软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.007191]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

服务器无感知平台性能度量研究*

温金凤[1], 陈震鹏[2], 柳 熠[3], 刘譞哲[1]

[1](北京大学 计算机学院, 北京 100871)

[2](Department of Computer Science, University College London, London WC1E 6BT, UK)

[3](北京大数据先进技术研究院, 北京 100091)

通信作者: 刘譞哲, E-mail: xzl@pku.edu.cn

摘 要: 服务器无感知计算是一种新兴的云计算范型, 它允许开发者专注于应用逻辑的开发, 而不需要负责底层复杂的任务管理. 通过这种范型, 开发者可以快速构建更小粒度的应用, 即函数级别的应用. 随着服务器无感知计算的日益流行, 各大云计算厂商相继推出各自的商业服务器无感知平台. 然而, 这些平台的特点尚未得到系统的研究和可靠的比较. 全面分析这些特点可以帮助开发者选择合适的服务器无感知平台, 并以正确的方式开发和执行基于服务器无感知计算的应用. 为此, 开展了面向主流的商业服务器无感知平台特征的实证研究. 涵盖的主流服务器无感知平台包括亚马逊 Lambda、谷歌 Cloud Functions、微软 Azure Functions 和阿里巴巴 Function Compute. 研究内容主要分为两大类: 特征总结和运行时性能分析. 在特征总结中, 通过对这些服务器无感知平台的官方文档进行探究, 从开发、部署和运行时 3 个方面的关键特征进行总结和比较. 在运行时性能分析中, 我们使用代表性的基准测试程序, 从多个维度分析了这些服务器无感知平台提供的运行时性能. 具体而言, 首先分析了影响应用冷启动性能的关键因素, 如编程语言和内存大小. 其次, 探究了服务器无感知平台执行各类任务的执行性能. 基于特征总结和运行时性能分析的结果, 总结了一系列发现, 并为开发者、云计算厂商和研究者提供了具有现实指导意义的启示和潜在的研究机会.

关键词: 度量分析; 服务器无感知计算; 软件工程; 云计算; 平台比较

中图法分类号: TP311

中文引用格式: 温金凤, 陈震鹏, 柳熠, 刘譞哲. 服务器无感知平台性能度量研究. 软件学报. http://www.jos.org.cn/1000-9825/7191.htm

英文引用格式: Wen JF, Chen ZP, Liu Y, Liu XZ. Measurement Study on Performance of Serverless Platforms. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7191.htm

Measurement Study on Performance of Serverless Platforms

WEN Jin-Feng[1], CHEN Zhen-Peng[2], LIU Yi[3], LIU Xuan-Zhe[1]

[1](School of Computer Science, Peking University, Beijing 100871, China)

[2](Department of Computer Science, University College London, London WC1E 6BT, UK)

[3](Advanced Institute of Big Data, Beijing 100091, China)

Abstract: Serverless computing is an emerging paradigm of cloud computing, allowing developers to focus only on application logic development without the need to manage complex underlying tasks. This paradigm allows developers to quickly build smaller-granularity applications, the one at the function level. With the increasing popularity of serverless computing, major cloud computing vendors have introduced their commercial serverless platforms one after another. However, the characteristics of these platforms have yet to be systematically studied and reliably compared. A comprehensive analysis of these characteristics can help developers choose an appropriate serverless platform while developing and executing serverless applications in the right way. To this end, an empirical study is conducted

* 基金项目: 国家重点研发计划 (2022YFB4500700); 国家杰出青年科学基金 (62325201)
收稿时间: 2023-07-13; 修改时间: 2023-10-10, 2023-11-30; 采用时间: 2024-03-27; jos 在线出版时间: 2024-06-14

on the characteristics of mainstream commercial serverless platforms. This study involves such mainstream serverless platforms as AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, and Alibaba Function Compute. This study is divided into two major parts: feature summarization and runtime performance analysis. In the feature summarization, the official documents of these serverless platforms are discussed and their key features are summarized and compared in terms of development, deployment, and runtime. In the runtime performance analysis, representative benchmarks are applied to analyze the runtime performance offered by these serverless platforms on a multidimensional basis. Specifically, key factors for the cold-start performance of the applications are first analyzed, such as programming languages and memory sizes. Furthermore, the tasks-executing performance of serverless platforms is discussed. Based on the results of feature summarization and runtime performance analysis, this study sums up a series of findings and provides practical insights and potential research opportunities for developers, cloud computing vendors, and researchers.

Key words: measurement study; serverless computing; software engineering; cloud computing; platform comparison

服务器无感知计算是云计算的一种新兴范型, 在视频处理[1,2]、机器学习[3,4]、科学计算[5]和大数据分析[6,7]等领域得到广泛应用. 它使开发者摆脱了底层云平台复杂且容易出错的管理任务, 如负载均衡、监控和伸缩性, 从而专注于应用逻辑的开发. 该范型的核心概念是函数即服务 (function-as-a-service, FaaS), 表示应用可以由多个独立的函数组成, 这些函数称为服务器无感知函数, 用于实现特定的小任务. 由一个或多个服务器无感知函数组成的应用称为函数即服务应用 (FaaS 应用). 服务器无感知计算采用按需付费的友好模式, 开发者根据实际资源消耗或分配进行付费, 计费粒度精确到毫秒级. 服务器无感知计算不仅使开发者受益, 也为云计算厂商提供了更好的资源利用和分配方式. 主流的云计算厂商已经推出了相应的服务器无感知平台, 如亚马逊的 Lambda[8]、谷歌的 Cloud Functions[9]、微软的 Azure Functions[10]以及阿里巴巴的 Function Compute[11]. 据预测, 到 2025 年, 预计全球有 50% 的企业将采用服务器无感知计算[12]. 此外, 市场规模也将从 2017 年的 30 亿美元增长到 2025 年的近 220 亿美元[13].

开发者广泛使用服务器无感知平台开发和执行他们的任务[14,15]. 然而, 不同的服务器无感知平台有着不同的开发特点和运行时表现. 在这样的情况下, 不同的服务器无感知平台和不同的配置使得获得的性能表现不一致. 开发者需要一个全面的指导来告诉他们选择哪个服务器无感知平台可以满足他们的应用需求, 以及如何配置参数才能保证应用执行的可用性. 一些度量研究工作[16,17]对开源的服务器无感知平台 (如 OpenWhisk[18]、Knative[19]) 进行了性能度量分析. 然而, 由于开源平台在实际应用中需要额外的平台部署工作和工程努力, 这可能成为开发者的挑战. 考虑到服务器无感知计算的独特优势, 即减轻开发者底层管理任务的负担, 使开发者能够更专注于应用层面的开发, 因此, 许多开发者更倾向于使用商业的公有云服务器无感知平台, 因为它们提供了更简单易用的部署流程和应用管理功能, 这对提供生产力和减少开发周期具有显著的优势. 此外, 服务器无感知计算概念的广泛流行也直接源自公有云的亚马逊 Lambda 服务的提出. 其他云提供商紧跟其后, 相继推出了类似的服务, 将服务器无感知计算引入主流. 这一趋势对云计算领域产生了深远的影响. 然而, 先前开源平台性能度量的分析结果[16,17]并不能泛化到广泛使用的商业服务器无感知平台, 且相应的指导不适用于相关的开发者. 因此, 需要对基于公有云的服务器无感知平台开展性能度量分析, 以指导广泛使用服务器无感知计算范型的开发者做出明确的决策. 我们提出关于商业服务器无感知平台的性能度量工作[20], 但是该工作没有比较不同平台在执行不同类型任务的执行性能. 任务的执行效率是开发者重点关注的应用指标. 另外, 该工作中对冷启动延迟的估算是基于两次调用的时间差, 但第二次调用产生的实例调度延迟往往是无法忽略的, 不能假定认为没有延迟. 因此, 获得的冷启动延迟可能会偏低, 导致结果不准确. 此外, 该工作在性能度量实验中对每组实验开展的重复次数较低, 可能导致结论不可靠. 因此, 现有的性能度量相关工作对主流的公有云服务器无感知平台性能的全面比较和可靠分析方面存在不足, 这使得现有研究难以准确地提供关于应用的平台选择和配置实践的指导和启示.

为了填补这一知识空白, 在本文中, 我们将开展这样一项面向主流的公有云服务器无感知平台的度量研究. 本文主要关注的服务器无感知平台, 包括亚马逊 Lambda、谷歌 Cloud Functions、微软 Azure Functions 和阿里巴巴的 Function Compute. 这些平台是当前市场上广泛使用的平台, 由云计算领域的领先云厂商提供, 并且在功能、性能和生态系统等方面都有一定的优势. 通过对这些公有云平台进行深入的研究和多维度的分析, 旨在为软件开发者提供指导和对比, 帮助他们更好地了解和选择适合自己需求的服务器无感知平台. 本文的目标不在于提出新的

方法, 而更重要的是在实际商业环境中应用已被广泛认可的可靠分析方法, 以展示出新的可靠的公有云平台性能度量结果.

在本文的度量研究中, 主要采用了特征总结和运行时性能分析两种方式. 具体而言, 在特征总结方面, 我们主要考虑我们先前的工作[20]是发表在 2020 年, 开展新的度量分析之前需要了解一些关键的平台配置信息. 因此, 本文总结并更新了不同服务器无感知平台在应用开发、部署和运行时 3 方面的特点和使用限制. 例如, 开发方面总结支持的编程语言和触发器等; 部署方面总结部署包大小限制、内存分配大小和超时时间限制等; 运行时方面总结调用类型和付费模型等. 通过这样的总结, 可以帮助开发者更好地理解服务器无感知平台所支持的具体特性, 从而促进进一步的应用开发实践. 开发者可以根据这些特点来选择最适合他们需求的平台, 并确保他们的应用能够在所选平台上顺利运行.

在运行时性能分析方面, 本文通过多个维度对服务器无感知平台的实际运行性能进行全面可靠的探究, 以帮助开发者根据其应用特点选择合适的服务器无感知平台, 从而提高应用性能. 在服务器无感知计算领域, 冷启动性能一直是学术界和工业界广泛关注的一个重要挑战, 提供可靠的冷启动性能分析是重要的. 除此之外, 开发者任务执行在服务器无感知平台上的执行效率也是他们关注的重要指标, 以更好地服务用户. 因此, 我们研究平台准备应用运行时环境的冷启动性能以及实际执行应用的执行性能. 我们的方法设计如下.

首先, 我们分析了不同服务器无感知平台所产生的冷启动性能. 考虑到冷启动延迟并非开发者应用的实际执行延迟, 但其大小将会影响应用的整体服务质量. 因此, 我们进一步探究可能影响冷启动延迟的因素, 例如编程语言和内存大小. 其次, 我们对函数即服务应用的运行时性能进行度量, 通过一组代表性的基准测试程序来比较不同服务器无感知平台的执行效率. 代表性的基准测试程序分为两类: 微基准测试程序和宏基准测试程序. 具体而言, 微基准测试程序由一组简单的工作负载组成, 专注于特定的资源消耗, 如 CPU、内存、网络和磁盘 I/O; 宏基准测试程序由一组真实的任务组成, 例如图片处理、语音识别、图计算、机器学习训练和机器学习推理应用, 这些应用需要消耗各种系统资源. 通过以上方法, 我们能够全面评估不同服务器无感知平台的冷启动性能和应用执行性能, 帮助开发者提升应用性能.

基于特征总结和运行时性能分析的结果, 我们得出一系列深入的发现, 并为开发者、云计算厂商以及研究者提供了具有重要实际指导意义的启示和潜在的研究机会. 对于开发者而言, 本文的研究发现可以帮助他们在选择服务器无感知平台时做出明智决策. 通过深入分析不同平台的特征和运行时性能, 开发者能够根据自身的应用特点选择最合适的平台和配置, 以实现最佳性能. 此外, 本文还强调了冷启动延迟和任务执行延迟等关键指标, 这些指标可作为评估和优化应用性能的重要参考依据. 对于云计算厂商而言, 本文的研究结果可以为他们改进和优化服务器无感知平台提供有益的指导. 云计算厂商可以有针对性地改进平台的设计和实施, 以提升冷启动性能和资源效率等方面, 此外, 本文的研究还揭示了开发者的关注重点和需求, 云计算厂商可基于这些信息优化平台的功能和服务, 提高更优质的开发体验和性能支持. 对于科学研究而言, 我们的研究结果提供了一系列有潜力的研究机会. 例如, 如何将长时应用分解为短时函数, 如何单独配置和学习内存和 CPU, 如何对不同应用场景进行性能优化等. 另外, 为了促进其他研究者的复现工作和进一步研究, 我们在 GitHub[21]上开源了本文所使用代码和数据, 包括所使用的基准测试程序.

本文第 1 节介绍了研究背景及相关工作. 第 2 节介绍了本文的方法设计, 涵盖了所使用的基准测试程序详情和实验设置等. 第 3 节展示了对不同的服务器无感知平台的特征总结的结果. 第 4 节介绍了不同服务器无感知平台展现的运行时性能分析的结果. 第 5 节讨论了本文的结果对开发者和云计算厂商的启示, 以及对研究者有潜力的研究机会. 第 6 节讨论了本文的局限性. 最后, 第 7 节总结全文.

1 研究背景及相关工作

云计算已成为一种广泛采用的范型, 通过互联网提供计算服务. 主要的云计算服务模式包括基础设施为服务 (infrastructure-as-a-service, IaaS)、平台即服务 (platform-as-a-service, PaaS) 和软件即服务 (software-as-a-service, SaaS). 它们在云计算堆栈中形成了层次结构, 逐层减轻了繁琐和容易出错的基础设施管理任务[22], 例如负载均衡

和自动扩展. 在云计算领域, 服务器无感知计算是一种新的范型, 为应用开发带来了全新的机遇. 它使开发者能够专注于应用逻辑, 节省成本, 而无需深入了解云计算专业知识. 函数即服务 (function-as-a-service, FaaS) 是服务器无感知计算的核心, 将函数作为基本单位. 开发者只需编写特定事件触发器的函数, 打包上传代码, 并指定所需运行配置. 然后, 服务器无感知平台提供沙盒环境来托管函数, 并自动处理可伸缩性、容错和其他运行时问题. 这与传统的云计算不同, 传统云计算通常需要开发者管理和运行服务器, 并支付相关费用. 然而, 服务器无感知计算的编程范型抽象了大部分底层操作任务, 简化了云上的应用开发过程. 同时, 它的存储和计算是单独扩展的, 并具有独立的配置和定价方式.

服务器无感知计算的优点和广阔前景使其在各种领域得到广泛应用, 例如视频处理[1,2]、机器学习[3,4]、科学计算[5]和大数据分析[6,7]等. 图 1 展示了开发者在开发、部署和执行面向服务器无感知计算应用的过程.

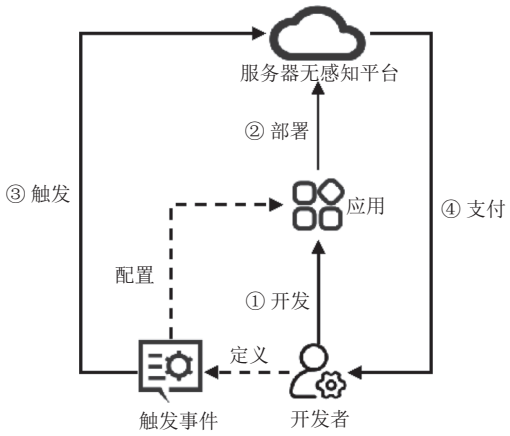

图 1 开发、部署和执行面向服务器无感知计算的应用的过程

(1) 开发: 开发者使用高级语言 (如 Python 和 JavaScript) 编写服务器无感知计算类型的应用, 将其实现为一组无状态且事件驱动的函数. 此外, 开发者可以借助成熟云服务 (如云存储和云数据库) 来辅助功能的实现. 同时, 通过指定的配置文件, 开发者可以定义具体的事件规则, 将函数与触发事件进行绑定. 在配置文件中, 开发者还可以为函数配置所需的云资源和权限.

(2) 部署: 函数及其依赖包被打包在一起, 并通过命令行界面或控制台部署到服务器无感知平台上. 在此期间, 开发者需要提供一些必要的运行配置, 如运行时环境和内存大小. 平台能够提供必要的资源和限制, 例如在亚马逊 Lambda 中, 函数执行时间限制为 15 min, 以实现对请求的灵活和可扩展处理.

(3) 触发: 当预定义的事件触发服务器无感知函数时, 服务器无感知平台将自动准备其运行环境. 在新启动的实例上执行的函数将经历冷启动过程, 并产生不可忽略的冷启动延迟. 具体来说, 冷启动延迟是准备执行环境的延迟, 主要包括初始化虚拟机或容器、通过网络传输应用、加载应用代码等. 当同一函数的后续请求在很短时间内到达 (如亚马逊的 7 min), 已启动的函数实例将被重用, 并经历热启动过程以产生热启动延迟. 热启动延迟是指函数功能真正执行之前产生的延迟, 它表示平台调度可用实例来服务请求的时间. 一般来说, 热启动延迟比冷启动延迟更低. 如果没有后续请求, 这些实例将进入空闲状态, 相应的资源会自动释放.

(4) 支付: 在函数执行完毕后, 开发者只需要为实际消耗或分配的资源量付费. 服务器无感知平台提供了一种细粒度的计费模式, 它是基于函数的毫秒级执行单元.

目前, 许多研究者致力于性能度量分析[23–29], 这是发现软件性能特性的有效手段之一, 可以评估其运行速度、资源利用率、可靠性等指标, 并确定需要改进和优化的方面. 在服务器无感知计算研究中, 性能度量工作主要包括以下几类.

第 1 类工作是为服务器无感知计算领域提供性能度量的基准测试程序集. 例如, Yu 等人[24]提出了一套名为

ServerlessBench 的基准测试程序集, 用于研究服务器无感知函数之间的通信效率、冷启动性能、无状态开销和性能隔离等特性. 这项工作的度量结果表明, 将应用分解为一组服务器无感知函数是有利于节省成本并提高应用性能. 然而, 服务器无感知计算中的无状态特性会影响服务器无感知函数的执行效率. Maissen 等人[25]和 Kim 等人[26]分别设计了开源的基准测试程序集 FaaSDom 和 FunctionBench, 以促进研究者对服务器无感知平台性能的研究. 这两个基准测试程序集都提供了微基准测试程序集, 而 FunctionBench 还提供了一些复杂的宏基准测试程序来表示真实的函数即服务应用.

第 2 类工作是分析服务器无感知计算的工作负载性能特点. 为了更好地理解函数即服务应用的运行时特点, Shahrad 等人[27]分析了微软 Azure Functions 在生产环境中的工作负载的特点. 他们发现, 大多数服务器无感知函数的调用频率很低, 存在 8 个数量级的调用频率范围. 此外, 服务器无感知函数使用了各种类型触发器, 产生的调用模式往往难以预测. 在资源需求方面, 它们显示出 4 倍的函数内存使用范围, 并且有 50% 的服务器无感知函数运行时间少于 1 s. Zhang 等人[28]分析了基于视频处理任务的服务器无感知函数的性能, 并发现内存分配对于视频处理类型的应用至关重要, 而配置最大内存并不总是获得最佳性能结果, 这表明动态分析工作负载的资源需求是必要的, 以进一步确定最佳的函数内存配置.

第 3 类工作是探究服务器无感知平台的底层机制和性能问题. 例如, Wang 等人[29]启动了超过 5 万个函数实例, 以揭示服务器无感知平台的架构、性能和资源管理有效性. 这项工作揭示了服务器无感知平台如何使用虚拟机或容器来隔离不同账户的服务器无感知函数, 这在安全方面具有重要的意义. 此外, 这项工作还揭示了亚马逊 Lambda 采用类似装箱策略来最大化虚拟机的内存利用率.

