软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(1):47−78 [doi: 10.13328/j.cnki.jos.007190] [CSTR: 32375.14.jos.007190]

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

服务器无感知计算系统性能优化技术研究综述*

杨 光[1,2], 刘 杰[1,3], 曲慕子[1,2], 王 帅[2], 叶 丹[1,3], 钟 华[1,2]

[1](中国科学院大学, 北京 100049)

[2](中国科学院 软件研究所 软件工程技术研究开发中心, 北京 100190)

[3](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

通信作者: 王帅, E-mail: wangshuai@otcaix.iscas.ac.cn

摘 要: 服务器无感知计算是新兴的云计算模式, 它基于“函数即服务 (FaaS)”的范式, 以函数为部署和调度的基本单位, 为用户提供大规模并行和自动伸缩的函数执行服务, 且无需用户管理底层资源. 对于用户, 服务器无感知计算能够帮助他们摆脱集群底层基础设施管理的负担, 专注于业务层的开发和创新; 对于服务提供商, 服务器无感知计算将应用分解为细粒度的函数, 极大地提高了调度效率和资源利用率. 显著的优势让服务器无感知计算迅速吸引了业界的注意, 然而, 服务器无感知计算与传统云计算迥然不同的计算模式以及对任务各方面的严格限制给应用的迁移带来了诸多障碍, 各种越来越复杂的任务也对服务器无感知计算的性能提出了越来越高的要求, 服务器无感知计算的性能优化成为一个重要的研究课题. 从 4 个方面对服务器无感知计算系统性能优化技术的相关研究工作进行梳理和综述, 并介绍现有的系统实现. (1) 介绍面向典型任务的优化技术, 包括任务适配和针对特定任务的系统优化; (2) 综述沙箱环境的优化工作, 包括沙箱方案和冷启动优化技术, 它们是决定函数运行速度的核心; (3) 概括 I/O 和通信技术的优化, 它们是服务器无感知计算应用程序的主要性能瓶颈; (4) 简述相关的资源调度技术, 包括面向平台和面向用户的调度策略, 它们决定着系统的资源利用率和任务的执行效率. 最后, 总结当前服务器无感知计算性能优化技术所面临的问题和挑战, 并展望未来可能的发展方向.

关键词: 服务器无感知计算; 函数即服务; 云函数; 云计算; 性能优化

中图法分类号: TP311

中文引用格式: 杨光, 刘杰, 曲慕子, 王帅, 叶丹, 钟华. 服务器无感知计算系统性能优化技术研究综述. 软件学报, 2025, 36(1): 47–78. http://www.jos.org.cn/1000-9825/7190.htm

英文引用格式: Yang G, Liu J, Qu MZ, Wang S, Ye D, Zhong H. Review on Performance Optimization Technology in Serverless Computing System. Ruan Jian Xue Bao/Journal of Software, 2025, 36(1): 47–78 (in Chinese). http://www.jos.org.cn/1000-9825/7190.htm

Review on Performance Optimization Technology in Serverless Computing System

YANG Guang[1,2], LIU Jie[1,3], QU Mu-Zi[1,2], WANG Shuai[2], YE Dan[1,3], ZHONG Hua[1,2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)

[2](Software Engineering Technology Research and Development Center, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[3](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

Abstract: Serverless computing is an emerging cloud computing model based on the “function as a service (FaaS)” paradigm. Functions serve as the fundamental unit for deployment and scheduling, providing users with massively parallel and automatically scalable function

* 基金项目: 国家自然科学基金 (61972386)
收稿时间: 2023-12-31; 修改时间: 2024-02-07; 采用时间: 2024-03-25; jos 在线出版时间: 2024-06-14
CNKI 网络首发时间: 2024-06-19

execution services without the need to manage underlying resources. For users, serverless computing helps them alleviate the burden of managing cluster-level infrastructure, enabling them to focus on business-layer development and innovation. For service providers, applications are decomposed into fine-grained functions, leading to significantly improved scheduling efficiency and resource utilization. The significant advantages have swiftly drawn the attention from the industry and propelled serverless computing into popularity. However, the distinct computing mode of serverless computing, divergent from traditional cloud computing, along with its stringent limitations on various aspects of tasks, poses numerous obstacles to application migration. The escalating complexity of migrated tasks also imposes higher performance requirements on serverless computing. Therefore, performance optimization technology for serverless computing systems has emerged as a critical research topic. This study reviews and summarizes research efforts on performance optimization of serverless computing from four perspectives, and introduces existing system. Firstly, this study introduces the optimization technologies for typical tasks, including task adaptation and system optimization for specific task types. Secondly, it reviews the optimization work on sandbox environments, encompassing sandbox solutions and cold start optimization methods, which play a crucial role in the execution of serverless functions. Thirdly, it provides an overview of the optimization in I/O and communication technologies, which are major performance bottlenecks of serverless applications. Lastly, it briefly outlines related resource scheduling technologies, including platform-oriented and user-oriented scheduling strategies, which determine system resource utilization and task execution efficiency. In conclusion, it summarizes the current issues and challenges of performance optimization technologies of serverless computing and anticipates potential future research directions.

Key words: Serverless computing; function as a service (FaaS); cloud function; cloud computing; performance optimization

在过去的十几年, 云计算产业快速发展, 越来越多的企业将业务迁移到云平台, 云计算已经成为现代互联网的重要基础设施. 而云服务商也不断发展迭代技术体系, 从早期的 IaaS 托管基础设施, 到后来的 PaaS 把操作系统平台作为一种服务提供, 到 SaaS 直接提供应用软件服务, 云计算服务的综合性能不断提升, 管理难度逐步降低, 越来越简化的抽象层替代了庞大复杂的底层硬件集群, 协助用户管理云端的计算和存储能力, 云计算正向着更高虚拟化、更高抽象化、更低维护难度的目标不断前行探索. 服务器无感知计算 (Serverless computing) 就是顺应这个潮流诞生的一种新的云计算服务模式. 服务器无感知计算也翻译为“无服务器计算”, 本文使用 2023 年 CCF 发布的“服务器无感知计算”术语 (https://www.ccf.org.cn/Media_list/gzwyh/jsjsysdwyh/2023-03-17/789780.shtml), 下文中也会简写为“Serverless”.

