft.com/zh-cn/azure/azure-functions/functions-scale

[5] Tencent. Tencent function general problem[EB/OL]. 2022 [2022-11-30].https://cloud.tencent.com/document/product/583/9180

[6] Yang Yanan, Zhao Laiping, Li Yiming, et al. INFless: A native serverless system for low-latency, high-throughput inference[C]// Proc of the 27th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2022: 768−781

[7] Guo Jing, Chang Zihao, Wang Sa, et al. Who limits the resource efficiency of my datacenter: An analysis of Alibaba datacenter traces[C/OL]//Proc of the 27th IEEE/ACM Int Symp on Quality of Service. Piscataway, NJ: IEEE, 2019[2023-01-11].https://ieeexplore.ieee.org/document/9068614

[8] Qiu Haoran, Banerjee S S, Jha S, et al. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices[C]//Proc of the 14th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2020: 805−825

[9] Lo D, Cheng Liqun, Govindaraju R, et al. Heracles: Improving resource efficiency at scale[C]//Proc of the 42nd Annual Int Symp on Computer Architecture. New York: ACM, 2015: 450−462

[10] Chen Shuang, Delimitrou C, Martínez J F. PARTIES: QoS-aware resource partitioning for multiple interactive services[C]//Proc of the 24th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2019: 107−120

[11] Patel T, Tiwari D. CLITE: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers[C]//Proc of the 26th IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2020: 193−206

[12] Delimitrou C, Kozyrakis C. Quasar: Resource-efficient and QoS-aware cluster management[J]. ACM SIGPLAN Notices, 2014, 49(4): 127−144

[13] Agache A, Brooker M, Iordache A, et al. Firecracker: Lightweight virtualization for serverless applications[C]//Proc of the 17th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2020: 419−434

[14] Kaffes K, Yadwadkar N J, Kozyrakis C. Centralized core-granular scheduling for serverless functions[C]//Proc of the 10th ACM Symp on Cloud Computing. New York: ACM, 2019: 158−164

[15] Yu Tianyi, Liu Qingyuan, Du Dong, et al. Characterizing serverless platforms with serverlessbench[C]//Proc of the 11th ACM Symp on Cloud Computing. New York: ACM, 2020: 30−44

[16] AWS. AWS Lambda pricing[EB/OL]. 2022 [2022-12-19].https://aws.amazon.com/cn/lambda/pricing/

[17] IBM. IBM cloud function pricing[EB/OL]. 2022 [2022-12-19]. https://cloud.ibm.com/functions/learn/pricing

[18] Azure. Azure functions pricing[EB/OL]. 2022 [2022-12-19].https://azure.microsoft.com/en-us/pricing/details/functions/

[19] Schall D, Margaritov A, Ustiugov D, et al. Lukewarm serverless functions: Characterization and optimization[C]//Proc of the 49th Annual Int Symp on Computer Architecture. New York: ACM, 2022: 757−770

[20] Qiu Haoran, Jha S, Banerjee S S, et al. Is Function-as-a-service a good fit for latency-critical services[C/OL]//Proc of the 7th Int Workshop on Serverless Computing. New York: ACM, 2021[2023-04-21]. https://dl.acm.org/doi/abs/10.1145/3493651.3493666

[21] Zhao Laiping, Yang Yanan, Zhang Kaixuan, et al. Rhythm: Component-distinguishable workload deployment in datacenters[C/OL]//Proc of the 15th European Conf on Computer Systems. New York: ACM, 2020[2023-01-11].https://dl.acm.org/doi/abs/10.1145/3342195.3387534

[22] Roy R B, Patel T, Tiwari D. SATORI: Efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains[C]// Proc of the 48th ACM/IEEE Annual Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2021: 292−305

[23] Alipourfard O, Liu H H, Chen Jianshu, et al. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics[C]// Proc of the 14th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2017: 469−482