第 4 类工作是比较不同服务器无感知平台. 首先, 现有工作对多个开源服务器无感知平台的性能展开度量分析. 例如, Li 等人[16]探讨了 4 个开源平台的整体架构和关键构件, 评估了特定设计的影响和平台的自动伸缩能力. Mohanty 等人[17]分析了 3 个开源平台的响应时间和请求成功率, 强调了它们在不同负载条件下的特征. 而 McGrath 和 Brenner[30]提出了一个服务器无感知计算系统原型, 并通过与亚马逊 Lambda、微软 Azure Functions、谷歌 Cloud Functions 和 Apache OpenWhisk 的比较, 验证了该原型设计在并发和退避测试方面的良好效果. Palade 等人[31]针对边缘计算环境, 对 4 个开源平台的吞吐量、请求成功率和部署难易程度进行了深入研究.

本研究的关注点在于比较主流公有云平台的冷启动性能以及在执行不同类型任务时这些平台所产生的性能表现. 因此, 本研究与这些面向开源平台的度量工作的关注点有所不同. 而且, 这些面向开源平台的度量工作所获得的分析结果并不能泛化到广泛使用的公有云服务器无感知平台, 且相应的指导不适用于相关的开发者. 另外, 一些其他工作致力于对公有云服务器无感知平台进行性能度量. Wang 等人[29]深入探讨了亚马逊 Lambda、谷歌 Cloud Functions 和 Azure Functions 这 3 个公有云平台在资源管理和性能隔离方面的特征, 而 Lloyd 等人[32]调查了亚马逊 Lambda 和微软 Azure Functions 平台在不同冷热状态下的基础设施管理特点. Ustiugov 等人[33]虽然探究了影响冷启动性能的因子, 但他们工作的主要目的是提出一个开源度量框架, 并通过研究公有云平台延迟性能差异的原因, 如函数实例本身、数据通信和集群调度等方面, 为这一领域的研究做出了贡献. 与上述工作的不同之处在于, 本文研究的主要目的在于为开发者提供实际指导, 深入总结其应用开发流程中的关键特征和实践. 因此, 本文研究不仅对不同关键特征进行了总结和全面比较, 还深入研究了不同任务类型在不同公有云平台和不同配置下的性能表现, 以为进一步的平台选择和任务开发、部署提供有益的指导信息. 总体而言, 本文研究的关注点与现有度量工作存在明显区别, 强调为实际应用场景提供有针对性的实用指导. 本文旨在进行一项全面可靠的关于主流公有云服务器无感知平台性能的度量研究, 以准确地提供关于应用的平台选择和配置实践的指导和启示. 具体而言, 我们主要关注亚马逊 Lambda、谷歌 Cloud Functions、微软 Azure Functions 和阿里巴巴 Function Compute 这些主流商业平台. 在本文研究中, 主要目标是总结并更新不同应用阶段下各个服务器无感知平台的关键特征, 并从多个维度对这些平台的实际运行性能展开可靠的分析和全面的比较.

2 方法设计

本节介绍本文所提出的度量分析框架, 如图 2 所示. 该度量框架的目标是全面分析和可靠比较 4 个主流的商

业服务器无感知平台的特点, 包括亚马逊 Lambda、谷歌 Cloud Functions、微软 Azure Functions 和阿里巴巴 Function Compute. 这 4 个平台的发布时间分别为: 亚马逊 Lambda 于 2014 年 11 月发布, 谷歌 Cloud Functions 于 2018 年 7 月发布, 微软 Azure Functions 于 2016 年 11 月发布, 阿里巴巴 Function Compute 于 2017 年 4 月发布. 在本文的度量框架中, 研究目标主要包括两方面. 首先, 对不同服务器无感知平台的关键特征进行全面总结和分析. 其次, 对不同类型任务在不同服务器无感知平台上的运行时性能进行评估和比较.

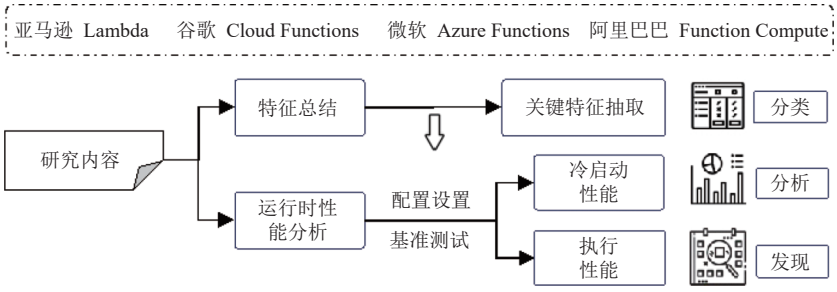

图 2 本文的度量分析框架

在特征总结中, 我们目标是为了更新关键特征的信息, 并帮助开发者理解不同服务器无感知平台的开发实践特点, 并判断他们的应用是否适合在目标平台上实现或执行. 为了实现这一目标, 我们收集了来自 4 个平台的官方文档, 然后从这些文档中提取并总结了关键特征, 这些特征反映了平台的内在限制和特点. 具体而言, 我们提取关于应用开发、部署和运行时的特点. 在开发方面, 我们总结了平台支持的编程语言、触发器类型和开发方式等特点, 这些特点帮助开发者了解在每个平台上使用哪些编程语言, 以及触发器的类型和如何进行开发. 在部署方面, 我们总结了部署包大小限制、内存分配大小、CPU 和 GPU 情况、超时时间限制、本地磁盘大小和支持区域等特点, 这些特点对开发者也至关重要, 因为它们决定了应用在平台上的资源限制和可用性. 在运行方面, 我们总结了调用类型、负载大小、运行环境和付费模型等特点, 这些特点帮助开发者了解平台支持的运行特征, 从而更准确地评估平台的适用性.

在运行时性能分析中, 我们目标是从多个维度对在不同服务器无感知平台上运行的应用性能进行分析. 考虑到函数即服务应用的整体性能主要受两方面延迟的影响. 首先是准备执行环境所需的冷启动延迟, 它反映了冷启动性能. 其次是执行应用本身功能所需的延迟, 代表了执行性能. 因此, 我们探究不同服务器无感知平台在执行应用时产生的冷启动性能和执行性能. 在服务器无感知计算的场景下, 任务通常在毫秒级别内完成, 这使得冷启动延迟的开销变得相当显著. 对开发者来说, 以较低的冷启动延迟执行任务非常重要, 因为它直接影响用户体验. 因此, 理解并优化冷启动延迟成为服务器无感知计算领域中一个关键的挑战. 另外, 不同类型的任务对资源的需求也各不相同. 了解不同类型任务在不同服务器无感知平台上的执行性能对开发者同样重要, 这有助于他们挑选执行效率最高的平台来托管他们的应用. 因此, 本文的运行时性能分析旨在探究不同服务器无感知平台在执行不同类型任务时的冷启动性能和执行性能.

为了实现全面可靠的冷启动性能和执行性能分析和比较, 我们还设计和实现模块化和可扩展方法, 为不同平台提供了相似的功能接口, 包括应用打包、应用部署和创建、应用调用以及返回统一风格的日志信息, 同时还支持应用删除. 开发者可以为使用的基准测试程序指定特定的配置来部署、创建和调用函数即服务应用. 调用后返回的日志信息采用统一的风格, 方便比较和分析不同应用在不同平台下的冷启动性能和执行性能. 在配置设置方面, 开发者可以配置编程语言运行时和内存大小. 默认情况下, 我们使用 HTTP 触发器, 因为它是不同平台上都支持且最常用的事件触发器. 对于基准测试程序, 我们主要构建两类基准测试程序集, 以探究不同类型任务的运行性能.

2.1 基准测试程序集

本节介绍本文使用的基准测试程序集, 分为微基准测试程序集和宏基准测试程序集两部分.

微基准测试程序集主要包含了一系列简单的工作负载, 具体的信息展示在表 1 中, 包括 Hello World 任务以及

消耗特定资源的任务, 如 CPU 密集型任务、内存密集型任务、磁盘 IO 密集型任务和网络密集型任务. Hello World 任务的实现涵盖了不同编程语言版本, 旨在在运行时性能分析中探究不同服务器无感知平台下执行不同编程语言应用下的冷启动性能. 消耗特定资源的任务旨在在运行时性能分析中研究不同类型资源消耗的应用在不同平台上的执行性能. 在这些任务中, CPU 密集型性任务实现了线性方程求解, 它需要消耗更多的计算量. 内存密集型任务实现了斐波那契数列的递归计算, 它需要消耗大量内存. 磁盘 IO 密集型任务实现了对磁盘内容的写和压缩操作, 它需要频繁地进行磁盘读写. 网络密集型任务实现了对特定网页内容的读取并返回数据, 它需要借助网络传输数据.

表 1　微基准测试程序

任务类型	任务缩写表示	任务描述
Hello World任务	Hello_task	打印Hello World消息, 实现为不同编程语言版本
CPU密集型任务[34]	CPU_task	线性方程求解, 输入为矩阵大小
内存密集型任务[35]	Memory_task	采用递归函数计算斐波那契数列, 输入为要计算的数列元素下标
磁盘IO密集型任务[36]	DiskIO_task	将磁盘的内容进行读和写, 输入为文件大小
网络密集型任务[37]	Network_task	从网页中读取数据并进行反序列化, 输入为一个网页链接

宏基准测试程序集包含了一系列消耗多种系统资源的真实任务, 这些任务的具体信息展示在表 2 中. 这些任务包括图片处理任务、语音识别任务、图计算任务、机器学习训练任务和机器学习推理任务. 图片处理任务实现了图片转化的功能任务. 它首先从云存储中获取待处理的图片, 然后应用不同的转换效果, 如旋转和调色. 最后, 更新后的图片会再次上传到云存储. 语音识别任务实现了对语音文件进行分批处理, 将其转化为文本. 图计算任务实现了图网络的构建. 在该网络中, 节点之间的连接是基于目标节点的连接数概率分布来建立的. 然后, 对生成的图中的节点进行重要程度排序. 机器学习训练任务利用食物评论的文本数据集来训练一个逻辑回归模型. 机器学习推理任务利用已训练好的逻辑回归模型来对输入的用户评论文本进行情感分数的预测.

表 2　宏基准测试程序

任务类型	任务缩写表示	任务描述
图片处理任务[38]	Image_task	实现对图片的翻转、旋转、过滤、调色和缩略图转化, 输入为图片位置
语音识别任务[39]	Speech_task	实现对语音文件内容的批处理, 将其转化为文本
图计算任务[40]	Graph_task	创建一个图网络, 并选择出重要的节点, 输入为图大小
机器学习训练任务[41]	Train_task	使用食物评论文本数据集进行逻辑回归模型训练, 可预测评论的情感分数
机器学习推理任务[42]	Inference_task	对任意评论进行情感预测, 得到一个情感分数

2.2　实验设置

本节描述关于运行时性能分析的实验设置. 为了全面可靠地评估平台的运行时性能特性, 我们考虑应用在不同云环境下可能存在显著的性能差异. 现有工作仅使用少数测量次数进行分析和比较 (如 10 次[17]或 20 次[20]重复执行), 这种方法可能不足以得出可靠的结论. 为了获得更可靠的结果, 我们参考了现有的服务器无感知计算研究中使用的重复次数[1,2,5–7,43], 这些重复次数都不超过 1 000 次. 因此, 本文对每组实验测量都进行了最高的重复次数, 即 1 000 次, 以获得性能数据分布情况. 为了进行公平比较, 我们将基准测试程序部署到各个服务器无感知平台的相同服务地区. 这确保了在更高粒度的节点服务区域上保持一致, 以最大程度地减少网络开销的影响, 这样也可以更相对公平地比较各个平台的性能. 本文以美国东部区域为例, 具体为亚马逊 Lambda 的 us-east-1、谷歌 Cloud Functions 的 us-east1、微软 Azure Functions 的 eastus 和阿里巴巴 Function Compute 的 us-east-1. 另外, 在性能数据分析方面, 我们使用所获得的 1 000 组性能数据的分布、中位数和分位数来比较不同服务器无感知平台执行任务的能力. 为了清晰地展示性能数据的分布情况, 我们使用箱线图的形式进行展示. 实验的设计、分析和总结、应用部署以及性能数据获取是在 2023 年 4 月 26 日–2023 年 7 月 15 日之间进行.

在本文的运行时性能分析中, 我们对冷启动性能和执行性能进行了评估, 它们分别是在冷启动和热启动下调用应用所获得的结果. 为了获得冷启动性能数据, 我们在释放之前调用应用中服务器无感知函数的资源后, 然后再次调用函数, 这将使应用经历冷启动. 为了确保函数是以冷启动方式调用的, 我们使用了 20 分钟的调用频率. 通过这种方式, 我们可以获得应用的冷启动延迟. 对于执行性能数据的获取, 我们在平台释放先前调用的函数资源之前来调用应用. 因此, 我们在函数经历第一次冷启动调用后还存在函数资源的情况下, 使用 20 秒的调用频率来再次执行函数以获得热启动下的执行性能. 通过这种方式, 我们可以获得应用的执行延迟. 为了获得可比较的冷启动延迟和执行延迟的数据, 考虑到不同服务器无感知平台返回内容的不一致, 本文采用了时间戳获取的方式. 具体来说, 冷启动延迟是指从请求开始到服务器无感知函数任务实际开始执行之前的时间, 而执行延迟是函数任务实际执行的时间. 我们在请求开始时和服务器无感知函数代码体的最开始位置处设置时间断点, 以获得冷启动延迟. 特别地, 冷启动延迟是从请求发出的时间开始计算的. 这种冷启动性能数据的获取方式是已被业界和学术界[24,44,45]广泛认可的合理方法. 对于执行延迟的获取, 我们通过在服务器无感知函数代码体的最开始位置处和请求结束后分别设置时间断点来获取时间. 最后, 基于规范化的日志信息输出, 我们对不同服务器无感知平台进行了分析和比较.

3 特征总结结果

在本节中, 我们总结并更新关于不同主流服务器无感知平台的关键特征分析结果. 具体而言, 我们比较亚马逊 Lambda、谷歌 Cloud Functions、微软 Azure Functions 和阿里巴巴 Function Compute 在应用开发、部署和运行时阶段的关键特点或使用限制. 以下所列举的特征信息可能和我们之前工作所获得的结果存在差别, 我们在梳理特征的同时更新其具体内容或限制.

3.1 开发特点

对于应用开发特点, 我们在表 3 中总结支持的编程语言、触发器类型、开发方式和工作流支持性.

表 3 不同服务器无感知平台的开发特点

特点	亚马逊Lambda	谷歌Cloud Functions	微软Azure Functions	阿里巴巴Function Compute
支持的编程语言	JavaScript、Python、Java、Ruby、Go、.Net、自定义	JavaScript、Python、Java、Ruby、Go、PHP、.Net、自定义	JavaScript、Python、Java、PowerShell、TypeScript、自定义	JavaScript、Python、Java、PHP、.Net、Go、自定义
触发器	HTTP请求、亚马逊云服务	HTTP请求、谷歌云服务	HTTP请求、微软云服务	HTTP请求、阿里巴巴云服务
开发方式	控制台、命令行工具、API、软件开发工具包	控制台、命令行工具、API、软件开发工具包	控制台、命令行工具、API、软件开发工具包、Visual Studio、Visual Studio Code	控制台、命令行工具、API、软件开发工具包
工作流支持性	支持(Step Functions)	支持(Cloud Workflows)	支持(Azure Durable Functions)	支持(Serverless Workflow)

不同的服务器无感知平台对于支持的编程语言有所不同. 在亚马逊 Lambda 上, 可以使用 JavaScript、Python、Java、Ruby、Go 和.Net 等语言编写函数. 谷歌 Cloud Functions 支持 JavaScript、Python、Java、Ruby、Go、PHP 和.Net 语言等. 微软 Azure Functions 支持 JavaScript、Python、Java、PowerShell 和 TypeScript 语言等. 而阿里巴巴 Function Compute 则支持 JavaScript、Python、Java、PHP、.Net 和 Go 语言等. 此外, 这 4 个平台均支持使用自定义的方式部署和创建应用, 这种方式允许开发者使用任何编程语言来开发应用. 从比较中可以发现 JavaScript、Python 和 Java 是这些平台都广泛支持的编程语言, 也是服务器无感知计算社区中被广泛使用的语言[14]. 相比于之前工作报道的支持语言类型, 谷歌 Cloud Functions 增加了对 PHP 语言的支持, 阿里巴巴 Function Compute 增加了对 Go 语言的支持. 支持更多的编程语言有助于吸引更多的开发者使用这些服务器无感知平台. 不同开发者可能擅长的编程语言不同, 因此扩展编程语言的支持可以扩宽平台的用户基础. 而且, 支持多种编程语言

可以丰富平台的生态系统, 这意味着开发者可以更容易地集成已有的库、框架和工具, 从而提高开发效率. 不同的编程语言也适用于不同的用例和需求, 例如 Python 通常用于数据分析和智能化任务, 而 Go 通常用于高性能后端开发. 多语言支持意味着开发者可以根据任务需求选择最合适的语言.

对于触发器类型, 4 个平台上的函数都可以通过 HTTP 请求触发. 此外, 这些平台上的函数还可以被各自平台上的云服务触发. 考虑到先前的工作并没有具体指出可触发的云服务类型, 在本文中, 我们列出不同服务器无感知平台目前所支持触发的一些云服务. 在亚马逊 Lambda 上, 函数可以被 API 网关、S3 云存储、DynamoDB 数据库、Kinesis 数据流、SQS 队列、SES 邮件、SNS 通知、CloudFormation 资源管理和更新、CloudFront 内容分发网络、CloudWatch Logs 日志流、CodeCommit 代码托管、CodePipeline 持续交付、Cognito 身份验证、IoT 物联网、Lex 智能语音交付和 MQ 消息代理云服务触发; 在谷歌 Cloud Functions, 函数可以被 Pub/Sub 主题、Storage 存储、Firestore 数据库云服务触发; 在微软 Azure Functions 上, 函数可以被 Blob 存储、Kafka 主题、Cosmos DB 数据库、SQL 关系数据库、事件网格、事件中心、IoT 中心、队列存储、SendGrid 电子邮件、SignalR 实时 Web 和定时器云服务触发; 在阿里巴巴 Function Compute 上, 函数可以被对象存储 OSS、API 网关、定时器、日志 SLS、内容分发 CDN、表格存储 Tablestore、消息 MNS、消息队列 MQ 版、消息队列 Kafka 版、消息队列 MQTT 版和数据流 DTS 云服务触发. 支持多种触发器类型允许开发者根据任务的具体需求选择合适的触发器. 而且, 这些服务器无感知平台支持与特定云服务的集成, 例如存储、数据库、消息队列等, 且这些云服务具有伸缩能力, 适应于服务器无感知计算的关键特性. 这使得开发者可以轻松构建可伸缩的复杂云应用, 且充分利用云计算生态系统的各种功能. 然而, 不同触发器类型和云服务的使用也可能增加开发者的学习负担, 开发者需要进一步了解它们的工作方式和配置.