2014 年, 亚马逊 (Amazon Inc.) 上线了全新的云计算服务 AWS Lambda[1], 正式提出了服务器无感知计算的概念. 服务器无感知计算提供了一种新型的云计算模式, 它无需用户理会底层基础设施的任何管理细节, 无需部署、管理或调度, 为用户提供了一种似乎“没有服务器存在”的体验. 与传统的 IaaS 和 PaaS 相比, 服务器无感知计算基于“函数即服务 (FaaS)”的范式, FaaS 提供给用户管理的抽象层次更高, 仅次于 SaaS, 如图 1 所示. FaaS 由用户管理的只有函数, 基于事件驱动的编程模型, 用户只需提供函数代码, 绑定触发函数执行的事件 (例如收到了 HTTP 请求、云存储中被加入了新的文件, 或者数据库表中被加入了新的条目等), 服务器无感知系统会自动管理其余一切事务, 如代码部署、执行、资源分配、弹性伸缩、监控、日志、安全等方面. 并且, 服务器无感知计算提供了更精细的计费方式, 用户只需为自己实际使用的资源付费. 与传统的“有服务器计算”相比, 服务器无感知计算的特点主要有以下几点.

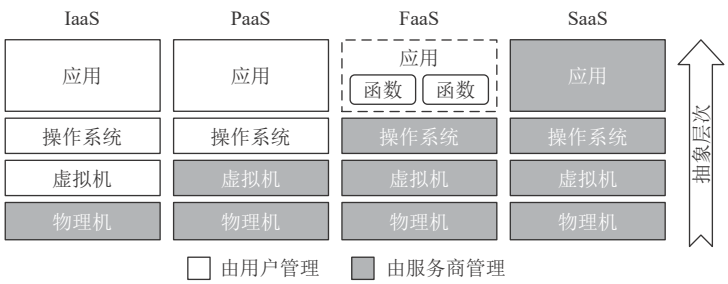

图 1 几种云服务模式的区别比较

(1) 函数无状态. 在服务器无感知模型中, 基本的调度单元函数是短寿命且无状态的, 它不支持长期存储状态信息, 需要持久存储的信息需要保存到独立的外部存储服务中. 这种人为的限制虽然制约了业务开发的能力, 但大大便利了服务的调度和伸缩.

(2) 无需管理底层资源. 服务器无感知计算几乎帮用户摆脱掉了所有底层资源配置和调度管理的负担, 用户仅需提供代码, 简单的设置函数资源分配, 即可实现高效的应用程序.

(3) 按使用付费而不是按资源分配付费. 函数只有在被执行的时候计费, 其具体计费规则只和执行相关, 如执行时间、使用硬件性能等. AWS Lambda 的计费时间精度已经达到了 1 ms, 这种超细粒度的计费方式无疑可以为用户节省很多成本.

对于用户, 服务器无感知计算能够帮助他们摆脱服务器集群等底层基础设施管理的负担, 专注于业务层的开发和创新; 对于服务提供商, 服务器无感知计算将应用分解为细粒度的函数, 函数成为部署和调度的基本单位, 极大地提高了资源利用率. 显著的优势让服务器无感知计算迅速流行起来, 继 AWS Lambda[1]之后, 微软[2]、谷歌[3]、IBM[4]都推出了各自的服务器无感知计算服务, 国内的阿里云[5], 百度云[6]也推出了函数计算服务, 在开源社区, 也有了 OpenWhisk[7]、OpenLambda[8]、OpenFaaS[9]、Fission[10]、Knative[11]、Kubeless[12](已停止维护) 等较为成熟的服务器无感知计算系统.

在应用方面, 服务器无感知计算最初的设计目标是针对偶发性的多线程 Web 服务, 例如图片压缩服务等, 主流的应用类型也一直都是 Web 服务 (占比 32%, 2018 年统计[13]). 此类任务中, 服务器无感知计算体现出了显著的优势, Villamizar[14]综合评估了 AWS Lambda、单体机器和微服务架构在各种运行 Web 应用时的底层服务成本, AWS Lambda 是最低的. 此后, 服务器无感知计算的优势吸引了越来越多的其他类型应用迁移到服务器无感知平台, 如大数据分析、机器学习、视频处理、代码编译、科学计算等, 服务器无感知计算的应用范围越来越广, 成为最具潜力的云计算模式之一. 加州大学伯克利分校的报告[15]认为, 未来服务器无感知架构将取代现有的以容器和虚拟机为基础的架构成为下一代云计算环境开发和运行的主流架构. 科学家们认为服务器无感知计算提出的简化编程计算模型, 可以类比 MapReduce 对传统并行编程的简化, 具有革命性的意义.

随着迁移到服务器无感知计算的应用类型越来越多, 各种越来越复杂的任务对服务器无感知计算的性能提出了越来越高的要求, 服务器无感知计算的性能优化成为一个重要的研究课题, 也是本文所关注的问题. 许多文献已经在服务器无感知计算领域做出了综述报告, 除最早的加州大学伯克利分校的报告[15]之外, 文献 [16,17] 也总结了服务器无感知计算的发展情况, 提出了许多开放性的问题, 不过这些文献时间较早, 已经无法涵盖最新的研究工作. 文献 [18–20] 的综述涵盖了较新的研究, 不过他们着眼于对整个服务器无感知计算技术进行总结. 文献 [21–25] 则是针对特定技术方向进行总结. 纵观所有文献, 当前还没有聚焦于服务器无感知计算系统内的性能优化技术进行总结分析的工作. 因此, 本文将对服务器无感知计算系统性能优化技术的相关研究工作进行全面深入的综述, 总结服务器无感知计算性能优化所面临的主要问题和现有的解决方案, 并介绍各种相关系统的实现.