[24] Akhtar N, Raza A, Ishakian V, et al. COSE: Configuring serverless functions using statistical learning[C]//Proc of the 39th IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2020: 129−138

[25] Bilal M, Canini M, Fonseca R, et al. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions [J]. arXiv preprint, arXiv: 2105.14845, 2021

[26] Guo Zhiyuan, Blanco Z, Shahrad M, et al. Resource-centric serverless computing [J]. arXiv preprint, arXiv: 2206.13444, 2022

[27] Jaleel A, Hasenplaugh W, Qureshi M, et al. Adaptive insertion policies for managing shared caches[C]//Proc of the 17th Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2008: 208−219

[28] Sanchez D, Kozyrakis C. Vantage: Scalable and efficient fine-grain cache partitioning[C]//Proc of the 38th Annual Int Symp on Computer Architecture. New York: ACM, 2011: 57−68

[29] Nishtala R, Carpenter P, Petrucci V, et al. Hipster: Hybrid task manager for latency-critical cloud workloads[C]//Proc of the 23rd IEEE Int Symp on High Performance Computer Architecture.

Piscataway, NJ: IEEE, 2017: 409−420

[30] Cui Jiaxu, Yang Bo. Survey on bayesian optimization methodology and applications[J]. Journal of Software, 2018, 29(10): 3068−3090 (in Chinese)
(崔佳旭, 杨博. 贝叶斯优化方法和应用综述[J]. 软件学报, 2018, 29(10): 3068−3090)

[31] Kawaguchi K, Kaelbling L P, Lozano-Pérez T. Bayesian optimization with exponential convergence[C]//Proc of the 28th Int Conf on Neural Information Processing Systems. New York: ACM, 2015: 2809–2817

[32] El-Sayed N, Mukkara A, Tsai P A, et al. KPart: A hybrid cache partitioning-sharing technique for commodity multicores[C]//Proc of the 24th IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2018: 104−117

[33] Kim J, Lee K. FunctionBench: A suite of workloads for serverless cloud function service[C]//Proc of the 12th Int Conf on Cloud Computing. Piscataway, NJ: IEEE, 2019: 502−504

[34] Shahrad M, Balkind J, Wentzlaff D. Architectural implications of function-as-a-service computing[C]//Proc of the 52nd Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2019: 1063−1075

[35] Gao Wanling, Zhan Jianfeng, Wang Lei, et al. Data motif-based proxy benchmarks for big data and AI workloads[C]//Proc of the 14th IEEE Int Symp on Workload Characterization. Piscataway, NJ: IEEE, 2018: 48−58

[36] Zhang Yanqi, Hua Weizhe, Zhou Zhuangzhuang, et al. Sinan: ML-based and QoS-aware resource management for cloud microservices[C]//Proc of the 26th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2021: 167−181

[37] Bashir N, Deng Nan, Rzadca K, et al. Take it to the limit: Peak prediction-driven resource overcommitment in datacenters[C]//Proc of the 16th European Conf on Computer Systems. New York: ACM, 2021: 556−573



Guo Jing, born in 1994. PhD. Student member of CCF. Her main research interests include cloud computing and resource management.

郭　静, 1994 年生. 博士. CCF 学生会员. 主要研究方向为云计算和资源管理.



Hu Cunchen, born in 1991. PhD candidate. Student member of CCF. His main research interests include cloud computing and resource management. (duihuhu@gmail.com)

胡存琛, 1991 年生. 博士研究生. CCF 学生会员. 主要研究方向为云计算和资源管理.



Bao Yungang, born in 1980. PhD, professor, PhD supervisor. Member of CCF. His main research interests include computer architecture, operating system, open-source hardware, agile chip design, and datacenter architecture.

包云岗, 1980 年生. 博士, 教授, 博士生导师. CCF 会员. 主要研究方向为计算机系统结构、操作系统、开源硬件、处理器芯片敏捷设计、数据中心体系结构.