在服务器无感知平台上进行应用开发时, 这 4 个平台都提供了图形化控制台、命令行工具、API 和软件开发工具包 (SDK) 进行应用开发. 此外, 微软 Azure Functions 还可以使用 Visual Studio Code 工具以及 Visual Studio 进行开发. 然而, 先前的研究忽略了 Visual Studio 工具. 通过分析微软 Azure Functions 的更新历史, 我们发现 Visual Studio 和 Visual Studio Code 工具一直以来受到微软 Azure Functions 的积极支持. 提供多种开发工具选项有助于满足不同开发者和团队的偏好. 对于某些团队和开发者, 他们可能已经在使用 Visual Studio 或 Visual Studio Code 等工具. 微软 Azure Functions 的支持使他们能够无缝集成这些工具, 从而提高开发效率. 然而, 针对特定平台的开发工具通常会带来平台依赖性. 这也使得服务器无感知计算社区出现了面向不同平台的编程开发框架, 如 Serverless Framework[46]. 在本文后续的运行时性能分析实验中, 我们也将使用命令行工具或软件开发工具包来创建、部署和调用不同的基准测试程序.

在工作流支持方面, 4 个云服务提供商都支持基于函数服务的工作流构建. 但是, 每个云服务提供商所使用的工作流服务不同. 亚马逊使用 Step Functions 服务来编排 Lambda 函数, 这个服务自 2016 年 12 月发布以来已得到广泛应用[47,48]. 谷歌使用 Cloud Workflows 服务来编排 Cloud Functions 函数, 该服务在 2020 年 8 月发布. 微软使用 Azure Durable Functions 来编排 Azure Functions 函数, 该服务自 2018 年 5 月发布以来也变得流行[49]. 阿里巴巴 Function Compute 使用 Serverless Workflow 服务来编排函数, 该服务自 2019 年 7 月发布以来逐渐崭露头角[48]. 每个工作流服务各自使用特定的语言或形式来完成函数之间的编排. 例如, 亚马逊 Step Functions 使用状态定义语言, 微软 Azure Durable Functions 使用代码形式的编排函数. 总结来说, 使用工作流服务可以轻松编排和协调多个函数, 使它们在特定顺序和条件下执行. 亚马逊 Step Functions 和阿里巴巴 Serverless Workflow 还提供了可视化界面, 允许用户检查工作流的创建流程.

发现: JavaScript、Python 和 Java 是 4 个平台都支持的编程语言. 谷歌 Cloud Functions 增加了对 PHP 语言的支持, 阿里巴巴 Function Compute 增加了对 Go 语言的支持. HTTP 类型的触发器是常见的触发类型, 并且应用中的函数都可以被各自平台上的多种云服务触发, 其中, 亚马逊 Lambda 支持最多种类的云服务触发. 开发者可以使用各个平台的图形化控制台、命令行工具、API 或软件开发工具包进行应用开发. 另外, 微软 Azure Functions 还可以利用 Visual Studio Code 和 Visual Studio 工具提高开发效率. 4 个平台都支持基于函数服务的工作流构建过程, 但每个平台的工作流构建过程都需要使用特定的定义方式来完成函数编排.

3.2 部署特点

对于应用部署特点, 我们在表 4 中总结部署包大小、内存分配大小、CPU 分配、GPU 支持性、超时时间、实例磁盘大小和支持的区域数量.

表 4 不同服务器无感知平台的部署特点

特点	亚马逊Lambda	谷歌Cloud Functions	微软Azure Functions	阿里巴巴Function Compute
部署包大小	50 MB (已压缩) 250 MB (未压缩)	100 MB (已压缩) 500 MB (未压缩)	1 GB (已压缩)	500 MB (已压缩)
内存分配大小	128–10 240 MB, 以每1 MB为增量	128 MB、256 MB、512 MB、1 024 MB、2 048 MB、4 096 MB、8 192 MB	不需要分配, 内存消耗范围128–1 536 MB之间	128–32 768 MB, 以64 MB的倍数进行递增
CPU分配	根据配置的内存量按比例自动分配	根据配置的内存量按比例自动或手动分配	未知	根据配置的内存量按比例自动或手动分配
GPU支持性	不支持	不支持	不支持	支持
超时时间 (s)	900	540	600	86 400
实例磁盘大小	512–10 240 MB之间	未知	未知	512 MB或10 GB
支持区域数量 (个)	22	29	11	20

在应用部署包大小方面, 亚马逊 Lambda 允许部署压缩包大小在 50 MB 以内的应用, 或者未压缩包大小为 250 MB 以内的应用. 相比之下, 谷歌 Cloud Functions 允许更大的部署包大小, 支持 100 MB 以内的压缩包应用和 500 MB 以内的未压缩包应用. 对于微软 Azure Functions, 先前工作[20,50]提到该平台没有部署包大小限制. 然而, 根据现在的调研结果, 该平台上明确提到开发者需要使用.zip 格式的压缩包文件进行部署, 且支持的最大大小为 1 GB. 可能的原因是, 限制部署包的大小可以提供一种可靠的方式, 使平台能够确定资源分配策略和自动伸缩能力, 以满足需求的变化. 对于阿里巴巴 Function Compute, 先前工作[20]提到该平台最高支持 100 MB 的压缩部署包大小. 然而, 目前阿里巴巴 Function Compute 允许使用压缩为.zip 或.jar 格式的部署包, 大小上限为 500 MB. 总结来说, 对部署包大小进行限制可能帮助服务器无感知平台更好地管理资源、分配计算资源和维持可靠性能, 甚至可以帮助平台预测和处理不同应用的资源需求. 另外, 不同平台制定部署包大小的约束可能也是为了促进平台的自动伸缩策略. 然而, 限制部署包大小可能要求开发者在部署时严格管理依赖项. 如果依赖项太大, 可能会导致部署包大小超出允许的范围. 目前, 各个平台在保证维持正常功能的同时尽量提高可部署的代码大小, 让更多的任务类型能够成功部署并执行在服务器无感知平台上.

关于应用可使用的内存大小, 亚马逊 Lambda 允许配置 128–10 240 MB 之间的数字, 并且是以 1 MB 粒度进行递增和修改. 然而, 在对运行时性能进行探究的实验中, 我们发现实际内存配置大小不能超过 3 008 MB, 这说明官方文档与实际开发可能存在不一致, 令开发者感到困惑. 谷歌 Cloud Functions 允许开发者配置固定大小的内存, 包括 128 MB、256 MB、512 MB、1 024 MB、2 048 MB、4 096 MB 和 8 192 MB. 然而, 之前工作报道该平台允许的内存配置大小不包括 8 192 MB. 阿里巴巴 Function Compute 允许开发者配置 128 MB 到 32 GB 的内存大小, 但递增的粒度是以 64 的倍数进行. 然而, 我们先前的工作总结阿里巴巴函数计算的配置上限是 3 072 MB. 提高可分配的内存大小可以帮助开发者完成更多的任务类型, 同时加快任务执行速度. 与这 3 个平台不同, 微软 Azure Functions 根据实际内存消耗来执行函数, 不支持开发者自定义配置内存大小, 但运行在该平台的函数最多能消耗 1 536 MB. 不同平台对内存大小的限制和配置方式反映了它们在资源分配和控制方面的策略, 但这些都是为了更好地管理资源以满足不同应用的需求. 谷歌 Cloud Functions 选择使用特定的内存大小选项简化了配置选择的复杂性. 亚马逊 Lambda 或阿里巴巴 Function Compute 提供了递增粒度, 如 1 MB 或 64 MB 的倍数, 这也给开发者在内存大小选择方面提供了灵活性. 微软 Azure Functions 采用根据实际内存消耗来执行函数策略, 不支持手动配置内存大小, 这可能也对需求更多内存控制的开发者构成限制.

在 CPU 分配方面, 亚马逊的 Lambda、谷歌 Cloud Functions 和阿里巴巴 Function Compute 根据配置的内存量

按比例分配 CPU 处理能力. 实际的 CPU 分配量可能在不同的函数调用中略有不同, 这可能是因为平台采用了一种灵活的资源分配方式, 根据函数的需求动态分配资源. 但大多数平台都可以根据分配的内存大小自动确定一定 CPU 分配量. 具体来说, 亚马逊 Lambda 在内存大小约为 1 769 MB 时得到一个 vCPU 的处理能力. 谷歌 Cloud Functions 在内存大小约为 2 048 MB 时得到一个 vCPU. 阿里巴巴 Function Compute 在内存大小约为 1 024 MB 时得到一个 vCPU. 注意, vCPU 表示虚拟机实例可用的 CPU 计算资源的一种抽象. 不同平台根据内存分配得到的 vCPU 能力不同的原因可能是各个平台采用不同的资源分配策略和性能标准. 具体来说, 不同的平台使用不同的硬件架构和虚拟化技术, 这些性能特性可能会影响内存和 CPU 分配的方式. 因此, 每个平台可能会根据其硬件和虚拟化技术来确定资源分配策略. 另外, 每个平台通常需要进行大量的性能测试和优化, 以确定在特定内存配置下分配的 vCPU 大小可以提供最佳性能, 这些测试结果可能也会影响平台的资源分配策略. 不同于先前工作的描述, 谷歌 Cloud Functions 和阿里巴巴 Function Compute 已经允许开发者手动修改 CPU 大小以适应任务独特的需求, 这提高了配置的灵活性. 微软 Azure Functions 的 CPU 分配策略尚不清楚. CPU 分配策略反映了不同平台的资源管理和性能调优策略. 开发者需要了解 CPU 配置选项, 并根据需求选择 CPU 大小.

对于 GPU 支持情况, 我们先前的工作报道 4 个平台都暂不支持 GPU. 在我们调研期间, 我们观察到亚马逊 Lambda、谷歌 Cloud Functions 和微软 Azure Functions 目前仍然不支持为函数启动 GPU 实例, 而阿里巴巴 Function Compute 目前允许开发者部署 GPU 类型的函数, 该平台默认通过按量 GPU 实例来提供实时应用场景的执行环境. 具体来说, 阿里巴巴 Function Compute 提供了一种灵活的方式来利用 GPU 计算资源, 开发者只需根据实际需求选择合适的 GPU 型号和计算资源规模, 即可随时启动和停止 GPU 计算, 无需事先规划资源使用情况. 该平台提供 Ampere 以及 Turing 架构的 GPU 实例, 包括 Tesla 系列 T4 卡型和 Ampere 系列 A10 卡型的实例规格[51]. 开发者可以根据业务需求选择不同配置的 GPU 实例. 而且, 阿里巴巴 Function Compute 上的 GPU 实例的使用仅支持通过容器镜像方式部署的函数. 这样一个变化预示着未来的服务器无感知计算将逐渐支持 GPU 资源以用于函数执行, 带来更强大的计算能力. GPU 对于某些工作负载 (如机器学习、图像处理、科学计算等) 可以提供显著的性能优势. 允许函数使用 GPU 可以增强其计算能力, 使其能够更高效地处理复杂任务. 但是 GPU 资源通常比普通的 CPU 资源更昂贵, 开发者需要权衡性能需求和成本, 以确保不会产生不必要的费用. 而且, 不是所有应用都能够受益于 GPU 加速, 一些任务可能更适合在 CPU 上执行.

对于执行的超时时间限制, 与先前工作相同的是, 亚马逊 Lambda 允许执行时间不超过 900 s (15 min) 的任务. 谷歌 Cloud Functions 和微软 Azure Functions 执行事件驱动型函数的超时时长上限分别是为 540 s (9 min) 和 600 s (10 min). 然而, 我们调研了解到目前阿里巴巴 Function Compute 的函数执行超时时间从 900 s (15 min) 更新为默认为 60 s, 但最长可达 86 400 s. 限制函数的执行时间本质上有助于平台管理资源, 避免因执行时间过长而占用大量的计算资源, 导致并发和资源竞争问题, 从而影响其他函数的执行. 然而, 增加函数超时时间也为长时任务的执行提供了机会, 例如批处理、数据分析或大规模计算任务, 并使平台更适合这些长时间运行的应用场景.

关于平台提供的实例磁盘大小, 不同平台对本地存储空间大小有不同的限制. 亚马逊 Lambda 允许开发者配置 512 MB 与 10 240 MB 之间的值, 以 1 MB 为递增粒度. 比较先前工作[20]的总结, 亚马逊 Lambda 目前支持了比原来固定 512 MB 大小更大的存储空间. 这样可以提高函数在执行期间的本地数据存储能力. 然而, 谷歌 Cloud Functions 和微软 Azure Functions 提供的实例磁盘大小尚不清楚. 对于阿里巴巴 Function Compute, 先前工作提到该平台只支持 512 MB 磁盘大小, 但目前该平台对于函数实例磁盘大小有两种选择: 一种是默认的 512 MB, 不计费; 另一种是可选择的 10 GB, 根据业务情况进行选择, 是计费的. 随着服务器无感知计算编程模式的发展, 越来越多的任务选择在这些平台上开发和执行. 这些任务可能涉及更复杂的数据处理、临时存储需求或大规模计算, 因此, 需要更多的磁盘容量来满足其需求. 另外, 亚马逊 Lambda 和阿里巴巴 Function Compute 增加了磁盘大小的配置选项, 以提供更大的灵活性. 然而, 即使提供更大的磁盘容量, 仍然会受到物理存储资源的限制. 因此, 如果需要大量本地存储, 可能需要考虑其他存储解决方案 (如云存储).

对于支持的区域, 亚马逊 Lambda 的函数可以在 22 个特定的区域进行部署, 包括美国、欧洲、亚太地区、加拿大等. 谷歌 Cloud Functions 允许在 29 个区域部署函数, 包括欧洲、亚太地区、美国和加拿大等. 然而, 微软

Azure Functions 仅支持 11 个区域, 主要集中在加拿大和美国. 阿里巴巴 Function Compute 支持 20 个区域, 主要在中国, 也有少量服务区域在美国和欧洲等地. 相比较先前工作总结的区域个数, 亚马逊 Lambda、谷歌 Cloud Functions 和阿里巴巴 Function Compute 所支持的区域个数相差不大, 而微软 Azure Function 所支持的区域个数变少. 然而, 先前工作却没有具体描述区域分布, 我们补充了这一部分内容. 总结来说, 不同平台支持的区域数量和分布反映了他们的市场战略和全球覆盖的目标. 服务器无感知平台将相应的服务节点部署在特定的地理位置以满足它们数据主权和法规要求. 然而, 支持更多区域也意味着平台需要投入更多的资源用于维护, 这可能增加了运营成本, 尤其是在全球范围内提供高质量的服务.

发现: 4 个平台在应用部署包大小和任务执行时间上都有不同的限制. 考虑到部署失败或执行失败的可能性, 开发者不能忽略这些限制. 除了微软 Azure Functions 采用基于内存消耗的方式执行服务器无感知函数, 其他平台都是根据预先指定的内存大小来执行函数. 由于不同任务对内存和 CPU 资源的依赖程度不同, 谷歌 Cloud Functions 和阿里巴巴 Function Compute 支持单独配置内存和 CPU 资源的大小. 目前大多数平台还不支持 GPU 实例. 研究的 4 个平台都支持在多个服务区域进行部署.

3.3 运行时特点

对于应用运行时特点, 在表 5 中总结调用类型、负载大小、输入输出结果存储、运行时系统和付费模型.

表 5 不同服务器无感知平台的运行时特点

特点	亚马逊Lambda	谷歌Cloud Functions	微软Azure Functions	阿里巴巴Function Compute
调用类型	同步/异步	同步/异步	同步/异步	同步/异步
负载大小	6 MB (同步)/256 KB (异步)	10 MB	100 MB	32 MB (同步)/128 KB (异步)
输入输出结果存储	S3云存储	Storage云存储	Blob存储	对象存储OSS
运行时系统	Linux, Linux 2	Ubuntu 18.04, Ubuntu 22.04	Linux, Windows	Debian 9 (Stretch), Debian 10 (Buster)
付费模型	按照分配内存的使用量	按照分配内存的使用量	按照消耗内存的使用量	按照分配内存的使用量

在调用类型方面, 本文所调研的平台主要支持两种调用: 同步和异步. 在同步调用时, 平台会运行该函数并等待响应. 当函数完成时, 平台返回来自函数的响应和相关数据. 而在异步调用时, 无需等待函数的响应, 平台会自动处理剩余代码部分. 本文所调研的 4 个平台都支持同步和异步两种调用类型. 同步调用需要等待函数执行完成, 可能会占用服务器资源并延长时间. 异步调用可以提高性能和资源的利用效率. 但是异步调用可能需要额外的错误处理机制, 以确保任务的正确执行和结果的可靠性.

对于负载方面, 不同平台对函数请求的大小也有所不同. 在亚马逊 Lambda 上, 开发者在请求函数时, 同步调用下的负载大小限制为 6 MB, 异步调用下为 256 KB. 而在阿里巴巴 Function Compute 上, 与先前工作所总结的特征不同的是, 其同步和异步请求下的负载大小分别 32 MB 和 128 KB. 先前工作报道该平台同步调用的负载大小支持最大为 6 MB. 然而, 目前这一限制提高到 32 MB. 这意味着阿里巴巴 Function Compute 可以用于处理大规模数据和复杂任务, 为同步任务的执行提供了更大的灵活性. 至于谷歌 Cloud Functions, 它的请求大小的上限为 10 MB. 而微软 Azure Functions 的函数请求大小最大可达 100 MB. 负载大小限制的增加可以满足处理不同规模数据的任务. 然而, 需要注意的是, 处理更大的请求负载可能需要更多的资源, 包括内存和计算资源. 开发者需要明确平台是否能够提供足够的资源来处理这些请求. 而且, 更大的请求负载可能需要更长时间的传输, 尤其在慢速或不稳定的网络连接下, 这可能影响应用的性能和响应时间.

在输入输出结果存储方面, 可以选择相应平台上的云存储服务. 亚马逊 Lambda 可使用 S3 云存储, 谷歌 Cloud Functions 可使用 Storage 云存储, 微软 Azure Functions 可使用 Blob 存储, 阿里巴巴函数计算可使用对象存储 OSS. 总体来说, 每个平台都可以选择特定的存储服务来保存输入输出数据. 这些云存储服务可以确保数据的持久性, 即使函数执行结束后, 数据仍然可用, 这有助于数据的长期保存和后续分析. 同时, 这些云存储服务具备一定的可扩展性, 这对于处理大规模的数据非常重要. 另外, 这些云存储在一定的权限设置下可以让多个函数或应用访问特定数

据, 从而实现数据共享和协作. 在运行时系统方面, 亚马逊 Lambda 使用了 Linux 以及 Linux 2 两种系统. 具体来说, Linux 系统是亚马逊 Lambda 的早期版本所使用的运行时环境. 当触发函数执行时, 亚马逊 Lambda 会创建一个 Linux 容器实例来运行该函数. 随着时间的推移, 亚马逊引入了 Linux 2 作为亚马逊 Lambda 的运行时环境. Linux 2 是亚马逊专门开发的 Linux 发行版, 它构建在自定义内核上[52], 并提供了更好的性能、安全性和稳定性. 对于被定义为容器镜像的函数, 可以在创建容器镜像时选择运行时和 Linux 发行版. 谷歌 Cloud Functions 则使用两种了 Ubuntu 系统: Ubuntu 18.04 和 Ubuntu 22.04. 相较于先前工作, 该平台增加了 Ubuntu 22.04 系统的使用, 为函数提供更先进的执行环境. 而微软 Azure Functions 对不同的支持语言提供了多个运行时系统选项. 例如, JavaScript 可以运行在 Linux 上, 也可以运行在 Windows, 但 Python 只能运行在 Linux 上. 至于阿里巴巴 Function Compute, 它的函数运行时基于特定的 Linux 发行版本制作, 先前工作报道该平台只支持 Debian 9 (Stretch). 然而, 目前该平台支持 Debian 9 (Stretch) 和 Debian 10 (Buster) 两种发行版本, 且运行时可以支持单个版本或多个版本的同一种语言, 也可以支持多种语言. 版本的使用寿命结束时, 指定语言或框架版本的运行时也将终止支持. 不同的运行时系统可以支持不同的编程语言, 提供多个运行时系统选项可以满足不同开发者和应用需求, 并提供更丰富的生态系统支持. 而且, 不同的运行时系统可能具有不同的性能特性和优化, 选择合适的运行时系统可以提高应用的性能和可扩展性. 我们也观察到平台对运行时系统的更新, 这可以帮助平台跟上新的技术和发展趋势, 提供更先进的运行时环境. 然而, 不同的运行时系统可能支持不同的库和依赖项, 开发者需要确保其应用程序的依赖项在所选系统下可用.