为获取本文综述的研究文献, 我们使用 Google Scholar 搜索引擎, 以 Serverless、FaaS、cloud function、function as a service 关键词检索文献, 筛选出 CCF 推荐的 B 类或更高级别的刊物和会议论文, 和《中国科学院文献情报中心期刊分区表》中二区及以上的期刊文献, 以及一些高引用率的 arXiv 论文, 再从中删去与服务器无感知计算性能优化无关的文献, 作为本文综述的基础. 本文中除一些综述报告外, 概括的系统和技术基本都来自以上筛选过程, 最后筛选出的文献主要集中于 OSDI、NSDI、USENIX ATC、SOCC、SIGMOD、INFOCOM、EuroSys 等知名会议.

图 2 统计了我们收集到的历年研究论文数量 (2023 年的文章尚未全部见刊, 故数量略少), 从中可以大致看出, 服务器无感知计算的研究热度逐年升高. 其中, 美国加州大学伯克利分校在该领域一直较为领先, 此外, 加州大学圣地亚哥分校、威斯康辛大学、波士顿大学、斯坦福大学、乔治梅森大学等也有相关研究. 企业界则以亚马逊、微软和 IBM 为主, 他们都有自己的服务器无感知计算业务. 国内的研究则是以上海交通大学、天津大学居多, 北京大学也有相关研究, 企业界则是以拥有服务器无感知计算业务的阿里云为主.

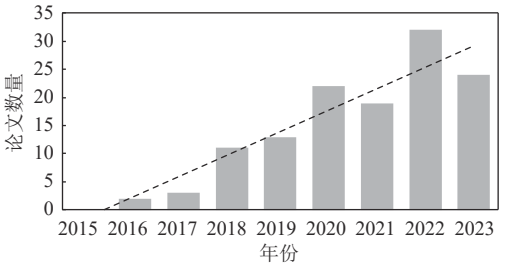

图 2 历年研究论文数量统计

1 服务器无感知计算系统性能优化技术

一般而言, 服务器无感知计算的性能问题主要出现在以下几个环节, 本文将基于这些性能瓶颈对现有研究工作进行分类总结.

(1) 任务适配. 服务器无感知计算特有的细粒度函数计算的模式, 让传统的计算任务在迁移时通常都需要重构程序, 用新的编程模型实现需求的功能. 这是一项颇具挑战的工作, 不同类型的任务有不同的计算流程, 在移植到服务器无感知环境的时候总是会遇到各种不同的障碍, 通常都需要对原有的功能进行取舍, 并且新的结构设计将直接影响程序的功能和性能. 此外, 各种不同类型的任务也给服务器无感知平台的适应性和性能提出了更多的挑战. 本文将在第 2 节概述当前研究对各种典型计算任务的适配和针对性系统优化工作.

(2) 沙箱环境. 相比传统的计算模式, 服务器无感知计算执行的函数虽然更为轻量, 但需要大量并发和频繁启停, 高性能的沙箱隔离环境是函数执行的必要基础. 常见的隔离方案通常以虚拟机或容器为基础, 但传统的虚拟机或容器并非为这种轻量高并发的模式设计, 在服务器无感知场景下, 这些虚拟化技术仍有很大的优化空间. 另一方面, Serverless 函数频繁启停的模式会放大沙箱的冷启动问题, 如果不做任何优化, 冷启动延迟会大大拖慢程序的运行速度. 本文将在第 3 节概述当前沙箱环境优化的相关研究工作.

(3) I/O 和通信. 由于 Serverless 函数是无状态的, 所有需要持久保存的数据都需要保存到外部存储服务中, 函数的生命时间限制更加大了来回存取操作的频率, 大量的存取操作会显著加大网络开销, 并在外部存储服务上产生更多的资源开销和费用. 同时, 服务器无感知平台的函数带宽通常并不大, 如 AWS Lambda 的函数带宽约为 60 MB/s, 这使得数据 I/O 成为服务器无感知应用程序中最常见的性能瓶颈. 另一方面, 服务器无感知系统通常都不允许函数间直接通信, 这给程序开发带来了很多限制, 制约了很多程序的性能. 因此, 对 I/O 和通信的优化是服务器无感知计算性能优化的一类重要工作, 本文将在第 4 节概述这方面的研究工作.

(4) 资源调度. 针对传统集群的资源调度技术已经较为成熟, 但服务器无感知计算带来了新的任务和负载模式, 传统的调度技术和思想有必要对服务器无感知计算进行重新优化. 此外, 服务器无感知计算的计费模式虽然精细, 但费率通常不低, 如果用户层没有合理的安排计算流程和资源分配, 很容易造成效率低下和成本浪费. 因此, 面向平台和面向用户的资源调度优化对服务器无感知系统的性能提升有重要的价值, 本文将在第 5 节概述服务器无感知计算资源调度方面的研究工作.

2 面向典型任务的优化

服务器无感知计算特殊的范式始终是传统任务移植的最大障碍, 将 Web 服务以外的传统任务迁移到服务器无感知环境需要重构代码, 将原有的程序结构改写为相互关联的函数组合, 同时还必须考虑到 Serverless 函数受限的资源分配, 研究者需要考虑任务分割、数据分发与传递、函数协调、容错等问题. 因此, 当前的研究一方面在应用程序层针对各种任务提出新的结构, 以便在服务器无感知环境下运行, 另一方面在平台层对服务器无感知框架进行各种针对性优化, 以适应各种不同类型的处理任务, 扩展服务器无感知计算的应用范围, 本节将综述这些系统的贡献.

2.1 大数据分析任务

大数据分析任务是除 Web 服务任务外最早迁移到服务器无感知平台的任务类型之一, 相关研究主要集中于任务适配方面, 包括 MapReduce、Spark 和科学计算等任务类型的迁移.

加州大学伯克利分校开发的 Pywren[26]是将服务器无感知计算应用于大数据分析的先行者. Pywren 是一个基于 AWS Lambda 的分布式计算系统 (原型), 它提供了类 MapReduce 的编程接口, 为用户屏蔽了服务器无感知计算的事件编程模型. Pywren 将函数代码和数据序列化传输, 自动处理了函数的上传、调用、执行和返回操作, 实现了犹如本地计算一样的 map 函数. 此后 OpenWhisk 阵营的 IBM 推出了类似的 IBM-Pywren[27], 运行在 IBM Cloud Functions[4]上, 代码和数据保存在 IBM 的对象存储服务 Cloud Object Storage 中, 它的执行流程与 Pywren 相似, 但做了许多功能上的扩展, 如支持更完整的 MapReduce 操作, 包括 Reduce 方法和类似 reduceByKey 的操作方式, 它支持用户自定义分区大小的数据分区方式和一定的函数复合特性. 由于基于 OpenWhisk 平台, IBM-Pywren 使用 Docker 管理容器, 也就可以更灵活的支持不同的 Python 运行时环境.

Numpywren[28]是 Pywren 团队的后续研究成果, 它使用 Pywren 分发 Serverless 函数任务, 实现了服务器无感知架构下的线性代数运算框架. Numpywren 针对超大型矩阵计算, 将大型矩阵运算拆解为小块矩阵运算的叠加, 并定义了自己的语法体系 LAmbdaPACK 来描述矩阵块的运算流程. 实际运行中, Numpywren 通过对 LAmbdaPACK 代码的静态分析, 得到程序的 DAG 图, 以协调 Lambda 函数的运行关系. Numpywren 设计了图 3 的系统架构. 其中 DAG 保存于 Redis 数据库, 协调器 Provisioner 负责启动 Lambda 函数, Lambda 函数更新 DAG 图状态并维护任务队列, 外部存储使用 AWS S3. 该架构也是典型的服务器无感知环境中执行大数据任务的架构, 通过外部对象存储保存大数据, 高速数据库服务维护任务状态, 同时设置任务队列, Serverless 函数作为执行器, 有时还需要引入虚拟机来负责一些高负载或持久化的工作. 论文中发现, Numpywren 的主要性能瓶颈在于网络通信, 这也是服务器无感知计算中的一个典型问题和优化方向.

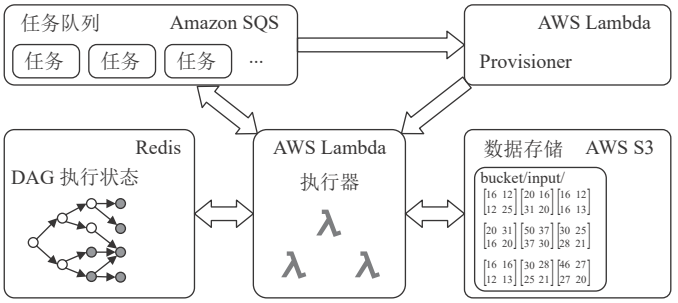

图 3 Numpywren 的系统结构

作为大数据分析领域的经典模型, 许多研究都致力于移植 MapReduce 模型到服务器无感知计算平台. MARLA[29]提供了一个完整的 MapReduce 解决方案, 它将数据保存到 S3 存储, 调用 Lambda 函数并行执行 Map 和 Reduce 操作, 系统的执行流程可以简化为图 4 所示. 这是一个典型的服务器无感知平台大数据分析流程, 将待处理数据分区后保存于 S3, 由 Lambda 函数来对每个分区的数据执行计算任务, 通过两次任务分发和聚合, 分别完成 Map 和 Reduce 操作. 此外, 也有一些开源的 MapReduce 实现, 如 ooso[30]、Corral[31]等.

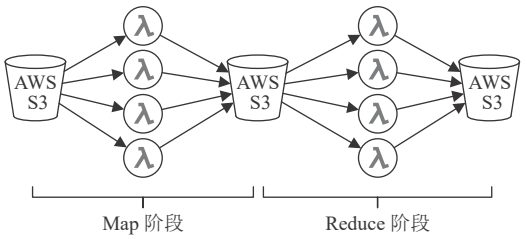

图 4 MARLA 的 MapReduce 模型执行流程

Lambada[32]是一个服务器无感知环境下的数据分析框架, 主要着重于交互式的数据查询. 它使用 S3 来存储大块数据和交换数据, AWS DynamoDB 来存储小块数据以便快速读写. Lambada 会调用 Serverless 函数来分布式处理查询任务, 并自动处理了任务分发和结果合并等中间细节, 实现了便利而经济的数据查询服务. Starling[33]定位于数据库检索引擎, 它将数据表存储于 S3, 使用 VM 上的协调器编译检索请求, 并将其转换为多个函数任务交给 AWS Lambda 执行. 对于检索执行的过程中的 Shuffle 操作, Starling 使用 S3 作为数据交换空间, 为了解决 Shuffle 操作读写过于频繁导致 S3 费用暴增的问题, Starling 使用了分阶段 Shuffle 方案, 将多次小块数据读写合并为一次大块数据读写, 从而降低读写频率.

Pheromone[34]打破了服务器无感知计算以函数为中心的设计理念, 提出以数据为中心设计整个平台. Pheromone 开发接口的核心是一个“数据桶 (Data bucket)”抽象, 用于存储和交换数据, 并提供了一系列基于数据的触发原语 (trigger primitive) 来启动函数并控制函数的输出. 这种结构类似于 AWS Lambda+S3 的组合, 但拥有更丰富灵活的触发方式, 使其可以基于数据构建和驱动工作流程, 更高效地实现复杂的函数交互. 这种模式为服务器无感知计算下的大数据处理提供了新的思路.