在付费模型方面, 这些平台都提供了细粒度的付费模式. 然而, 各个平台的计费方式有所不同, 具体可分为两种: 基于分配的内存使用量和基于消耗的内存使用量. 亚马逊 Lambda、谷歌 Cloud Functions 和阿里巴巴 Function Compute 都是基于函数执行时间和实际分配的内存大小来计算成本, 且执行时间以更细粒度的 1 ms 为单位计算. 概括来讲, 这些平台的计费方式主要由函数调用次数以及相应的资源使用量计费. 资源使用量一般包括函数执行使用的内存资源量和 CPU 资源量. 在默认情况下, 这些平台的 CPU 大小随分配的内存大小自动分配和确定. 因此, 可以依据函数执行时间进一步计算出所使用的内存资源量和 CPU 资源量. 因此, 默认情况下, 这些平台被认定为是基于分配的内存使用量来确定成本. 由第 3.2 节的 CPU 分配可知, 谷歌 Cloud Functions 和阿里巴巴 Function Compute 允许开发者手动修改 CPU 资源大小. 如果开发者在这两个平台上手动修改 CPU 大小, 成本中的资源使用量将分别按照各自配置的内存大小和 CPU 大小计算相应的资源使用量. 然而, 微软 Azure Functions 的付费模型是根据消耗量计费, 即函数在运行时消耗的内存大小和执行时间, 其中消耗内存范围从 128 MB 到 1 536 MB 不等, 并且执行时间也以 1 ms 为单位计算. 这 4 个平台都提供了细粒度的计费模型, 允许开发者按照实际资源确定成本, 从而提供更公平的计费方式. 对于使用基于消耗资源使用量的计费可能需要更多的监控和记录, 以确保准确的计费, 但这可能增加了监控和记录的复杂性.

发现: 调研的 4 个平台都支持同步和异步函数调用, 并且对请求的负载大小有不同的限制. 当函数的输入输出数据需要存储时, 开发者可以利用各个平台提供的云存储服务. 它们大多数支持的运行时系统都是在 Linux 下的具体版本, 但微软 Azure Functions 对特定语言还支持 Windows 上运行. 另外, 亚马逊 Lambda、谷歌 Cloud Functions 和阿里巴巴 Function Compute 使用的付费模型是基于分配的内存使用量, 而微软 Azure Functions 则是基于消耗的内存使用量进行计费.

4 运行时性能分析结果

本节基于不同基准测试程序集, 对亚马逊 Lambda、谷歌 Cloud Functions、微软 Azure Functions 和阿里巴巴 Function Compute 的冷启动性能和执行性能进行了探究.

4.1 冷启动性能

冷启动性能是服务器无感知计算面临的关键挑战. 为了帮助开发者更好地理解不同服务器无感知平台产生的冷启动性能, 本节使用微基准测试程序集中的 Hello World 应用, 该应用实现为不同编程语言版本, 以比较不同平台之间的冷启动延迟. 具体而言, 我们研究了编程语言类型和内存分配大小对冷启动延迟的影响.

在编程语言方面, 我们主要探究了 Python、JavaScript 和 Java 这 3 种广泛应用于服务器无感知计算的编程语言[14,15]. 考虑到本文中特征总结的结果, 除了微软 Azure Functions 采用动态内存分配的策略外, 其他 3 个平台都需要开发者提前分配指定的内存大小. 为了统一这 3 个平台的内存分配大小, 本文选择它们都支持的内存大小. 我们了解到, 谷歌 Cloud Functions 支持固定的内存分配大小, 包括 128 MB、256 MB、512 MB、1 024 MB、2 048 MB、4 096 MB、8 192 MB. 亚马逊 Lambda 允许配置 128–10 240 MB 大小之间的数字. 阿里巴巴 Function Compute 则支持从 128 MB 到 32 GB 的内存大小. 然而, 在实验中, 我们发现在亚马逊 Lambda 上部署函数时, 实际上只能设置小于等于 3 008 MB 的内存, 这表明亚马逊 Lambda 官方文档的说明和应用实践可能存在不一致的情况. 因此, 综合谷歌 Cloud Functions 和阿里巴巴 Function Compute 的内存分配大小, 我们探究了分别在 128 MB、256 MB、512 MB、1 024 MB 和 2 048 MB 下不同编程语言编写的函数即服务应用的冷启动延迟. 为了直观比较结果, 我们总结了每组实验中 1 000 次度量结果的中位数和上下四分位数范围, 展示在表 6 中. 此外, 具体性能数据分布将展示在图 3–图 9 中. 接下来, 我们将分析平台内和平台间冷启动延迟结果.

表 6　不同平台在不同内存分配下执行不同编程语言应用的中位数冷启动延迟和上下四分位数范围 (ms)

编程语言	内存大小 (MB)	亚马逊 Lambda	谷歌 Cloud Functions	微软 Azure Functions	阿里巴巴 Function Compute
Python	128	373.32 ([356.93, 397.30])	3 426.77 ([2 943.20, 4 327.96])		2 717.00 ([2 622.52, 2 830.69])
	256	380.33 ([359.14, 410.69])	2 716.03 ([2 433.82, 3 102.91])		2 278.95 ([2 212.39, 2 356.42])
	512	377.87 ([358.34, 409.70])	2 695.39 ([2 393.64, 3 062.36])	2 824.34 ([2 344.57, 3 443.72])	2 251.13 ([1 133.24, 2 326.44])
	1 024	373.78 ([355.71, 408.57])	2 687.09 ([2 374.02, 3 086.26])		2 217.56 ([1 120.98, 2 311.82])
	2 048	379.51 ([357.44, 413.72])	2 678.67 ([2 407.61, 3 153.06])		1 217.99 ([1 119.33, 2 296.15])
JavaScript	128	430.52 ([410.46, 459.46])	2 596.22 ([2 275.23, 3 036.19])		1 165.22 ([1 128.32, 2 250.42])
	256	432.01 ([408.37, 459.66])	2 593.08 ([2 249.28, 2 972.23])		1 161.28 ([1 129.14, 1 542.85])
	512	430.60 ([411.44, 454.93])	2 581.95 ([2 298.93, 2 978.61])	2 385.69 ([1 943.22, 2 947.28])	1 153.54 ([1 128.48, 1 351.06])
	1 024	426.96 ([408.01, 450.22])	2 622.05 ([2 318.50, 3 021.09])		1 147.31 ([1 121.41, 1 236.02])
	2 048	424.68 ([405.80, 450.25])	2 733.99 ([2 388.54, 3 037.83])		1 161.13 ([1 137.64, 1 233.54])
Java	128	783.91 ([754.24, 813.84])	2 940.94 ([2 615.94, 3 336.05])		1 342.00 ([1 316.01, 1 400.25])
	256	747.09 ([721.93, 779.51])	2 790.70 ([2 486.41, 3 236.96])		1 346.92 ([1 400.25, 1 404.60])
	512	741.41 ([718.15, 770.26])	2 866.50 ([2 571.00, 3 304.19])	2 142.25 ([1 793.71, 2 777.07])	1 348.16 ([1 319.54, 1 405.88])
	1 024	736.08 ([711.02, 766.32])	2 839.60 ([2 532.14, 3 256.73])		1 354.29 ([1 324.68, 1 407.18])
	2 048	685.07 ([662.91, 718.81])	2 892.08 ([2 589.00, 3 326.58])		1 362.77 ([1 335.97, 1 416.47])

(1) 平台内冷启动延迟比较: 针对每个服务器无感知平台, 我们比较了不同编程语言编写的应用在各个内存分配大小下的冷启动延迟. 对于亚马逊 Lambda 的冷启动性能结果, 从图 3 中可以观察到, 在该平台上执行 Python 应用和 JavaScript 应用的冷启动延迟要低于 Java 应用. 这些结论在为这些应用分别分配 128 MB、256 MB、512 MB、1 024 MB 和 2 048 MB 内存时同样成立. 根据表 6 中亚马逊 Lambda 结果的中位数数据, 以 128 MB 内存大小为例, Python 应用的冷启动延迟为 373.32 ms, JavaScript 应用的冷启动延迟为 430.52 ms, 而 Java 应用的冷启动延迟

为 783.91 ms. 这表明在亚马逊 Lambda 上执行 Python 应用产生的冷启动延迟最低, JavaScript 应用的冷启动延迟相对低, 而 Java 应用的冷启动延迟最高, 大约是 Python 应用冷启动延迟的两倍. 这样的结果可能是由于 Python 和 JavaScript 是轻量级编程语言, 而 Java 是一种重量级编程语言. 此外, 对于这些编程语言的应用, 内存分配大小对冷启动延迟的影响不明显. 根据图 3 中的冷启动性能数据分布, 可以看出这 3 种编程语言的应用在亚马逊 Lambda 上执行的冷启动延迟相对集中. 根据表 6 中上下四分位数范围的数据, 这些应用在不同内存配置下的冷启动延迟差异大约为 50 ms.

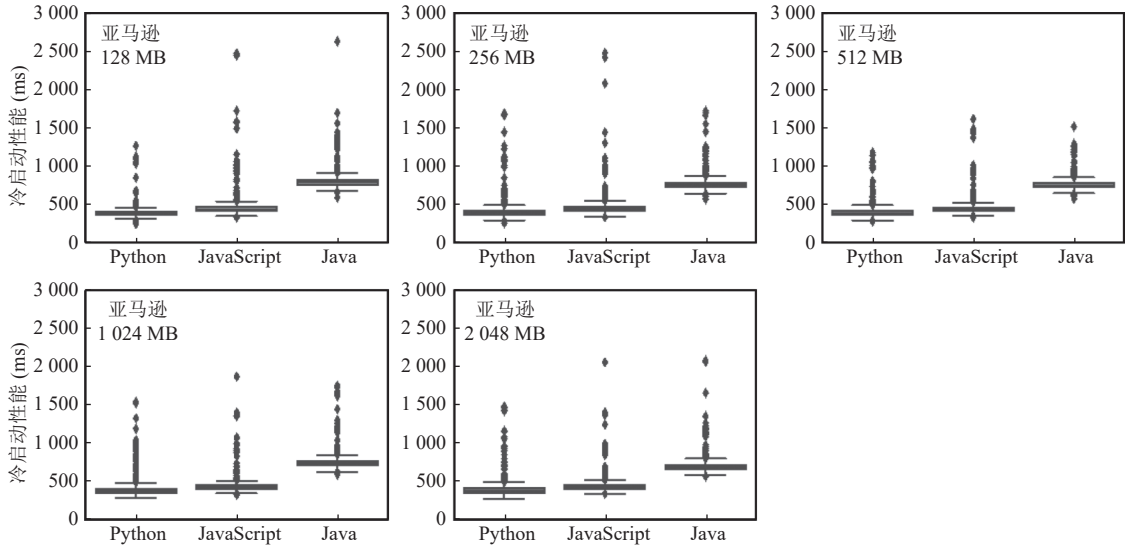

图 3　亚马逊 Lambda 在不同内存分配下执行不同编程语言编写的应用产生的冷启动延迟

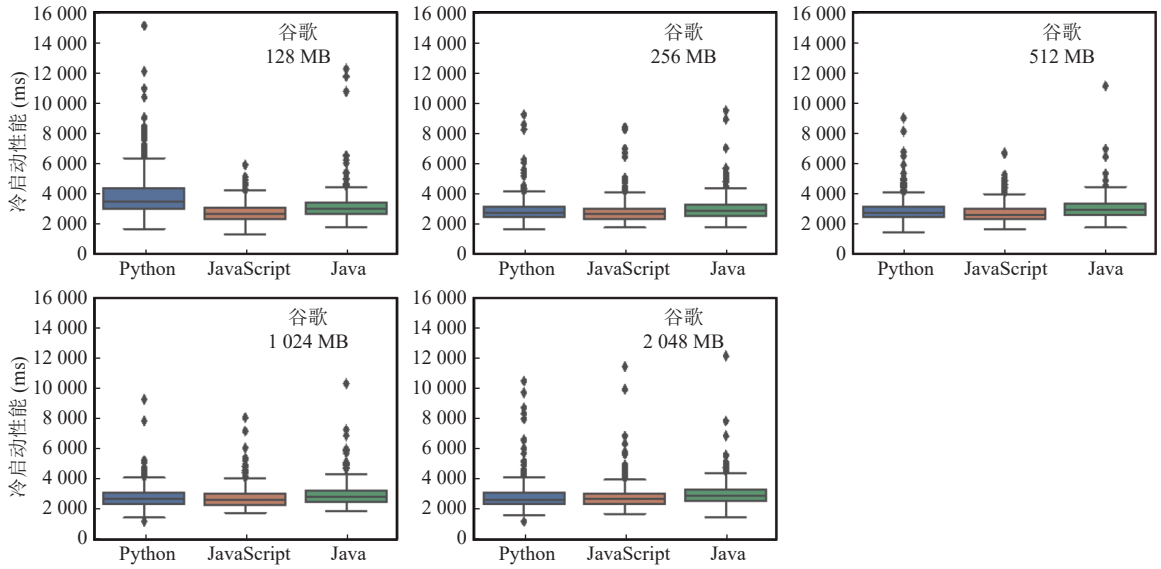

图 4　谷歌 Cloud Functions 在不同内存分配下执行不同编程语言编写的应用产生的冷启动延迟

对于谷歌 Cloud Functions 的冷启动性能, 从图 4 的整体趋势来看, 不同编程语言的应用在谷歌 Cloud Functions 上的冷启动延迟没有显著差异. 然而, 当内存设置为 128 MB 时, 产生的冷启动性能分布相对不稳定. 从图 4 中 128 MB 的结果可以看出, Python 应用的冷启动延迟结果分布广泛. 表 6 中展示出其上下四分数范围从

2 943.20 ms 到 4 327.96 ms, 相差将近 1 400 ms, 因此不建议将应用配置在这个内存大小上. 对于其他内存配置, 这些应用在不同内存下的冷启动延迟差异达到了 700 ms 左右, 这可从图 4 中的上下四分位数范围来观察到. 进一步观察表 6 中的谷歌 Cloud Functions 的中位数结果, 发现不同语言的应用的冷启动延迟在 2.5–2.8 s 左右. 举例来说, 当内存大小设置为 1 024 MB 时, Python 应用、JavaScript 应用和 Java 应用的冷启动延迟分别是 2 687.67 ms、2 622.05ms 和 2 839.60 ms. 当内存设置为 2 048 MB 时, 这 3 种应用的冷启动延迟分别是 2 678.67 ms、2 733.99 ms 和 2 892.08 ms. 总体而言, 谷歌 Cloud Functions 上执行的 Python 应用和 JavaScript 应用比 Java 应用具有更低的冷启动延迟, 减少了大约 100 ms. 此外, 对于这些语言的应用, 我们观察到内存分配的大小不会对冷启动延迟产生明显的提高或降低.

对于微软 Azure Functions 的冷启动性能, 该平台采用基于内存消耗的策略执行应用, 开发者无需提前指定内存大小. 因此, 我们直接展示不同编程语言应用的冷启动延迟结果和分布. 图 5 结果显示 Python 应用的箱线图整体略高于 JavaScript 和 Java 应用, 这与亚马逊 Lambda 和谷歌 Cloud Functions 的结果不同. 在微软 Azure Functions 上执行的 Python 应用、JavaScript 应用和 Java 应用的性能数据的上下四分位数范围分别为 2 344.57–3 443.72 ms、1 943.22–2 947.28 ms 和 1 793.71–2 777.07 ms, 这显示微软 Azure Functions 的性能差异达到了约 1 000 ms. 从表 6 中微软 Azure Functions 的中位数结果, 发现 Python 应用的冷启动延迟中位数为 2 824.34 ms, JavaScript 应用的冷启动延迟为 2 385.69 ms, 而 Java 应用的冷启动延迟为 2 142.25 ms, 这也说明在微软 Azure Functions 上执行 Java 应用产生的冷启动延迟最小, 而 Python 应用的冷启动延迟最大. 造成这一结果的原因可能是微软在 Java 应用的运行时环境初始化方面更为高效, 而 Python 应用的运行时环境可能相对较新, 在启动时需要进行更多的加载和配置工作, 因此产生了较高的冷启动性能开销.

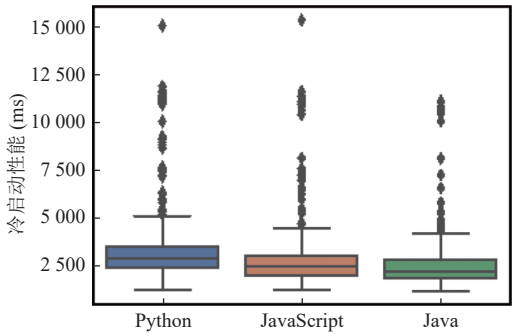

图 5 微软 Azure Functions 在不同内存分配下执行不同编程语言编写的应用产生的冷启动延迟

对于阿里巴巴 Function Compute 的冷启动性能, 从图 6 中性能结果分布可以看出, 在高内存分配 (大于 512 MB) 下, 阿里巴巴 Function Compute 执行 Python 应用的性能分布较不稳定, 而在低内存分配下, 冷启动性能结果更为稳定. 这可能是因为阿里巴巴 Function Compute 在执行 Python 应用时, 较高的内存分配大小可以明显降低冷启动延迟, 从而增强性能结果的稳定性. 对于 JavaScript 应用和 Java 应用, 内存分配的大小对冷启动延迟没有明显的影响. 具体而言, JavaScript 应用在不同内存下的中位数性能范围为 1 147.31–1 165.22 ms, 而 Java 应用在不同内存下中位数性能范围为 1 342.00–1 362.77 ms. 对于 JavaScript 应用, 在低内存分配下 (小于 512 MB) 性能较不稳定, 在高内存分配下性能较稳定. 尽管 JavaScript 应用的中位数结果没有显著变化, 但在低内存下执行 JavaScript 应用会产生较大的性能差异, 例如在 128 MB 下可以达到约 1 000 ms 的延迟. 对于 Java 应用, 在我们所度量的所有内存分配大小下都展示出较稳定的性能结果, 尽管也存在一些异常值. 从表 6 中的阿里巴巴 Function Compute 的中位数结果看, 执行 Python 应用产生的冷启动延迟整体高于 JavaScript 应用和 Java 应用. 例如, 在内存大小为 1 024 MB 时, Python 应用的冷启动延迟为 2 217.56 ms, 而 JavaScript 应用的冷启动延迟为 1 147.31 ms, Java 应用的冷启动延迟为 1 354.29 ms. 这说明阿里巴巴 Function Compute 执行 JavaScript 应用所产生的冷启动延迟最低, 而执行 Python 应用的冷启动延迟最高.