许多大数据分析任务都是遵循 DAG 控制流执行的计算任务, 即控制流可以描述为有向无环图的程序结构. WuKong[35,36]是建立在 AWS Lambda 上的 DAG 任务执行系统, 它设计了一套完整的 DAG 执行方案, 包括一个调度器用于监控整个 DAG 的执行过程, 使用外部 Redis 数据库来保存中间和最终结果. WuKong 从叶子节点向根节点执行 DAG, 每个节点使用一个 Lambda 函数执行, 同时尽量重复使用正在运行的 Lambda 函数, 让一个 Lambda 函数尽可能多的执行任务, 并且在扇入扇出节点上设计了等待方案. 这套方案显著减少了函数输入输出数据向外部存储的写入和取回. 不过由于单个 Lambda 函数的性能有限, 需要先小心的进行任务切分, 以保证每个节点都可以在单个 Lambda 函数内执行完成. Cloudburst[37]是一个独立的针对 DAG 任务设计的服务器无感知计算系统, 它的函数由 VM 内的进程并行执行, Cloudburst 在每个 VM 内设置了本地缓存, 该缓存服务对函数代码是完全透明的, 缓存将自动保存最近访问的外部存储 (Cloudburst 使用的是 Anna, 同样由加州大学伯克利分校推出的高效键值存储系统) 中的对象的副本, 并且该副本将自动与原对象保持同步. Cloudburst 还允许函数之间通过 TCP 连接直接进行点对点通信, 这种直接通信要比通过外部存储中转要高效得多. 便捷的通信和共享机制不但提高了 DAG 任务的执行效率, 也扩展了服务器无感知计算的概念.

2.2 机器学习任务

机器学习任务是 I/O 和 CPU 都密集的任务类型, 移植到服务器无感知平台面临的挑战很大, 但由于其应用广泛, 也是移植最热门的任务类型. 机器学习任务分为训练和推理两种, 分别代表了机器学习流程的两个阶段, 其中训练通常比推理更加复杂. 一般而言, 在服务器无感知平台上进行机器学习训练, 频繁的通信会导致性能严重下降, 而有限的函数执行时间会限制复杂模型的应用. 而且, 鉴于服务器无感知计算的费率, 即使基于服务器无感知计算的训练会更快, 也不一定会更便宜. 因此, 相关的研究主要聚焦于如何设计系统结构、减少通信开销和控制计算成本. 机器学习推理任务则相对简单, 相关研究主要集中于批量调度, 成本控制和更大规模模型的应用.

2.2.1 训练任务

在各种分布式机器学习模型中, 参数服务器模型应用最为广泛, 该模型中的工作节点之间并不需要直接通信, 天然地适合无法进行点对点通信的服务器无感知平台, 因此也是服务器无感知机器学习系统中最常见的结构. 在服务器无感知平台上实现的参数服务器模型, 通常可以简化表示为图 5 所示的结构, 训练集被预先切分好并保存于存储服务中, 由一个个 Serverless 函数担任工作节点, 每个节点负责一块或多块数据的训练迭代, 并将计算出的更新数据 (梯度) 汇总到参数服务器, 这里的参数服务器可以是一个函数, 或虚拟机, 或将模型直接保存于存储服务. 由于服务器无感知平台会提供均匀稳定的调度服务, 所以通常不需要独立的调度器模块. 当前, 几个较为完善的服务器无感知机器学习框架都是基于参数服务器模型或它的变体.

Cirrus[38,39]服务器无感知机器学习的先行者. Cirrus 基于参数服务器模型, 其中参数服务器是 EC2 虚拟机, Worker 为 Lambda 函数. Cirrus 实现了异步 SDG、AdaGrad 和 Momentum 优化算法, 机器学习算法包括逻辑回归和协同过滤算法, 提供了机器学习任务从数据预处理到训练到超参数调节的一整套功能, 并通过 API 提供给用户.

实验表明 Cirrus 相对 Spark、TensorFlow 和 Bosen 都表现出了更快的收敛速度, 并展现出了理想的伸缩性, 证明了服务器无感知平台上的机器学习可以得到比传统集群更高的性能.

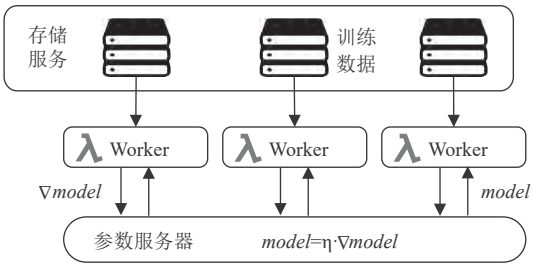

图 5 简化的参数服务器模型

SMLT[40]也基于参数服务器模型, 它拥有自适应的资源调度系统, 该系统基于贝叶斯优化去寻找最佳的函数资源分配方案, 能够根据工作负载的实时变化动态调整资源配置. Siren[41]改进了参数服务器模型, 取消了参数服务器节点, 将所有模型参数都保存于外部存储 S3, 所有的 Worker (函数) 直接向 S3 写入参数、更新模型, 并拉取新的模型参数. 这种结构将模型更新任务分摊给函数, 但容易在 S3 上产生写入排队问题. 在资源调度方面, Siren 使用深度强化学习算法来决定函数的内存大小和并行数量, 并设计了“混合同步模式”, 在 Epoch 之间插入调度动作, 从而在任务执行过程中动态调整资源分配, 兼顾了性能和成本. 与 Cirrus 不同, Siren 使用了精简版的 MXNET 框架 (以便能放入函数内), 可以更方便地使用现有的机器学习代码.

MLLess[42,43]的训练模型与 Siren 相似, 也取消了参数服务器节点, 不过它将模型以副本的形式保存在每个函数之内, 外部存储只负责交换梯度数据. 每个函数到外部存储上传和拉取梯度数据, 维护自己本地的模型副本更新. 为了提高训练的效率, MLLess 做了两方面的优化, 一方面是函数在上传更新数据时会过滤掉“非显著”的更新数据, 将其留在本地累积而不上传. 这种策略可以减少通信开销, 但可能会影响到模型收敛速度, 所以 MLLess 允许用户根据实际情况调整“显著”阈值的严格程度. 另一方面, MLLess 根据任务运行情况预估剩余的时间和成本, 在不影响性能的情况下, 在合适的时机缩减节点数量, 为用户节约计算成本.

λDNN[44]基于参数服务器模型实现了 DNN (深度神经网络) 的训练, 并实现了一个针对 DNN 的性能预测模型. 该模型基于平台和任务的各种参数指标, 包括网络和 I/O 带宽、函数并行数量、内存大小、任务训练集大小、模型参数大小等, 可以准确 (误差 0.98%–6.0%) 的预测任务的完成时间. 基于该性能模型的预测, λDNN 能够根据用户设置的目标训练时间选择最合适、最经济的函数资源配置方案, 实现性能和成本的平衡.

Dorylus[45]同时使用传统服务器集群和服务器无感知计算实现 GNN (图神经网络) 的高效并行训练. 它将 GNN 训练任务分为两种任务类型, 图计算任务如 Gather、Scatter 等, 交由传统 CPU 集群执行, 张量计算任务如梯度计算等线性代数计算, 交由动态扩充的 Serverless 函数负责, 从而同时利用两种集群的优点.

2.2.2 推理任务

相比机器学习训练任务, 推理任务通常计算量小得多, 但往往请求量很大, 服务器无感知平台快速伸缩的特性有助于推理服务的高效响应. 如 ML-FaaS[46]使用 Serverless 函数来执行机器学习推理任务 (它也可以训练, 不过训练是在本地进行的), 实现了高效的执行和灵活可扩展的工作流程组合. 不过在大量请求之下, 推理函数的反复启动、模型的反复加载会导致显著的算力和时间浪费, 在商业服务器无感知平台内还会导致成本过高. 对此, Batch[47]提出了任务批处理模式 (Batching), 通过将推理任务分组、批量执行, 减少函数启动和模型加载次数, 有效地降低了成本. INFless[48]也使用了内置的批处理和非均匀扩展机制实现高吞吐量, 并可以根据推理模型的资源需求 SLO 动态选择不同的批处理大小和资源配置. 另外, INFless 还可以通过 NVIDIA-docker 提供的容器访问接口来访问和管理 GPU 资源, 并提供统一的资源抽象.

Gillis[49]聚焦于大型神经网络模型在服务器无感知环境下的推理计算. 它的基本思想是将大模型切分, 分散到多个 Serverless 函数中并行执行. 但推理过程中模型的分区之间会产生大量的数据通信, 为解决这个问题, Gillis 使

用了粗粒度分区, 将模型的多个连续的层划分为“层组 (layer group)”, 一个函数负责一个层组内的所有层, 只有跨层组的通信才需要跨函数通信, 降低了通信频率. 进一步的, 为了确定层的分组策略, Gillis 使用了强化学习算法, 以推理成本和时间作为奖励函数进行训练, 最后使得系统可以自动选择合适的层分组策略, 实现高效的训练. AMPS-Inf[50]的思路与 Gillis 类似, 都是通过按层切分模型, 不过 AMPS-Inf 的层切分更为细粒度一些. AMPS-Inf 使用最优化方法来寻找最佳的模型切分策略, 由于神经网络模型的推理运行性能通常较为稳定, AMPS-Inf 基于平台基础性能, 如内存大小、函数带宽、单位数据量计算时间等, 对模型推理的性能表现进行建模, 得到性能目标函数, 再根据函数数量限制、成本目标、SLO 等设立约束条件, 将模型分割问题转化为混合整数二次规划 (MIQP) 问题, 并使用已有工具求解, 从而从数学层面上直接解出最佳分割策略. AsyFunc[51]聚焦于深度学习推理, 作者的研究发现, 深度神经网络的每个层对计算资源的敏感程度与其内存使用率大多呈反相关. 基于这个发现, AsyFunc 提出“影子函数 (shadow function)”的概念, 这种函数不会载入整个模型, 而是只载入那些资源敏感的层, 整体内存占用较低. 当请求量增加, 系统扩展时, AsyFunc 为现有函数 (称为 body function) 创建影子函数, 而不是载入完整模型的新函数. 在推理计算时, 对模型中资源不敏感的层, 原函数就可以很好的应对请求增加, 而对于资源敏感的层, 原函数会和影子函数配对并行计算. 通过对影子函数资源分配的动态调整, AsyFunc 可以适应负载的动态变化, 从而在扩展时节省了大量内存资源, 同时还满足了 SLO 保证. Tetris[52]也针对深度学习推理, 它通过建立在每个主机上的 tmpfs 内存文件系统来实现张量共享. Tetris 拥有自己的缩放和调度引擎, 可以自主选择重用或新建实例, 实现高效的共享并减少内存占用.

MArk[53]综合利用了 AWS 上多种不同类型的云服务来执行机器学习推理任务, 包括 VM、硬件加速器 (如 GPU) 和服务器无感知计算. 其中服务器无感知计算用于应对偶发的请求高峰, 利用它快速伸缩的性能来填补传统服务器集群无法及时扩容响应的时间空隙, 只是会付出略高一点的价格.