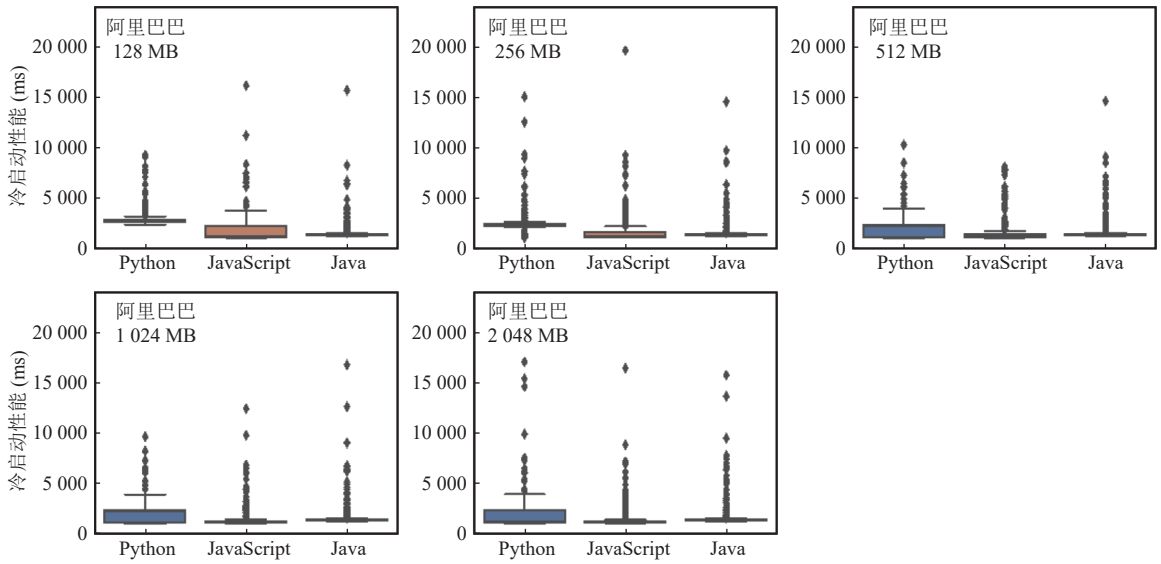

图 6 阿里巴巴 Function Compute 在不同内存分配下执行不同编程语言编写的应用产生的冷启动延迟

发现: 每个服务器无感知平台在执行不同编程语言的应用时, 冷启动性能表现不一致. 亚马逊 Lambda 和谷歌 Cloud Functions 在执行 Python 和 JavaScript 应用时的冷启动延迟低于执行 Java 应用时的冷启动延迟, 而且这两个平台的内存分配大小对冷启动延迟没有明显影响. 与此不同, 微软 Azure Functions 执行 Java 应用时会产生更低的冷启动延迟, 而阿里巴巴 Function Compute 在执行 JavaScript 应用时产生更低的冷启动延迟, 执行 Python 应用时产生最高的冷启动延迟, 且该延迟受内存分配大小的影响.

(2) 平台间冷启动延迟比较: 为了直观地比较不同平台对于特定编程语言应用的冷启动延迟大小, 本文进行跨平台的冷启动延迟比较. 探究了每种编程语言在 4 个平台上在特定内存下的冷启动延迟. 考虑到微软 Azure Functions 不需要提前进行内存分配, 因此在进行特定内存大的比较时, 我们使用对应编程语言应用在微软 Azure Functions 上产生的冷启动延迟与其他平台进行比较.

对于 Python 应用, 根据图 7 的结果, 亚马逊 Lambda 在所有测试的内存大小下都表现出比其他平台更低的冷启动延迟. 具体来说, 在表 6 中, Python 应用在内存 128 MB 下的中位数性能为 373.32 ms, 在内存 256 MB 下为 380.33 ms, 在内存 512 MB 下为 377.87 ms, 在 1 024 MB 下为 373.78 ms, 以及在内存 2 048 MB 下为 379.51 ms. 进一步观察图 7 的性能分布, 可以看出亚马逊 Lambda 所产生的所有性能数据比其他平台更为集中. 对于其他 3 个平台, 可以观察到在不同内存大小下, 阿里巴巴 Function Compute 的性能表现整体上优于谷歌 Cloud Functions 和微软 Azure Functions 的结果. 因此, 如果开发者希望为 Python 应用提供较低的冷启动延迟和稳定的性能分布, 他们可以优先考虑使用亚马逊 Lambda 和阿里巴巴 Function Compute 这两个平台.

对于 JavaScript 应用, 根据图 8 的结果可以得出结论, 亚马逊 Lambda 和阿里巴巴 Function Compute 在所测试的内存大小下, 产生的冷启动延迟都比谷歌 Cloud Functions 和微软 Azure Functions 要低, 而且它们产生的延迟分布也相对集中. 具体来说, 表 6 展示亚马逊 Lambda 在不同内存下执行 JavaScript 应用产生的性能中位数介于 424.96–432.01 ms 之间, 而阿里巴巴 Function Compute 产生的性能中位数介于 1 147.31–1 165.28 ms 之间. 与此相比, 微软 Azure Functions 的性能结果中位数是 2 385.69 ms, 谷歌 Cloud Functions 的中位数结果在 2 581.95–2 733.99 ms 之间. 这些结果表明亚马逊 Lambda 的冷启动延迟比阿里巴巴 Function Compute 更小, 而谷歌 Cloud Functions 的冷启动延迟比微软 Azure Functions 更大. 因此, 如果开发者希望为 JavaScript 应用提供较低的冷启动延迟和稳定的性能结果, 他们可以优先选择亚马逊 Lambda 和阿里巴巴 Function Compute 这两个平台.

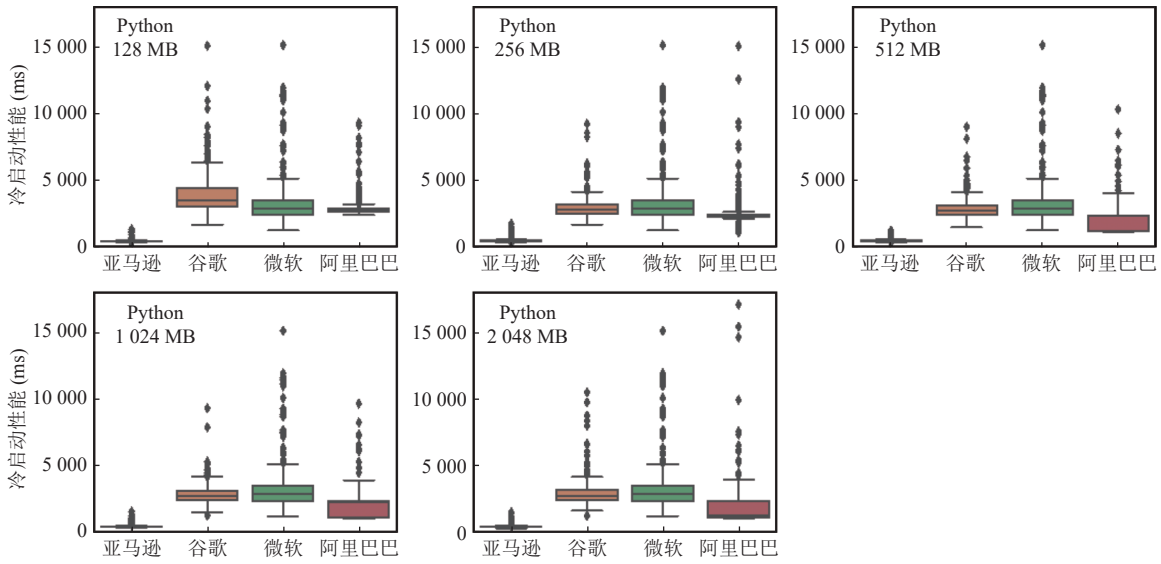

图 7　不同平台在不同内存分配大小下执行 Python 应用的冷启动延迟

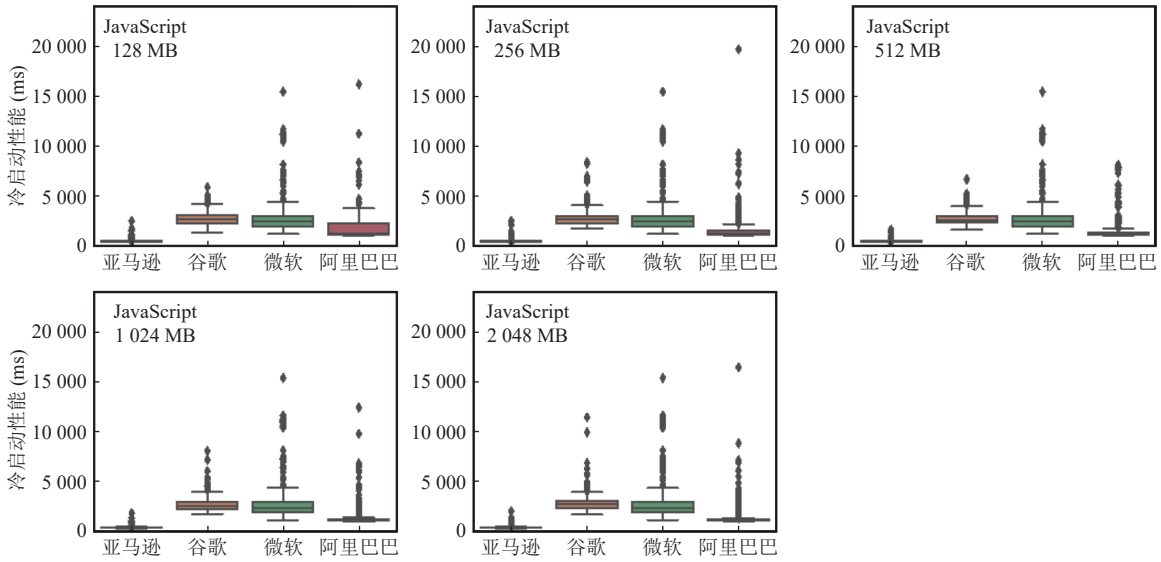

图 8　不同平台在不同内存分配大小下执行 JavaScript 应用的冷启动延迟

对于 Java 应用, 根据图 9 的结果, 可以得出与 Python 应用和 JavaScript 应用相似的结论, 即亚马逊 Lambda 和阿里巴巴 Function Compute 产生的冷启动延迟比谷歌 Cloud Functions 和微软 Azure Functions 更低, 而且它们产生的延迟分布也更为集中. 具体来说, 在表 6 中可见, 亚马逊 Lambda 在不同内存大小下执行 Java 应用产生的性能中位数介于 685.07–783.91 ms 之间, 而阿里巴巴 Function Compute 产生的性能中位数介于 1 342.00–1 362.77 ms 之间. 与此相比, 微软 Azure Functions 执行 Java 应用的冷启动延迟为 2 145.25 ms, 而谷歌 Cloud Functions 在不同内存分配下超过 2 700 ms, 这表明谷歌 Cloud Functions 产生的冷启动开销更大. 因此, 如果开发者希望为 Java 应用提供较低的冷启动延迟和稳定的性能分布, 他们可以优先选择亚马逊 Lambda 和阿里巴巴 Function Compute 这两个平台.

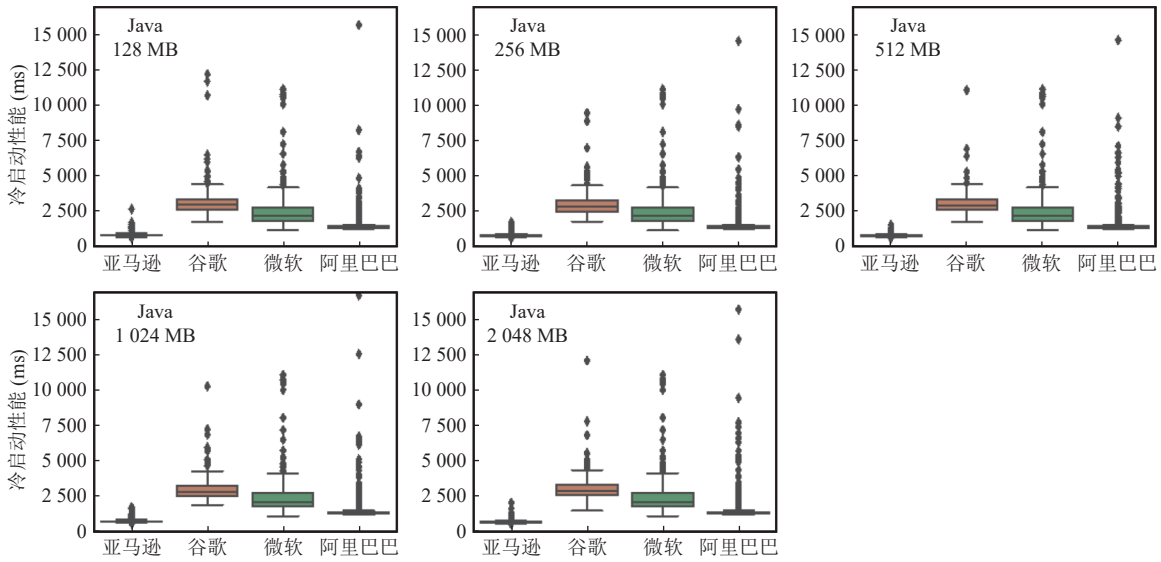

图 9 不同平台在不同内存分配大小下执行 Java 应用的冷启动延迟

发现: 不同服务器无感知平台下的 3 类主流编程语言应用在不同内存下表现出相似的结果. 具体而言, 亚马逊 Lambda 和阿里巴巴 Function Compute 在执行 Python 应用、JavaScript 应用和 Java 应用时, 都表现出比谷歌 Cloud Functions 和微软 Azure Functions 更低的冷启动延迟. 然而, 亚马逊 Lambda 仍然能够获得最低的延迟开销, 大约在 300–800 ms 之间, 而阿里巴巴 Function Compute 的冷启动延迟介于 1 100–2 800 ms 之间. 对于谷歌 Cloud Functions 和微软 Azure Functions, 谷歌 Cloud Functions 产生的冷启动开销更高, 接近 3 000 ms, 而微软 Azure Functions 的冷启动开销也在 2 100–2 900 ms 之间. 这些结果表明, 优化函数即服务应用的冷启动性能是服务器无感知计算领域中的关键挑战.

4.2 执行性能

本节使用了微基准测试程序和宏基准测试程序探究亚马逊 Lambda、谷歌 Cloud Functions、微软 Azure Functions 和阿里巴巴 Function Compute 在执行性能方面的表现. 我们关注的是针对消耗特定资源的任务和消耗多种资源的任务在不同的服务器无感知平台上的执行性能表现. 通过使用微基准测试程序, 我们可以针对特定资源 (如 CPU 和内存等) 进行测试, 并观察每个平台在处理此类任务时的性能差异, 这有助于了解每个平台对于单一资源需求的处理能力, 从而为特定应用场景做出选择. 而宏基准测试程序则更加全面, 涵盖了多种资源的消耗情况. 通过执行此类应用, 我们可以更全面地评估每个平台在处理多种资源需求的任务时的性能表现, 这有助于了解平台的综合性能, 并为多资源消耗型应用场景做出决策. 为了确保各平台的实现方式的一致性, 我们只在对每个平台要触发服务器无感知函数的定义格式和解析事件信息形式进行调整外, 其他的代码 (包括依赖包) 是完全一样的. 每个平台对要触发的函数的编程形式存在细微差异. 例如, 亚马逊 Lambda 函数使用类似于 handler(event, context) 的定义方式, 然后对 event 进行解析, 而谷歌 Cloud Functions 函数使用类似于 handler(request) 的定义方式, 然后对 request 进行解析. 因此, 我们只对用户代码进行轻微调整.

基准测试程序都执行在 4 个平台共同支持的 Python 3.9 版本. 默认情况下, 超时时间默认设置为 5 min (300 s), 内存配置大小默认为 128 MB. 然而, 一些复杂的应用 (如语音识别、机器学习训练和机器学习推理) 在 128 MB 和 256 MB 的内存配置下无法成功调用, 这是因为它们在实际执行过程中需要更多的内存资源. 因此, 将语音识别、机器学习训练和机器学习推理应用的内存大小都提高到 512 MB, 以确保它们被成功运行和公平地比较.

我们也对每个基准测试程序的成本进行计算, 进而开展成本比较. 根据各个平台的计费方式, 计算每个应用的成本, 考虑了其在 1 000 次重复执行和中位数执行性能延迟下的成本.

(1) 微基准测试程序的执行性能: 表 7、图 10 和图 11 展示了微基准测试程序在不同平台上的执行性能结果. 我们将详细分析不同平台在处理 CPU 密集型任务、内存密集型任务、磁盘 IO 密集型任务和网络密集型任务时的执行性能表现.

表 7　微基准测试程序的执行性能结果和成本

任务缩写表示	执行性能	亚马逊 Lambda	谷歌 Cloud Functions	微软 Azure Functions	阿里巴巴 Function Compute
CPU_task	中位数 (ms)	123.91	186.28	70.20	138.97
	成本 (美元)	0.000 46	0.000 78	0.000 34	0.000 49
	上四分位数 (ms)	125.22	234.48	98.35	143.53
	下四分位数 (ms)	122.52	141.10	38.70	136.88
Memory_task	中位数 (ms)	556.36	709.57	116.36	341.98
	成本 (美元)	0.001 36	0.001 86	0.000 43	0.000 91
	上四分位数 (ms)	574.46	762.25	159.30	407.23
	下四分位数 (ms)	538.77	656.31	69.66	302.92
DiskIO_task	中位数 (ms)	4 233.19	4 380.13	469.36	3 349.69
	成本 (美元)	0.009 02	0.009 43	0.001 14	0.007 18
	上四分位数 (ms)	4 280.30	4 660.86	520.47	3 491.91
	下四分位数 (ms)	4 178.49	4 123.68	421.06	3 254.63
Network_task	中位数 (ms)	368.69	312.50	109.57	159.82
	成本 (美元)	0.000 97	0.001 04	0.000 42	0.000 53
	上四分位数 (ms)	379.25	355.30	167.82	172.50
	下四分位数 (ms)	342.50	265.95	71.22	154.94

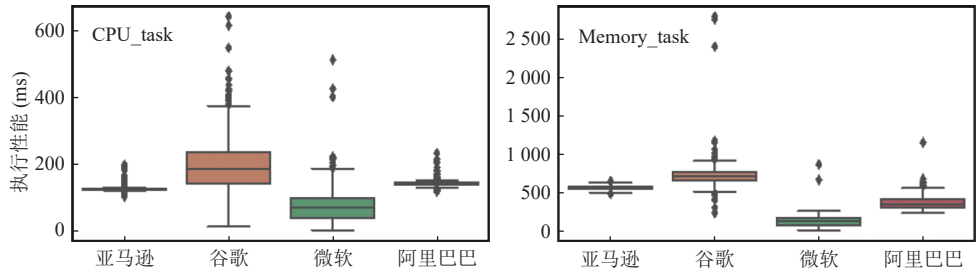

图 10　CPU 密集型任务和内存密集型任务在不同平台上的执行性能分布

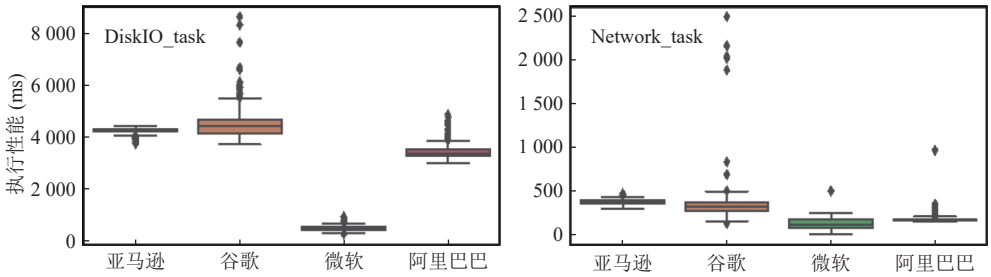

图 11　磁盘 IO 密集型任务和网络密集型任务在不同平台上的执行性能分布

对于 CPU 密集型任务, 根据表 7 中的中位数和四分位数性能结果, 微软 Azure Functions 在执行性能方面优于其他 3 个平台. 它以仅 70.20 ms 的速度执行 CPU 密集型任务, 而且上下四分位数范围为 38.70–98.35 ms. 这种平台的高执行速度可能归因于其底层实例内存大小的动态分配, 根据应用的执行需求进行自动调整. 而且, 微软

Azure Functions 的成本也是成本最低的. 然而, 亚马逊 Lambda、谷歌 Cloud Functions 和阿里巴巴 Function Compute 分别需要 123.91 ms、186.28 ms 和 138.97 ms 来执行该任务, 它们的上下四分位数范围也都高于微软 Azure Functions. 如果不考虑基于内存消耗策略的微软 Azure Functions, 亚马逊 Lambda 的执行时间更短, 花费的成本也是更低, 但阿里巴巴 Function Compute 的执行结果和亚马逊 Lambda 相近, 只高出约 15 ms. 在这两个平台执行 CPU 密集型任务的成本上, 阿里巴巴 Function Compute 和亚马逊 Lambda 也相近, 只高出 0.000 03 美元. 根据图 10 中关于 CPU 密集型任务的性能分布, 谷歌 Cloud Functions 和微软 Azure Functions 所获得的执行性能数据分布更加分散, 而亚马逊 Lambda 和阿里巴巴 Function Compute 所获的性能数据分布较为稳定. 这表明亚马逊 Lambda 和阿里巴巴 Function Compute 在执行 CPU 密集型任务时更加可靠, 大部分执行结果分别落在 122.52–125.22 ms 之间, 以及 136.88–143.53 ms 之间. 此外, 从图 10 中还可以观察到谷歌 Cloud Functions 和微软 Azure Functions 平台产生了大量异常值, 这些异常值表示与其他性能数据相比明显偏离正常范围的数据点. 因此, 综合考虑任务执行延迟大小, CPU 密集型任务适合在微软 Azure Functions、亚马逊 Lambda 和阿里巴巴 Function Compute 上执行, 而且这些平台花费的成本相对更少; 而就性能稳定性而言, CPU 密集型任务更适合执行在亚马逊 Lambda 和阿里巴巴 Function Compute 这两个平台上执行.