2.3 其他类型任务

(1) 函数编排. 为了简化使用函数构造应用程序的流程, Serverless 函数编排工具应运而生, 如 AWS 的 Step Functions[54], 微软 Azure Functions 中的 Durable Functions (DF)[55]扩展等. 除了这些独立的编排器外, Unum[56]提出了在应用程序层运行的编排器. 它表现为一个运行时库, 与用户定义的函数一起执行, 支持与 AWS Step Functions 相同的编程接口和执行保证, 同时提供了更多的灵活性. Kappa[57]提供了一套接口简单的编程框架, 它屏蔽了服务器无感知计算的细节, 自动帮用户处理了并发和函数间消息传递任务, 并用检查点技术处理了容错问题, 大大简化了服务器无感知应用的开发工作.

(2) 视频处理. 视频处理是常见的高负载任务, ExCamera[58]在服务器无感知环境中实现了高效的视频编辑、转换和编码操作. 此外, ExCamera 还实现了一个通用库 mu, 专用于在服务器无感知平台上快速部署并执行大批量任务. Sprocket[59]受到 ExCamera 的启发, 使用了 ExCamera 的函数部署库 mu 并做了适应性修改, 与 ExCamera 不同的是, Sprocket 给用户提供了一套流水线模式的描述语言, 可以让用户自定义视频的处理流程.

(3) 代码编译. gg[60]使用服务器无感知计算来编译代码, 为了解决编译的依赖问题, gg 包含了一个可以在函数内部运行的轻量级虚拟机模块 thunk, 每一个文件被发给一个 thunk 编译, 各文件的依赖关系构成一个 DAG. 编译过程从 DAG 的根节点开始, thunk 使用惰性模式执行, 将相互依赖的文件依次编译连接, 最终完成编译.

(4) 科学工作流. DayDream[61]在服务器无感知平台上执行科学工作流. 它针对常见的 DAG 任务设计了调度器, 提前启动函数实例以降低延迟. 在 AWS Lambda 上, DayDream 在时间和成本上的表现都优于与传统 HPC.

(5) 实时计算. 常规的服务器无感知平台无法支持实时计算的要求, Szalay 等人[62]提出了一个实时 FaaS 平台, 针对实时计算的要求设计了分析模型和启发式分区调度算法, 旨在将给定的任务集合分配到合适的集群节点 CPU 上, 以满足任务的截止时间要求. 当然, 该系统的实时性需要底层集群性能和网络速度的保证.

(6) 服务器无感知系统分析. 为了更好地研究服务器无感知系统, 相关的分析工具和 Benchmark 必不可少. FaaSProfiler[63]是一个开源的服务器无感知平台测试和分析工具, LambdaML[64]可以对 AWS Lambda 上的机器学习任务和传统 IaaS 平台进行性能比较和评估, FunctionBench[65]、FaaSdom[66]和上海交通大学的 ServerlessBench[67]都是对服务器无感知系统进行全方面的评估的 Benchmark 工具包.

2.4 对比与小结

表 1 总结了本节提到的所有系统并做出了比较. 可以看出, 一方面, 许多任务类型已经开始迁移到服务器无感知计算环境中, 大部分的任务都可以从服务器无感知计算高效伸缩和精细计费的特点中受益, 未来仍可以迁移更多的任务类型到服务器无感知环境中. 另一方面, 迁移到服务器无感知环境中的任务, 或多或少都做出了功能上的妥协, 或是计算规模受限, 或是计算类型单一, 难以实现原任务的完整特性, 其主要原因是服务器无感知计算的各种资源和通信限制. 不过, 目前已经有许多研究针对这些限制做出了各种改进 (这些工作将在下文中介绍), 这些新的系统可以支持更多类型的计算任务, 并提升现有任务的性能表现, 利用新的系统实现更多类型的计算任务仍是有意义的工作.

表 1 本节概述的所有系统及比较

系统和工作	基于FaaS系统	任务类型	主要技术内容
Pywren[26]	AWS Lambda	大数据分析	自动map分发任务执行, 简化服务器无感知程序开发流程
IBM-Pywren[27]	IBM Cloud Functions	大数据分析	OpenWhisk上的Pywren[26]扩展版
Numpywren[28]	AWS Lambda	矩阵计算	AWS Lambda上的线性代数计算, 通过切分矩阵计算
MARLA[29]	AWS Lambda	大数据分析	基于AWS Lambda的MapReduce, 支持自动分区
Lambada[32]	AWS Lambda	数据查询	交互式数据查询框架
Starling[33]	AWS Lambda	数据库检索	数据库检索引擎, 通过分阶段Shuffle降低成本
Pheromone[34]	Cloudburst[37]	大数据分析	以数据为中心的数据处理系统
WuKong[35,36]	AWS Lambda	DAG任务	基于AWS Lambda的DAG任务执行
Cloudburst[37]	—	DAG任务	支持本地缓存和函数间通信的DAG任务执行系统
Cirrus[38,39]	AWS Lambda	ML训练	基于参数服务器模型的机器学习系统
SMLT[40]	AWS Lambda	ML训练	带有贝叶斯优化资源调度策略的机器学习系统
Siren[41]	AWS Lambda	ML训练	使用强化学习调度资源的机器学习系统
MLLess[42,43]	IBM Cloud Functions	ML训练	过滤“非显著”更新、自动资源缩减的机器学习系统
λDNN[44]	AWS Lambda	DNN训练	通过任务性能模型, 自动选择资源配置方案的DNN训练系统
Dorylus[45]	AWS Lambda	GNN训练	由传统VM负责图计算, FaaS负责张量计算GNN训练系统
ML-FaaS[46]	OpenWhisk[7]	ML推理	使用服务器无感知计算进行机器学习推理
Batch[47]	AWS Lambda	ML推理	使用批处理执行ML推理任务, 提高资源利用率
INFless[48]	OpenFaaS[9]	ML推理	使用非均匀批处理执行ML推理任务, 支持异构资源统一抽象
Gillis[49]	多平台	NN推理	基于“层组”切分模型的大型神经网络推理系统
AMPS-Inf[50]	AWS Lambda	NN推理	基于性能建模切分模型的大型神经网络推理系统
AsyFunc[51]	Knative[11]	DL推理	扩展时候只扩展出函数来计算资源敏感的层, 从而节省内存
Teris[52]	OpenFaaS[9]	DL推理	支持张量共享的深度学习推理系统
MArk[53]	AWS	ML推理	同时使用多种云服务执行推理任务
Unum[56]	多平台	函数编排	以运行时方式运行的Serverless函数编排器
Kappa[57]	AWS Lambda	函数编排	简化的服务器无感知计算编程框架
ExCamera[58]	AWS Lambda	视频处理	基于AWS Lambda的视频处理系统
Sprocket[59]	AWS Lambda	视频处理	基于AWS Lambda的视频处理系统, 支持用户自定义处理流程
gg[60]	AWS Lambda	代码编译	基于AWS Lambda的代码编译系统
DayDream[61]	AWS Lambda	科学工作流	基于AWS Lambda科学工作流执行
Szalay等人[62]	—	实时计算	支持实时计算的服务器无感知系统
FaaSProfiler[63]	OpenWhisk[7]	系统分析	开源的服务器无感知平台测试和分析工具
LambdaML[64]	AWS Lambda	系统分析	针对机器学习任务在FaaS和IaaS之间的性能比较评估
FunctionBench[65]	不限	Benchmark	服务器无感知计算Benchmark
FaaSdom[66]	不限	Benchmark	服务器无感知计算Benchmark
ServerlessBench[67]	不限	Benchmark	服务器无感知计算Benchmark