对于内存密集型任务, 根据表 7 的中位数结果, 微软 Azure Functions 执行时间为 116.36 ms, 阿里巴巴 Function Compute 执行时间为 341.98 ms, 亚马逊 Lambda 执行时间为 556.36 ms, 而谷歌 Cloud Functions 执行时间为 709.57 ms. 可以看出, 微软 Azure Functions 在执行内存密集型任务时所需的时间最少, 而谷歌 Cloud Functions 执行这类任务所需时间则是微软 Azure Functions 的 7 倍, 这表明谷歌 Cloud Functions 并不适合执行内存密集型任务. 而且, 微软 Azure Functions 执行该任务花费的成本也是最低 (0.000 43 美元). 谷歌 Cloud Functions 执行该任务的成本约是微软 Azure Functions 成本的 5 倍. 若不考虑基于内存消耗的微软 Azure Functions, 其他 3 个平台中阿里巴巴 Function Compute 的执行时间相对较短, 成本也是相对较低. 从图 10 中关于内存密集型任务的性能数据分布来看, 亚马逊 Lambda 获得的执行性能数据最为稳定, 其执行时间限定在 538.77–574.46 ms 之间. 而其他平台产生的性能数据相对不稳定, 且存在异常值. 因此, 考虑任务执行性能数据的大小, 内存密集型任务适合在微软 Azure Functions 和阿里巴巴 Function Compute 上执行, 且这些平台执行该任务的成本更低; 而就任务执行的性能稳定性而言, 内存密集型任务适合在亚马逊 Lambda 上执行.

对于磁盘 IO 密集型任务, 根据表 7 的中位数结果, 亚马逊 Lambda、谷歌 Cloud Functions 和阿里巴巴 Function Compute 分别执行 4 233.19 ms、4 380.13 ms 和 3 349.69 ms. 总体来看, 这 3 个平台在执行磁盘 IO 密集型任务时需要超过 3 s 的执行时间, 但其中阿里巴巴 Function Compute 所需时间相对较短, 且其花费的成本相对更少. 与此相比, 微软 Azure Functions 仅需 469 ms 的执行时间, 几乎提升了 10 倍. 另外, 相比较于其他平台, 微软 Azure Functions 在执行该任务的成本上也是最低的. 从图 11 的性能数据分布来看, 亚马逊 Lambda 和微软 Azure Functions 展现出稳定的执行结果, 而谷歌 Cloud Functions 和阿里巴巴 Function Compute 所产生的性能数据更加分散, 并且存在更多异常值. 因此, 考虑任务执行性能数据的大小, 磁盘 IO 密集型任务适合在微软 Azure Functions 和阿里巴巴 Function Compute 上执行, 且这两个平台相比较其他平台来说, 执行该任务的成本相对更低; 而就执行的性能稳定性而言, 该任务适合在亚马逊 Lambda 和微软 Azure Functions 上执行.

对于网络密集型任务, 根据表 7 的中位数结果, 微软 Azure Functions 和阿里巴巴 Function Compute 相比亚马逊 Lambda 和谷歌 Cloud Functions 获得了更好的执行性能结果和更低的成本. 具体来说, 微软 Azure Functions 执行时间为 109.57 ms, 成本是 0.000 42 美元. 阿里巴巴 Function Compute 执行时间为 159.82 ms, 成本是 0.000 53 美元. 而亚马逊 Lambda 和谷歌 Cloud Functions 执行性能则慢于微软 Azure Functions 和阿里巴巴 Function Compute 的 2 倍左右, 花费的成本上也高两倍左右. 从图 11 中的网络密集型任务的性能数据分布来看, 阿里巴巴 Function Compute 和亚马逊 Lambda 执行的性能结果最为稳定, 而微软 Azure Functions 和谷歌 Cloud Functions 所获得的性能结果相对较为分散. 因此, 考虑任务执行性能数据的大小, 网络密集型任务适合在微软 Azure Functions 和阿里巴巴 Function Compute 上执行, 且这些平台执行该任务花费的成本也低; 而就任务执行的性能稳定性而言, 网络密集型任务适合在亚马逊 Lambda 和阿里巴巴 Function Compute 上执行.

我们进一步分析了各个平台的成本计算方式. 成本主要由两部分组成: 调用次数成本和资源使用量成本. 在调用次数成本方面, 观察到各个平台的单次调用价格相差不大. 具体来说, 亚马逊 Lambda、微软 Azure Functions 和阿里巴巴 Function Compute 的单次调用价格均为 0.000 000 2 美元, 而谷歌 Cloud Functions 的单次调用价格为 0.000 000 4 美元. 在资源使用量成本方面, 各个平台的每秒资源使用价格也相近. 这里的每秒资源使用价格是基于每 1 GB 内存大小和平台在此内存大小分配的 vCPU 大小下函数每秒所要花费的成本. 具体来说, 亚马逊 Lambda 的每秒资源使用价格为 0.000 016 66 美元, 谷歌 Cloud Functions 的每秒资源使用价格为 0.000 016 50 美元, 微软 Azure Functions 的价格是 0.000 016 00 美元, 阿里巴巴 Function Compute 的价格为 0.000 016 68 美元. 总之, 各平台计费单价相差不大. 因此, 在内存大小一致的情况下, 任务成本的大小主要取决于平台执行该任务的执行时间. 基于此, 我们得出了任务执行时间较短的平台也将让开发者支付相对低的成本.

发现: 微基准测试程序旨在消耗服务器无感知平台特定资源, 例如 CPU、内存、磁盘读写和网络. 不同平台在执行性能方面表现不一致. 结果显示, 微软 Azure Functions 提供了最快的速度, 可能是因为该平台采用了与其他平台不同的内存消耗策略, 而且基于更快的执行性能, 在微软 Azure Functions 上花费更低的成本. 对于其他 3 个采用基于内存提前分配策略的平台而言, 阿里巴巴 Function Compute 可以获得相对较快的执行速度和相对较低的成本. 然而, 考虑平台的执行性能稳定性, 不同平台在执行不同类型任务时的稳定性表现不一致. 举例来说, CPU 密集型任务和网络密集型任务在亚马逊 Lambda 和阿里巴巴 Function Compute 上执行更为稳定, 而内存密集型任务在亚马逊 Lambda 上执行更为稳定, 磁盘 IO 密集型任务在亚马逊 Lambda 和微软 Azure Functions 上执行更为稳定. 总体而言, 亚马逊 Lambda 在这些特定资源消耗任务上表现出较好的稳定性, 但该平台的执行速度不总是最优的. 如果同时考虑执行速度和执行稳定性, 目前还没有能够同时满足这两个需求的服务器无感知平台. 这意味着根据具体任务需求, 需要在执行速度和执行稳定性之间进行权衡选择适合的平台.

(2) 宏基准测试程序的执行性能: 表 8、图 12 和图 13 展示宏基准测试程序在不同平台上执行性能. 我们将详细分析图片处理任务、语音识别任务、图计算任务、机器学习训练任务和机器学习推理任务执行性能.

表 8　宏基准测试程序的执行性能结果和成本

任务缩写表示	执行性能	亚马逊 Lambda	谷歌 Cloud Functions	微软 Azure Functions	阿里巴巴 Function Compute
Image_task	中位数 (ms)	3 365.12	3 619.12	1 087.19	2 998.17
	成本 (美元)	0.007 21	0.007 86	0.002 37	0.006 45
	上四分位数 (ms)	3 443.24	3 826.90	1 169.21	3 190.15
	下四分位数 (ms)	3 286.33	3 475.99	1 009.83	1 076.59
Speech_task	中位数 (ms)	52 449.17	41 583.70	21 448.06	48 505.30
	成本 (美元)	0.437 28	0.343 47	0.134 25	0.404 73
	上四分位数 (ms)	53 508.18	43 590.91	22 237.16	50 746.99
	下四分位数 (ms)	51 500.23	39 742.40	17 910.65	46 329.59
Graph_task	中位数 (ms)	897.08	282.01	59.89	196.65
	成本 (美元)	0.002 07	0.000 98	0.000 32	0.000 61
	上四分位数 (ms)	2 174.70	356.61	94.22	273.28
	下四分位数 (ms)	238.88	221.64	34.77	144.64
Train_task	中位数 (ms)	37 949.61	40 856.46	11 985.54	35 950.95
	成本 (美元)	0.316 45	0.337 47	0.075 11	0.300 03
	上四分位数 (ms)	38 505.36	42 207.05	14 337.24	36 720.12
	下四分位数 (ms)	37 526.99	39 706.88	11 517.34	35 324.74
Inference_task	中位数 (ms)	12 855.08	11 254.29	5 130.30	11 126.94
	成本 (美元)	0.107 33	0.093 25	0.032 26	0.093 00
	上四分位数 (ms)	13 000.32	12 043.46	5 815.84	11 513.01
	下四分位数 (ms)	12 716.57	10 836.63	4 888.98	10 753.17

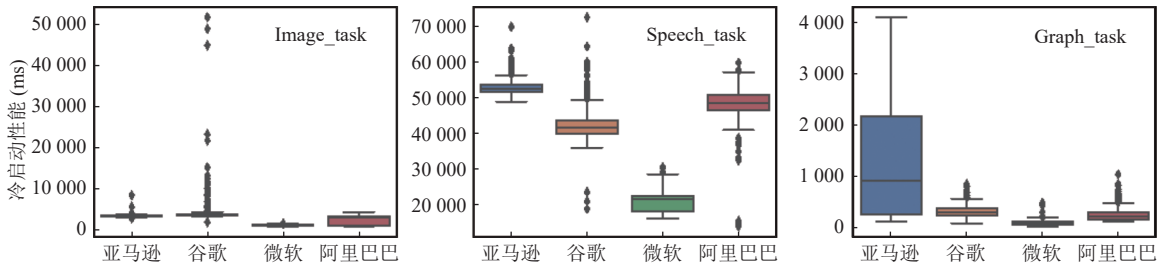

图 12 图片处理任务、语音识别任务和图计算任务在不同平台上的执行性能分布

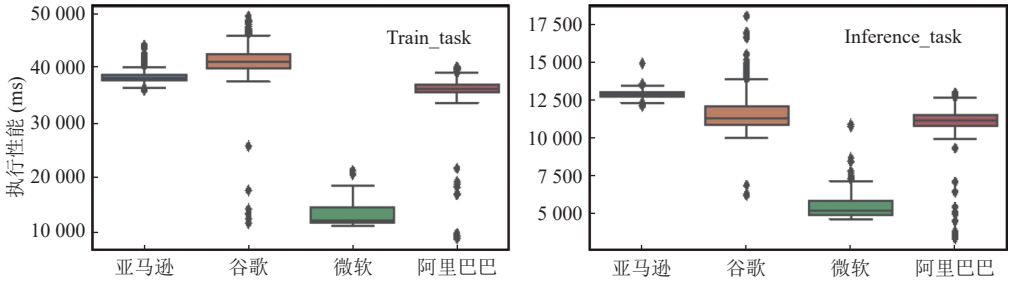

图 13 机器学习训练任务和机器学习推理任务在不同平台上的执行性能分布

对于图片处理任务, 根据表 8 中关于该任务的中位数性能结果, 亚马逊 Lambda、谷歌 Cloud Functions、微软 Azure Functions 和阿里巴巴 Function Compute 分别执行需要 3 365.12 ms、3 619.12 ms、1 087.19 ms 和 2 998.17 ms. 结果显示, 微软 Azure Functions 比其他 3 个平台执行图片处理任务快了近 3 倍, 花费的成本也是最低的 (0.002 37 美元). 若不考虑基于内存消耗策略的微软 Azure Functions, 其他 3 个平台中阿里巴巴 Function Compute 的执行该任务的时间相对较短, 花费的成本也相对较低, 为 0.006 45 美元. 根据上下四分位数范围的结果来看, 微软 Azure Functions 的四分位数范围 (1 009.83–1 169.21 ms) 也低于其他 3 个平台的范围, 即亚马逊 Lambda 的执行延迟结果在 3 286.33–3 443.24 ms、谷歌 Cloud Functions 的执行延迟结果在 3 475.99–3 826.90 ms, 阿里巴巴 Function Compute 的执行延迟结果在 1 076.59–3 190.15 ms. 根据图 12 中图片处理任务的性能分布, 阿里巴巴 Function Compute 的性能结果的箱线图是最大, 意味着其获得的性能数据是最为分散. 亚马逊 Lambda 和微软 Azure Functions 获得的性能数据相对集中, 而谷歌 Cloud Functions 获得的结果则存在大量偏离正常范围的异常值. 因此, 对于图片处理任务而言, 微软 Azure Functions 和阿里巴巴 Function Compute 可以提供最快的执行速度, 且这些平台产生的成本更低; 而考虑到平台所获性能结果的稳定性, 图片处理任务更适合在亚马逊 Lambda 和微软 Azure Functions 上执行.

对于语音识别任务, 相比图片处理任务, 它需要更长的任务执行时间. 根据表 8 中的语音识别任务的中位数结果, 亚马逊 Lambda 执行该任务需要约 52 s, 谷歌 Cloud Functions 需要约 41 s, 微软 Azure Functions 需要约 21s, 阿里巴巴 Function Compute 需要约 48 s. 在这类任务中, 亚马逊 Lambda 的执行时间较长, 产生的成本更高 (0.437 28 美元), 而微软 Azure Functions 的执行时间较短, 产生的成本较低 (0.134 25 美元). 若不考虑基于内存消耗的微软 Azure Functions, 谷歌 Cloud Functions 和阿里巴巴 Function Compute 在执行该任务的时间上相对较短. 根据图 12 中语音识别任务的执行结果分布, 谷歌 Cloud Functions、微软 Azure Functions 和阿里巴巴 Function Compute 获得的性能结果相对分散, 而亚马逊 Lambda 获得的性能结果相对集中, 亚马逊 Lambda 仍然存在一些异常值. 因此, 对于语音识别任务而言, 微软 Azure Functions、谷歌 Cloud Functions 和阿里巴巴 Function Compute 可以提供最快的执行性能, 产生的成本比亚马逊 Lambda 低; 考虑到所获性能结果的稳定性, 语音识别任务更适合在亚马逊 Lambda 上执行.

对于图计算任务, 根据表 8 关于该任务的中位数数据, 4 个平台能够以 1 s 内完成执行. 具体来说, 亚马逊 Lambda 执行时间为 897.08 ms, 谷歌 Cloud Functions 执行时间为 282.01 ms, 微软 Azure Functions 执行时间为

59.89 ms, 阿里巴巴 Function Compute 执行时间为 196.65 ms. 这些结果表明亚马逊 Lambda 执行图计算任务需要更长的时间, 且该平台产生的成本更高, 而微软 Azure Functions 和阿里巴巴 Function Compute 则需要较少的时间, 产生的成本分别是 0.000 32 美元和 0.000 61 美元. 图计算任务的特点是先构建固定节点个数大小的图, 但每次执行时节点连接信息都是不同的, 然后对生成的图执行 PageRank 算法以挑选出重要节点. 因此, 这种任务的计算复杂度是不确定的, 亚马逊 Lambda 在处理此类任务时的能力可能相对不稳定. 而且, 根据图 12 中图计算任务的性能数据分布来看, 亚马逊 Lambda 在执行这类任务时所产生的性能结果也表现出不稳定, 其上下四分位数的范围为 238.88 ms 到 2 174.70 ms, 存在着 10 倍的性能差距. 相反, 谷歌 Cloud Functions、微软 Azure Functions 和阿里巴巴 Function Compute 所获得的图计算任务结果相对较为稳定. 因此, 对于图计算任务而言, 微软 Azure Functions 和阿里巴巴 Function Compute 可以提供最快的执行性能, 这些平台产生相对低的成本; 考虑到所获性能的稳定性, 图计算任务不适合执行在亚马逊 Lambda 上执行, 而可以在谷歌 Cloud Functions、微软 Azure Functions 和阿里巴巴 Function Compute 上执行.

对于机器学习训练任务, 根据表 8 中该任务的中位数结果, 微软 Azure Functions 执行该任务所需的执行性能是最优的, 约需 11 s, 而且产生的成本也是最低的 (0.075 11 美元), 而其他 3 个平台的执行时间超过 35 s, 产生的成本超过 0.3 美元. 若不考虑基于内存消耗策略的微软 Azure Functions, 其他平台中阿里巴巴 Function Compute 的执行时间相对较短, 仅需 35 950.95 ms 完成机器学习训练任务, 而亚马逊 Lambda 和谷歌 Cloud Functions 分别执行 37 949.61 ms 和 40 856.46 ms. 根据图 13 中机器学习训练任务的性能数据分布, 亚马逊 Lambda 和阿里巴巴 Function Compute 获得的性能结果更为稳定, 其上下四分位数的范围分别为 37 526.99–38 505.36 ms, 以及 35 324.74–36 720.12 ms. 总体而言, 这些性能数据在范围内相差 1 000 ms 左右. 因此, 对于机器学习训练任务而言, 微软 Azure Functions 和阿里巴巴 Function Compute 可以提供相对快的执行性能, 且这两个平台产生的成本相对低; 考虑到性能数据分布的稳定性, 可选择在亚马逊 Lambda 和阿里巴巴 Function Compute 上执行该任务.

对于机器学习推理任务, 根据表 8 中该任务的中位数性能结果, 微软 Azure Functions 执行该任务的时间约 5 s, 而其他 3 个平台的执行时间超过 11 s. 从成本来看, 在微软 Azure Functions 产生的成本约为 0.03 美元, 而在其他平台产生的成本约为 0.1 美元. 若不考虑基于内存消耗策略的微软 Azure Functions, 其他 3 个平台中阿里巴巴 Function Compute 的执行时间相对较短, 约为 11 s 左右. 根据图 13 中机器学习推理任务的性能数据分布, 亚马逊 Lambda 和阿里巴巴 Function Compute 获得的性能结果更为稳定. 因此, 对于机器学习推理任务而言, 微软 Azure Functions 和阿里巴巴 Function Compute 可以提供相对快的执行性能, 这些平台产生相对低的成本; 考虑到性能数据分布的稳定性, 可以选择在亚马逊 Lambda 和阿里巴巴 Function Compute 上执行.

根据前面关于微基准测试程序的成本分析可知, 各个平台的计费价格相差不大. 在保持分配内存一致的情况下, 执行时间短的应用产生的成本较低. 这一现象在宏基准测试程序的执行性能和成本结果中也得到了验证.

发现: 宏基准测试程序是需要消耗平台上多种资源的复杂任务. 结果显示这些复杂任务在不同平台上的执行性能表现和微基准测试程序有着相似或不同之处. 考虑复杂任务的执行性能速度的大小, 仍然是基于内存消耗的微软 Azure Functions 提供最快的执行性能, 且这个平台产生的成本最低. 但对于其他 3 个基于内存分配的平台, 阿里巴巴 Function Compute 可以为复杂任务提供更快的执行效率, 并产生更低的成本. 这一点和微基准测试程序的结果保持一致. 对于 4 个服务器无感知平台的性能稳定性, 不同的任务在不同的平台上执行其稳定性存在差异. 图片处理任务在亚马逊 Lambda 和微软 Azure Functions 上执行稳定; 机器学习训练和推理任务在亚马逊 Lambda 和阿里巴巴 Function Compute 上执行稳定; 语音识别任务在亚马逊 Lambda 执行稳定. 从这些任务来看, 和微基准测试程序表现类似, 亚马逊 Lambda 为大部分任务提供强的执行稳定性, 但执行速度不一定是最优的. 不同的是, 对于图计算任务, 即任务执行的计算复杂度不确定的任务, 亚马逊 Lambda 是最不稳定的, 表明了该平台对这类任务的处理能力需进一步提高. 但是, 其他 3 个平台可以为该类任务提供稳定性能结果.

5 启示和机会

本节将讨论基于本文研究结果为开发者和云计算厂商带来的启示, 以及为研究者带来的潜在研究机会.

5.1 启 示

(1) 选择广泛采用的编程语言: 基于本文提供的编程语言结果, 开发者可以根据自己的偏好和需求来选择广泛采用的编程语言来开发应用. Python、JavaScript 和 Java 都是主流服务器无感知平台所支持的语言, 这表明它们在服务器无感知社区中非常流行[14], 并且被平台广泛支持. 对于新手的开发者来说, 选择这些流行的编程语言进行服务器无感知计算应用开发是一个明智的选择. 选择广泛采用的编程语言有几个好处. 首先, 当开发者想要将应用从一个平台移植到另一个平台时, 应用代码的改动量将大大降低. 因为这些广泛支持的编程语言在不同平台上都具有高度的兼容性, 开发者可能只需要适应不同平台对于服务器无感知函数的编写格式差异即可. 其次, 广泛采用的编程语言通常拥有更多的开发资源和支持. 由于它们在服务器无感知计算社区中的流行度, 开发者可以更容易地找到相关的文档、教程、库和工具来支持开发工作, 这将更快速地解决问题、提高开发效率.

(2) 注意平台的部署包大小限制: 每个平台都有不同的部署包大小限制, 开发者编写应用功能时应该意识到这一限制, 并确保应用不会超过限制, 以降低部署失败的概率. 此外, 由于服务器无感知计算更适用于执行轻量级任务, 对于一些复杂的任务, 最好尽量避免在此类平台上部署. 而且, 开发者还需要注意任务中使用的重量级依赖库, 因为这些库往往会加大部署包大小, 增加失败的风险. 这一点也建议开发者要去设计和开发轻量级的应用, 使用更精简的代码和依赖库, 以降低应用依赖和提高部署效率.

(3) 平衡内存分配大小与成本和性能之间的关系: 一般来说, 用户关注的端到端性能主要由冷启动性能和执行性能组成. 1) 对于冷启动性能, 本文的冷启动性能分析揭示了内存大小对冷启动性能和数据稳定性产生影响. 举例来说, 亚马逊 Lambda 在 128 MB 内存分配下表现出的冷启动性能数据并不稳定, 而阿里巴巴 Function Compute 在执行 Python 应用时, 内存越大所获得的冷启动性能越好. 2) 对于执行性能, 通过本文的第 3 节特征总结可知, 开发者需为任务在平台上真正执行的时间付费. 真正执行的时间受分配的内存大小影响, 因为它决定了平台分配给任务的资源大小, 这进一步影响了执行性能. 付费模型总结了在大多数平台上内存分配大小直接关系到开发者需要支付的费用. 具体来说, 在平台上分配的内存大小 (决定了资源使用量) 以及应用在该内存上真正执行的时间 (执行性能) 是影响应用成本计算的关键因子. 因此, 内存分配大小、端到端性能 (包括冷启动性能和执行性能) 和成本之间存在关系. 在这种情况下, 开发者需要谨慎考虑如何在冷启动性能、应用执行性能和其成本之间取得平衡, 以选择合适的内存分配大小. 过小的内存大小可能导致冷启动性能结果和稳定性下降, 从而使得应用响应时间延迟长, 甚至可能发生程序崩溃. 然而, 过大的内存大小可能导致应用在执行期间的资源浪费, 而且, 内存过大不一定使应用的执行性能结果得到进一步优化[53], 进而可能产生额外的费用支出. 在选择内存分配大小时, 开发者可以考虑以下几点. 首先, 他们应该了解应用执行时真正的内存需求, 以确定最低要求的内存大小与最优冷启动性能和最佳执行性能之间的平衡点. 通过对应用进行监测, 开发者可以获得有关应用运行时的内存使用情况以及冷启动性能和应用执行性能表现的数据, 从而更好地评估应用运行时的内存需求. 其次, 开发者应用考虑到预算限制, 他们需要综合可用预算和应用的性能需求 (稳定的冷启动性能, 以及成本效益高且相对快速的执行性能), 选择一个合理的内存分配大小, 以达到最佳的性价比.

(4) 注意负载请求大小: 在处理负载请求大小时, 开发者需要注意不同平台对请求负载数据大小的限制. 以同步调用为例, 亚马逊 Lambda 和阿里巴巴 Function Compute 分别将请求负载大小限制在 6 MB 和 32 MB 以内. 如果请求的大小超过了这些限制, 建议开发者使用云存储服务等方式先保存请求数据. 使用云储存服务可能是一种常见的解决方案. 开发者可以将超出限制的请求数据存储在可靠的云存储中, 如亚马逊 S3 或阿里云对象存储. 数据存储后, 开发者可以在代码中直接读取数据完成后续功能. 或者开发者设置事件触发器, 当新的数据写入数据时, 触发相应的函数执行. 这样, 开发者就可以绕过平台对请求负载大小的限制, 确保请求的顺利处理. 使用云存储服务来处理超大请求的优势在于, 它提供了高可靠性和可扩展性. 云存储服务通常会自动处理数据备份和冗余, 确保数据安全性和可用性. 此外, 云存储还具备高度可扩展特性, 处理大量的数据请求, 并在需要时自动进行扩展, 以满足并发和高吞吐量的要求. 当然, 使用云存储服务也需要额外的成本和延迟. 存储数据和触发函数之间存储一定的时间延迟, 可能会对应用的响应时间产生一定影响. 此外, 云存储服务的使用可能会增加额外的存储和网络传输

成本. 因此, 在设计应用时, 开发者需要综合考虑延迟和成本因素, 权衡使用云存储服务的利弊.

(5) 改进不同编程语言的冷启动机制: 根据第 4.1 节中关于平台内的冷启动延迟结果比较, 可以看出每个平台在不同编程语言应用的冷启动延迟方面存在差异. 例如, 亚马逊 Lambda 在执行 Python 和 JavaScript 应用时的冷启动延迟要低于执行 Java 应用的延迟, 几乎快了将近两倍. 而微软 Azure Functions 在执行 Java 应用时产生更低的冷启动延迟, 低于近 700 ms. 这表明每个平台对不同编程语言应用的启动机制和处理能力可能存在差异. 基于我们的结果, 各云计算厂商应该重视平台在这一问题上的表现, 并致力于改进冷启动机制. 一个理想的服务器无感知平台应该具备低冷启动延迟且在不同语言之间延迟差异不大. 开发者期待能够使用这样的平台, 无论是在 Python、JavaScript 还是 Java 等多种编程语言下, 都能获得类似的冷启动性能. 为了改进冷启动机制, 云计算厂商可以采取一系列措施. 首先, 他们可以优化服务器启动过程, 减少启动时间和资源消耗. 其次, 他们可以针对不同编程语言的特点, 针对性优化运行时环境和加载机制, 以提高冷启动效率. 此外, 他们还可以探索智能预热技术, 提前根据行为模式或预测需求进行函数的预加载, 以减少冷启动延迟. 这些改进措施将有助于提升平台的性能和用户体验. 总之, 改进不同编程语言的冷启动机制是云计算厂商的重要任务. 他们应该持续关注每个平台在这方面的表现, 并积极改进平台启动机制, 以满足开发者对低延迟且延迟差异不大的服务器无感知平台的期望.

(6) 选择提供低冷启动延迟的平台: 根据第 4.1 节中关于平台间的冷启动延迟结果比较, 可以得出结论: 在不同服务器无感知平台下, 3 类主流编程语言应用的表现相似, 即亚马逊 Lambda 和阿里巴巴 Function Compute 在执行 Python 应用、JavaScript 应用和 Java 应用时, 其冷启动延迟都明显低于谷歌 Cloud Functions 和微软 Azure Functions. 这意味着开发者可以有针对性地选择提供低冷启动延迟的平台来执行应用, 从而提高整体的用户体验. 通过选择具有较低冷启动延迟的平台, 开发者可以减少用户等待时间, 提高应用响应速度, 并增强用户对应用的满意度. 此外, 较低的冷启动延迟还能对应用的可伸缩性和弹性产生积极影响. 在面对突发的高并发流量或需要频繁启动新实例的场景下, 低冷启动延迟的平台能够更快速地响应需求, 确保应用的可用性和性能稳定性.

(7) 选择执行性能最优的平台: 1) 根据第 4.2 节的研究结果, 在基于内存消耗的微软 Azure Functions 上任何任务都可以获得最快的执行性能和最低的成本. 然而, 微软 Azure Functions 的缺点是开发者失去了对内存分配的控制权. 相比之下, 对于其他 3 个基于内存分配的平台, 阿里巴巴 Function Compute 能够提供更快的执行效率和更低的成本, 该平台在执行性能和成本之间提供了不错的平衡, 具有广阔的应用前景. 2) 不同任务类型对不同平台的执行性能稳定性产生影响. 例如, 亚马逊 Lambda 在图片处理任务上执行稳定, 但对于图计算任务表现不佳. 对于机器学习训练和推理任务, 亚马逊 Lambda 和阿里巴巴 Function Compute 表现较好. 因此, 选择执行性能最优的平台需要考虑特定任务的性质. 3) 我们比较了 4 个平台在执行同一任务的成本. 在保持平台内存大小一致的情况下, 任务成本取决于各平台执行该任务的执行时间. 我们观察到不同平台的计费单价相差不大, 因此, 成本取决于执行任务所需的时间. 短时间内完成任务的平台通常让开发者花费更低的成本, 因此, 选择执行性能最优的平台也涉及成本效益的考量. 选择执行性能最优的平台并非固定不变的选择, 而是一项需要动态权衡特定需求、任务性质和成本效益的决策过程. 随着云计算领域的不断演进, 开发者应该根据任务要求和性能目标灵活选择服务器无感知平台, 以最大程度地满足其需求.

5.2 研究机会

(1) 分解长时应用为短时函数: 服务器无感知平台存在着函数执行时间限制, 如亚马逊 Lambda 最多执行 15 min 函数, 这表明服务器无感知计算的架构设计主要适用于短时任务. 然而, 若帮助开发者实现长时应用的执行, 研究者可以从以下几个方面展开研究. 一种方法是通过对运行时执行进行分析, 将长时间任务的功能分解为多个短时的子任务函数, 可以确保每个子任务在允许的函数执行时间内完成, 并避免引起执行失败. 这样, 每个子任务可以在独立的函数中执行, 通过事件驱动机制或异步调用的方式进行协调和串联. 举例来说, 我们宏基准测试程序中包含一个图片处理任务, 该任务对待处理图片进行翻转、旋转、过滤、调色和缩略图等多种操作. 逐个处理完所有操作可能需要执行很长时间. 基于此, 将该任务的多个图像处理操作转化为多个短时函数可能是一个有效性的优化策略. 具体来说, 每个图像处理操作 (如翻转、旋转、过滤、调色和缩略图等) 可以作为一个独立的函数, 这样

的函数在有限的执行时间内完成特定任务, 且不容易违反超时时间限制. 此外, 这些独立的函数可以并行执行, 而不必等待前一个操作完成, 这样可以显著缩短总体处理时间以提高性能. 另一种方法是在长时间任务的功能中插入检查点, 以便在特定时间间隔或达到一定条件时进行检查并分解任务. 检查点用来保存任务的状态和进度, 以便在函数执行时恢复并继续执行. 在设计长时应用时, 开发者还应考虑任务分解的合理性和性能的影响. 合理的任务分解事根据任务的特性和依赖关系进行, 以确保拆分后子任务可以并行执行在最短时间内. 对于需要跨函数协作的长时任务, 还需设计适当的通信和数据传递机制, 以确保子任务之间的信息交换和协调.

(2) 单独配置和学习 CPU 和内存大小: 大多数平台根据分配的内存大小等比例分配 CPU 能力, 然而不同任务对资源需求是不同的, 而不同平台的处理能力也不尽相同. 以 CPU 密集型任务为例, 亚马逊 Lambda 能够提供更快的执行性能, 而在内存密集型任务方面, 阿里巴巴 Function Compute 则能够获得更快的执行性能. 这表明 CPU 能力不应仅仅根据内存分配大小进行硬性配置. 如果仍然按照传统的方式分配 CPU, 就会导致一些内存密集型任务浪费计算资源, 而在 CPU 密集型任务中则会浪费内存资源. 因此, 最好的方式是所有平台都提供单独配置 CPU 和内存大小的选项. 根据本文对部署特点的总结, 目前只有谷歌 Cloud Functions 和阿里巴巴 Function Compute 实现了这种方式. 我们以阿里巴巴 Function Compute 为例, 使用微基准测试程序的 CPU 密集型任务, 评估了单独配置 CPU 大小对应用性能优化的影响. 具体来说, 我们增大该任务的输入 (矩阵大小设置为 2 000), 以增加任务计算量来更好地比较性能结果的变化. 在测试不同配置下的应用性能时, 输入大小保持不变. 在这个例子中, 该任务的内存分配大小保持在 512 MB, 而 vCPU 大小分别设置为 0.15 核、0.25 核、0.35 核和 0.45 核. 表 9 中列出了 CPU 密集型任务在不同 vCPU 配置下的执行性能结果.

表 9　CPU 密集型任务在阿里巴巴 Function Compute 不同 vCPU 配置下的执行性能结果

vCPU (核)	0.15	0.25	0.35	0.45
执行性能 (ms)	2 186.67	1 260.82	890.29	721.05

从结果中观察到, 在保持分配内存大小不变的情况下, 提高 vCPU 的计算能力时, 该 CPU 密集型任务的执行延迟从 2 186.67 ms 优化到 721.05 ms. 因此, 可以得出通过单独配置 CPU 大小可以有效地优化应用的性能的结论. 然而, 这些平台要求开发者自行考虑配置多大的 CPU 值, 这可能会导致资源利用率不够优化. 因此, 一个研究机会是探究如何在最大化资源利用率、性能和成本的前提下, 为应用提供细粒度的内存和 CPU 配置选项. 一个可能的方法是设计一个目标优化问题, 利用细粒度的内存和 CPU 配置等来构建一个优化函数. 这样的优化函数可以考虑应用的特定需求以及平台的处理能力和资源分配策略, 从而推到处最佳的 CPU 和内存配置方案. 这样一来, 开发者就可以在充分利用资源、提高性能的同时, 最大限度地降低成本. 此外, 该研究还可以帮助平台提供商更好地理解用户需求, 改进其服务, 并为开发者提供更多的灵活性和可定制性.

(3) 支持 GPU 实例环境: 随着深度学习任务的迅速增长, 对 GPU 底层硬件支持的需求变得越来越迫切. 然而, 在所调研的 4 个主流商业服务器无感知平台中, 有 3 个平台都不支持部署 GPU 资源需求的函数. 这说明目前云计算厂商在支持 GPU 函数实例方面可能存在一定的技术难题. 尽管如此, 这也是对研究者的一个研究机会. 将 GPU 设计为按需使用的服务方式将对准实时推理应用等场景的实际需求和前景. 首先, 研究者可以着眼于改进云服务平台的基础架构, 以适应 GPU 计算的需求. 这可能涉及对硬件设施进行升级或优化, 以支持更高性能的 GPU 实例. 其次, 研究者可以考虑设计和实施针对 GPU 实例的调度和资源管理算法. 这些算法帮助平台有效地分配和管理 GPU 资源, 以满足不同用户和应用的需求. 例如, 可以研究智能调度策略, 根据任务的性能和优先级动态分配 GPU 实例, 以最大化资源利用率和系统性能. 此外, 研究者可以研究如何优化 GPU 函数实例的部署和运行. 这包括研究高效的容器化解决方案, 以提供快速的函数实例启动和执行环境. 另外, 还可以研究 GPU 函数实例的资源利用率和计算效率优化, 以提高整体性能并降低成本. 此外, 随着边缘计算的兴起, 研究者还可以探究将 GPU 实例环境扩展到边缘设备上的可能性. 这有助于在边缘环境中进行实时的深度学习推理和处理, 提供更快速的响应和更高的隐私保护. 在研究支持 GPU 实例环境的过程中, 还应该考虑如何解决安全性和隐私保护的问题. 因为 GPU 实例环境通常会包含敏感数据和模型, 确保数据的安全传输和处理是至关重要的.

(4) 重设计内存资源使用策略: 根据特征总结结果, 亚马逊 Lambda、谷歌 Cloud Functions、阿里巴巴 Function Compute 采用了开发者提前制定内存分配大小的策略来执行应用, 而微软 Azure Functions 则采用了动态内存分配的策略. 从第 5.2 节的执行性能结果来看, 微软 Azure Functions 在不同类型任务上展现出更好的执行性能, 这表明基于内存消耗的策略能够提供有效的资源使用和良好的性能结果. 与之相对, 目前广泛使用的提前分配内存大小的策略可能无法灵活满足任务的资源需求, 从而降低了一定的性能效益. 因此, 研究者可以进一步思考如何在服务器无感知计算上重设计内存资源的使用策略, 以提高应用的性能, 可以更好地满足不同应用的资源需求, 提高系统的整体性能. 然而, 需要注意的是, 使用基于内存消耗策略也让开发者失去对内存分配的控制权. 这是研究者需要进一步关注的方面. 在设计新策略时, 应考虑如何提供一定程度的控制, 以便针对特定应用的需求进行调优和定制. 同时, 还应该探索如何平衡动态内存分配策略的灵活性和开发者的需求, 以提供更加细粒度的内存管理功能.

(5) 优化冷启动性能: 根据我们的结果, 不同服务器无感知平台在执行不同编程语言的应用时, 出现明显的冷启动性能差异. 亚马逊 Lambda 和谷歌 Cloud Functions 在执行 Python 和 JavaScript 应用时表现出较低的冷启动延迟, 但在执行 Java 应用时延迟较高, 而似乎内存分配大小对这两个平台的冷启动延迟没有显著影响. 相反, 微软 Azure Functions 在执行 Java 应用时产生较低的冷启动延迟, 而阿里巴巴 Function Compute 在执行 JavaScript 应用时产生较低的冷启动延迟, 但在执行 Python 应用时产生最高的冷启动延迟. 基于此, 我们认为潜在的研究机会是如何为不同编程语言的应用提供一致的冷启动性能. 研究者可以尝试从以下几个方面考虑: 1) 开发通用的执行引擎, 这帮助适应不同编程语言的应用执行. 这个引擎可能采用跨语言的虚拟机或解释器, 以确保所有编程语言都能在相似的运行环境中执行, 从而提供一致的冷启动性能. 2) 制定服务器无感知平台通用的应用标准, 使开发者能够以一种方式编写应用, 而不必担心语言差异对性能的影响. 这些标准可以包括应用架构、内存管理和资源分配规范, 以确保不同编程语言的应用遵循一致的开发标准. 这些尝试可能帮助实现一致的冷启动性能, 不论应用是使用哪种编程语言编写, 从而提高服务器无感知计算的可用性和广泛适用性.

(6) 研究不同应用场景的性能优化: 根据第 4.2 节的执行性能分析结果, 不同类型的任务在不同平台的执行性能表现存在差异. 例如, 图片处理任务在亚马逊 Lambda 和微软 Azure Functions 上执行稳定; 机器学习训练和推理任务在亚马逊 Lambda 和阿里巴巴 Function Compute 上执行稳定; 语音识别任务在亚马逊 Lambda 执行稳定. 这说明平台对不同类型任务的处理能力尚不一致, 这激励着研究者进一步探究不同应用场景的性能优化工作. 具体而言, 研究者可以结合不同应用场景的特点以及应用开发的结构依赖信息, 来确定优化的关键流程. 例如, 对于语音识别任务, 可以考虑在处理过程中进行子任务并发以加快执行性能; 对于机器学习训练任务, 可以考虑流程分解, 同时对 CPU 资源需求高的子任务进行执行加速等. 通过深入研究不同应用场景的需求和特性, 可以制定更精确的优化策略, 进而提升应用的执行性能和效率.

6 局限性

我们讨论本文研究的局限性, 包括特征分析的时效性、平台的服务节点、CPU 密集型任务的线程、内存密集型任务的特征.

(1) 特征分析的时效性: 在本文的研究中, 我们总结了 4 个公有云服务器无感知平台的关键特征, 包括开发、部署和运行时方面特点. 总结的特征结果可能受时效性的影响. 然而, 我们观察到本文总结的特征分析结果受时效性影响较小. 具体来说, 调研的关键特征更新变化相对缓慢, 其结果是基于我们先前工作的结果进行了调整, 而且我们为每个特征提供了更详细的原因分析. 此外, 由分析结果可知, 调研的 4 个平台在每个关键特征上并非都有新的变化. 例如, 亚马逊 Lambda、谷歌 Cloud Functions 和微软 Azure Functions 在超时时长的上限值没有发生变化. 这表明这些平台在特征更新方面相对缓慢. 这可能的原因是这些平台已经发展得相对成熟, 它们的特征和参数相对稳定, 从而表明了本文所总结的结果受时效性影响较小.

(2) 平台的服务节点: 本文研究需要部署同一应用到不同服务器无感知平台的同一区域的节点上. 然而, 不同

平台的服务节点在物理位置上无法保证位于同一个区域, 从而影响性能度量的公平性. 这一局限性是不可避免的, 是因为每个服务器无感知平台都有其专门的服务器集群和服务位置. 因此, 各个平台即使在相同服务位置上, 其节点服务的物理区域可能也存在差异. 为了确保我们对 4 个平台性能比较具有公平性和有效性, 我们在选择节点服务位置时确保了它们在更高粒度的服务区域上保持一致, 比如 us-east-1, 以最大程度地减少实验测试机器到服务节点之间的网络开销影响. 在本文研究中, 我们着重保持了服务位置的一致性, 以尽量减少因物理位置差异而引起的性能差异.

(3) CPU 密集型任务的线程: 在我们微基准测试程序中包含 CPU 密集型任务. CPU 密集型任务可以使用多线程发挥平台的 CPU 潜力. 然而, 在我们实验场下, 我们对该任务没有使用多线程. 原因在于, 本文第 3 节的特征分析揭示, 大多数服务器无感知平台根据配置的内存量按比例分配 CPU 处理能力. 例如, 亚马逊 Lambda 在内存大小约为 1 769 MB 时得到一个 vCPU 的处理能力, 而谷歌 Cloud Functions 和阿里巴巴 Function Compute 分别在内存大小约为 2 048MB 和 1 024 MB 时得到一个 vCPU 的处理能力. 在这种情况下, 即使 CPU 密集型任务使用多线程, 平台也限制了 CPU 处理能力的分配. 因此, 在我们性能实验中探究的内存下, 不能保证各个平台都支持多线程行为. 基于此, 为了保证平台之间比较的公平性, 我们的 CPU 密集型任务关注于单线程的科学计算任务.

(4) 内存密集型任务的特征: 在我们微基准测试程序中包含内存密集型任务, 该任务完成斐波那契数列计算. 在资源方面, 该任务能从内存带宽和内存容量上充分撑满任意大小的资源. 因为该任务的计算时间是根据要计算的数列元素下标而定. 一般来说, 当求解的数列元素下标大时, 在进行内存读取和写入操作需要的内存带宽就高, 中间结果存储以及内存交换需要的内存容量也大. 在栈方面, 当栈过深时, 可能会发生栈溢出. 因为该任务使用递归的方式来实现. 每一个递归调用都会将一些数据压入栈中, 如果栈深度过深, 可能会导致栈溢出, 但最终还是受服务器无感知平台底层提供的函数实例的硬件限制. 为了防止在我们实验过程出现内存带宽和内存容量不够或栈溢出的情况, 我们不使用可能造成这些问题的大的输入数列元素来比较 4 个平台的执行性能.

7 总 结

在本文中, 我们对亚马逊 Lambda、谷歌 Cloud Functions、微软 Azure Functions 和阿里巴巴 Function Compute 这 4 个主流的公有云服务器无感知平台进行了全面可靠的度量研究. 通过特征总结和运行时性能分析, 本文深入研究了这些平台. 在特征总结部分, 本文更新并分析了平台官方文档中描述的开发、部署和运行时的关键特征. 我们发现不同平台在这些特征方面存在差异. 例如, 亚马逊 Lambda、谷歌 Cloud Functions 和阿里巴巴 Function Compute 采用基于分配的内存使用量的付费模型, 而微软 Azure Functions 则是基于消耗的内存使用量计费. 我们还发现了实践中的配置和文档之间存在不一致的现象, 例如亚马逊 Lambda 实际上无法配置超过 3 008 MB 大小的内存. 在运行时性能分析部分, 本文从两个性能角度对平台进行了研究, 即冷启动性能和执行性能. 在冷启动性能中, 本文研究不同编程语言和不同内存大小对平台的影响; 在执行性能中, 本文研究不同任务类型对平台执行性能的影响. 结果表明, 不同的平台对不同语言的冷启动性能表现不一致. 例如, 亚马逊 Lambda 和谷歌 Cloud Functions 执行 Python 应用和 JavaScript 应用比执行 Java 应用获得的冷启动开销更小. 不同平台对不同类型任务的执行能力也存在差异. 例如, 在考虑复杂任务的执行速度时, 基于内存消耗的微软 Azure Functions 提供最快的执行性能. 但对于其他 3 个基于内存分配的平台, 阿里巴巴 Function Compute 可为复杂任务提供更快的执行效率. 基于分析结果, 我们提供了一系列对开发者和云计算厂商具有实际指导意义的启示, 并为研究者提供了潜在的研究机会.

References:

[1] Fouladi S, Wahby RS, Shacklett B, Balasubramaniam KV, Zeng W, Bhalerao R, Sivaraman A, Porter G, Winstein K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In: Proc. of the 14th USENIX Symp. on Networked Systems Design and Implementation. Boston: USENIX Association, 2017. 363–376.

[2] Ao LX, Izhikevich L, Voelker GM, Porter G. Sprocket: A serverless video processing framework. In: Proc. of the 2018 ACM Symp. on Cloud Computing. Carlsbad: ACM, 2018. 263–274. [doi: 10.1145/3267809.3267815]

[3] Wu YC, Dinh TTA, Hu GY, Zhang MH, Chee YM, Ooi BC. Serverless data science-are we there yet? A case study of model serving. In: Proc. of the 2022 Int'l Conf. on Management of Data. Philadelphia: ACM, 2022. 1866–1875. [doi: 10.1145/3514221.3517905]

[4] Jiang JW, Gan SD, Liu Y, Wang FL, Alonso G, Klimovic A, Singla A, Wu WT, Zhang C. Towards demystifying serverless machine learning training. In: Proc. of the 2021 Int'l Conf. on Management of Data. Xi'an: ACM, 2021. 857–871. [doi: 10.1145/3448016.3459240]

[5] Li ZJ, Liu YS, Guo LS, Chen Q, Cheng JG, Zheng WL, Guo MY. FaaSFlow: Enable efficient workflow execution for function-as-a-service. In: Proc. of the 27th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2022. 782–796. [doi: 10.1145/3503222.3507717]

[6] Müller I, Marroquín R, Alonso G. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 115–130. [doi: 10.1145/3318464.3389758]

[7] Zhang H, Tang YP, Khandelwal A, Chen JR, Stoica I. Caerus: NIMBLE task scheduling for serverless analytics. In: Proc. of the 18th USENIX Symp. on Networked Systems Design and Implementation. USENIX Association, 2021. 653–669.

[8] https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

[9] https://cloud.google.com/functions/docs/console-quickstart?hl=zh-cn

[10] https://docs.microsoft.com/en-us/azure/azure-functions/

[11] https://help.aliyun.com/document_detail/52895.html

[12] https://www.gartner.com/smarterwithgartner/the-cios-guide-to-serverless-computing

[13] https://www.researchandmarkets.com/reports/4828585/serverless-architecture-market-by-deployment

[14] Eismann S, Scheuner J, van Eyk E, Schwinger M, Grohmann J, Herbst N, Abad CL, Iosup A. The state of serverless applications: Collection, characterization, and community consensus. IEEE Trans. on Software Engineering, 2022, 48(10): 4152–4166. [doi: 10.1109/TSE.2021.3113940]

[15] https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage

[16] Li JF, Kulkarni SG, Ramakrishnan KK, Li D. Understanding open source serverless platforms: Design considerations and performance. In: Proc. of the 5th Int'l Workshop on Serverless Computing. Davis: ACM, 2019. 37–42. [doi: 10.1145/3366623.3368139]

[17] Mohanty SK, Premsankar G, di Francesco M. An evaluation of open source serverless computing frameworks. In: Proc. of the 2018 IEEE Int'l Conf. on Cloud Computing Technology and Science. Nicosia: IEEE, 2018. 115–120. [doi: 10.1109/CloudCom2018.2018.00033]

[18] https://openwhisk.apache.org/

[19] https://knative.dev/docs/

[20] Wen JF, Liu Y, Chen ZP, Chen JK, Ma Y. Characterizing commodity serverless computing platforms. Journal of Software: Evolution and Process, 2023, 35(10): e2394. [doi: 10.1002/smr.2394]

[21] https://github.com/WenJinfeng/MeasurementStudy

[22] Wang YJ, Sun WD, Zhou S, Pei XQ, Li XY. Key technologies of distributed storage for cloud computing. Ruan Jian Xue Bao/Journal of Software, 2012, 23(4): 962–986 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4175.htm [doi: 10.3724/SP.J.1001.2012.04175]

[23] Song J, Li TT, Yan ZX, Na J, Zhu ZL. Energy-efficiency model and measuring approach for cloud computing. Ruan Jian Xue Bao/Journal of Software, 2012, 23(2): 200–214 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4144.htm [doi: 10.3724/SP.J.1001.2012.04144]

[24] Yu TY, Liu QY, Du D, Xia YB, Zang BY, Lu ZQ, Yang PC, Qin CG, Chen HB. Characterizing serverless platforms with serverlessbench. In: Proc. of the 11th ACM Symp. on Cloud Computing. ACM, 2020. 30–44. [doi: 10.1145/3419111.3421280]

[25] Maissen P, Felber P, Kropf P, Schiavoni V. FaaSdom: A benchmark suite for serverless computing. In: Proc. of the 14th ACM Int'l Conf. on Distributed and Event-based Systems. Montreal: ACM, 2020. 73–84. [doi: 10.1145/3401025.3401738]

[26] Kim J, Lee K. FunctionBench: A suite of workloads for serverless cloud function service. In: Proc. of the 12th IEEE Int'l Conf. on Cloud Computing. Milan: IEEE, 2019. 502–504. [doi: 10.1109/CLOUD.2019.00091]

[27] Shahrad M, Fonseca R, Goiri Í, Chaudhry G, Batum P, Cooke J, Laureano E, Tresness C, Russinovich M, Bianchini R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In: Proc. of the 2018 USENIX Annual Technical Conf. USENIX Association, 2020. 14.

[28] Zhang M, Zhu YF, Zhang C, Liu JC. Video processing with serverless computing: A measurement study. In: Proc. of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video. Amherst: ACM, 2019. 61–66. [doi: 10.1145/3304112.3325608]

[29] Wang L, Li MY, Zhang YQ, Ristenpart T, Swift M. Peeking behind the curtains of serverless platforms. In: Proc. of the 2018 USENIX

Annual Technical Conf. Boston: USENIX Association, 2018. 133–145.
[30] McGrath G, Brenner PR. Serverless computing: Design, implementation, and performance. In: Proc. of the 37th IEEE Int'l Conf. on Distributed Computing Systems Workshops. Atlanta: IEEE, 2017. 405–410. [doi: 10.1109/ICDCSW.2017.36]
[31] Palade A, Kazmi A, Clarke S. An evaluation of open source serverless computing frameworks support at the edge. In: Proc. of the 2019 IEEE World Congress on Services. Milan: IEEE, 2019. 206–211. [doi: 10.1109/SERVICES.2019.00057]
[32] Lloyd W, Ramesh S, Chinthalapati S, Ly L, Pallickara S. Serverless computing: An investigation of factors influencing microservice performance. In: Proc. of the 2018 IEEE Int'l Conf. on Cloud Engineering. Orlando: IEEE, 2018. 159–169. [doi: 10.1109/IC2E.2018.00039]
[33] Ustiugov D, Amariucai T, Grot B. Analyzing tail latency in serverless clouds with STeLLAR. In: Proc. of the 2021 IEEE Int'l Symp. on Workload Characterization. Storrs: IEEE, 2021. 51–62. [doi: 10.1109/IISWC53511.2021.00016]
[34] https://github.com/ddps-lab/serverless-faas-workbench/tree/master/aws/cpu-memory/linpack
[35] https://github.com/epsagon/lambda-memory-performance-benchmark/tree/master/fibonacci-function
[36] https://github.com/ddps-lab/serverless-faas-workbench/tree/master/aws/disk/gzip_compression
[37] https://github.com/ddps-lab/serverless-faas-workbench/tree/master/aws/network/json_dumps_loads
[38] https://github.com/Molecule-Serverless/serverless-faas-workbench/tree/296f5f2709b3e7b168db00473d0a28b8bf1c062f/aws/cpu-memory/image_processing
[39] https://github.com/Iam-ahsan/MBS/tree/main/lambda_package
[40] https://github.com/spcl/serverless-benchmarks/tree/master/benchmarks/500.scientific/501.graph-pagerank
[41] https://github.com/ddps-lab/serverless-faas-workbench/tree/master/aws/cpu-memory/model_training
[42] https://github.com/ddps-lab/serverless-faas-workbench/tree/master/aws/cpu-memory/model_serving/ml_lr_prediction
[43] Sreekanti V, Wu CG, Chhatrapati S, Gonzalez JE, Hellerstein JM, Faleiro JM. A fault-tolerance shim for serverless computing. In: Proc. of the 15th European Conf. on Computer Systems. Heraklion: ACM, 2020. 15. [doi: 10.1145/3342195.3387535]
[44] Liu XZ, Wen JF, Chen ZP, Li D, Chen JK, Liu Y, Wang HY, Jin X. FaaSLight: General application-level cold-start latency optimization for function-as-a-service in serverless computing. ACM Trans. on Software Engineering and Methodology, 2023, 32(5): 119. [doi: 10.1145/3585007]
[45] Kelly D, Glavin F, Barrett E. Serverless computing: Behind the scenes of major platforms. In: Proc. of the 13th IEEE Int'l Conf. on Cloud Computing. Beijing: IEEE, 2020. 304–312. [doi: 10.1109/CLOUD49709.2020.00050]
[46] https://www.serverless.com/
[47] Mathew A, Andrikopoulos V, Blaauw FJ. Exploring the cost and performance benefits of AWS step functions using a data processing pipeline. In: Proc. of the 14th IEEE/ACM Int'l Conf. on Utility and Cloud Computing. Leicester: ACM, 2021. 7. [doi: 10.1145/3468737.3494084]
[48] Wen JF, Liu Y. A measurement study on serverless workflow services. In: Proc. of the 2021 IEEE Int'l Conf. on Web Services. Chicago: IEEE, 2021. 741–750. [doi: 10.1109/ICWS53863.2021.00102]
[49] Burckhardt S, Gillum C, Justo D, Kallas K, McMahon C, Meiklejohn CS. Durable functions: Semantics for stateful serverless. Proc. of the ACM on Programming Languages, 2021, 5(OOPSLA): 133. [doi: 10.1145/3485510]
[50] Grogan J, Mulready C, McDermott J, Urbanavicius M, Yilmaz M, Abgaz Y, McCarren A, MacMahon ST, Garousi V, Elger P, Clarke P. A multivocal literature review of function-as-a-service (FaaS) infrastructures and implications for software developers. In: Proc. of the 27th European Conf. on Systems, Software and Services Process Improvement. Düsseldorf: Springer, 2020. 58–75. [doi: 10.1007/978-3-030-56441-4_5]
[51] https://www.alibabacloud.com/help/zh/fc/instance-types-and-instance-modes?spm=a2c63.p38356.0.0.78617a0fHIpikP
[52] https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html
[53] Wen ZJ, Wang YS, Liu FM. StepConf: SLO-Aware dynamic resource configuration for serverless function workflows. In: Proc. of the IEEE Conf. on Computer Communications. London: IEEE, 2022. 1868–1877. [doi: 10.1109/INFOCOM48880.2022.9796962]

附中文参考文献:

[22] 王意洁, 孙伟东, 周松, 裴晓强, 李小勇. 云计算环境下的分布存储关键技术. 软件学报, 2012, 23(4): 962–986. http://www.jos.org.cn/1000-9825/4175.htm [doi: 10.3724/SP.J.1001.2012.04175]
[23] 宋杰, 李甜甜, 闫振兴, 那俊, 朱志良. 一种云计算环境下的能效模型和度量方法. 软件学报, 2012, 23(2): 200–214. http://www.jos.org.cn/1000-9825/4144.htm [doi: 10.3724/SP.J.1001.2012.04144]

温金凤(1994－), 女, 博士, 主要研究领域为服务器无感知计算, 系统软件.

柳熠(1993－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为服务计算, 移动计算, 数字对象架构.

陈震鹏(1994－), 男, 博士, 主要研究领域为软件解析学.

刘譞哲(1980－), 男, 博士, 副教授, 博士生导师, CCF 杰出会员, 主要研究领域为服务计算, 系统软件.