纵观所有研究工作可见, 最多见的计算任务是机器学习类任务, 包括推理和训练两大类, 其中推理相对较为适合服务器无感知计算, 而训练任务仍然在规模或模型上受到限制. 事实上, 对大部分 ML 模型而言, 通过数据和模型切分, 在服务器无感知平台上完整实现并没有技术上的障碍, 真正的限制此类任务移植的是性能和成本问题, 其中性能问题主要来自通信和 I/O, 而这两个问题又会进一步推高计算成本, 阻碍了任务的迁移. 2018 年加州大学伯克利分校的研究[38]就指出缺乏高性能的存储方案是制约服务器无感知环境下机器学习性能的原因之一. 此后虽然有不少服务器无感知计算的存储优化工作, 但它们大多不会为 ML 进行优化, 如 Cloudburst[37], 使用缓存提高性能, 但它是为 DAG 任务设计的. 虽然这些系统理论上也可以用于执行 ML 任务, 但仍有明显的优化空间. 因此, 针对机器学习任务的通信和 I/O 优化是值得关注的研究方向.

在机器学习方面, Barrak 等人[22]做了针对性的研究综述, 具有较好的参考价值.

3 沙箱环境优化

沙箱是负责执行 Serverless 函数的隔离环境, 它通常是容器或虚拟机, 承载着大量函数的运行. Serverless 函数会有大量的突发和并发请求, 且需要被快速响应, 这对函数的沙箱环境提出了很高的要求, 沙箱的性能会显著影响整个系统的性能. Shahrad 等人[63]的研究指出, 函数计算的容器化执行至多会比本地原生执行慢 20 倍, 而冷启动可能超过短函数执行时间的 10 倍, 优化沙箱环境的性能是提高服务器无感知系统性能的核心. 本节将从沙箱实现方案和冷启动优化这两个关键方向综述相关研究工作.

3.1 沙箱方案

沙箱技术基于虚拟化技术, 具体的实现可以基于不同的系统抽象层, 不同的实现方案代表着不同的权衡选择. 主流的沙箱方案以虚拟机和容器为主, 也有容器内多进程、进程内多线程等细节优化方案. 通常, 沙箱环境的抽象层次越高, 就有越多的层被虚拟化, 减少了为用户分配的独占资源, 从而有更多的资源被共享, 性能也就越高. 不过, 共享的资源越多, 函数的隔离性和安全性就越低. 许多系统直接使用现有的沙箱框架, 如成熟的容器系统 Docker、Containerd, 以及 Unikernels、LightVM、CNTR、gVisor 等常用且性能出色的虚拟化技术, 不过这些通用技术并不是针对服务器无感知计算进行优化的研究, 本文并不详述.

3.1.1 虚拟机方案

在各种沙箱方案中, 虚拟机无疑是隔离性最好的选择, 但传统虚拟机的性能远不能满足 Serverless 函数的轻量级需求, 因此必须进行精简, 简单的结构不仅可以缩短启动时间, 还可以减少攻击面, 提高安全性. 亚马逊的虚拟机管理系统 Firecracker[68]是其中的典型代表, Firecracker 是 AWS Lambda[1]的后台组件之一, 它负责管理和监控大量的 Serverless 函数专用虚拟机 MicroVM, 其结构极其精简, 启动内核配置极少, 专为服务器无感知场景设计. 其中, 每个 MicroVM 只负责执行一个函数, 每个函数的运行环境完全独立, 尽可能地保证了安全性. Fireworks[69]基于 Firecracker 实现, 并采取从快照启动的机制以提高启动速度, 虽然 Firecracker 本身也支持从快照启动, 但 Fireworks 在创建快照前对函数代码使用了 JIT 即时编译, 而后将编译后的状态保存到快照中, 该方法除了能进一步减少启动时间外, 也可以有效提高内存利用率.

3.1.2 容器方案

容器是最常见的沙箱方案, 它是在性能和安全性之间较为合适的一个折衷, 且具有良好的可替换性, 大部分商业和开源服务器无感知平台都使用容器执行函数. 如果需要, 容器内还可以执行多个函数, 当然, 函数之间的隔离性会降低.

服务器无感知计算系统 OpenWhisk[7]、OpenLambda[8]、OpenFaaS[9]等都是使用 Docker 或 Kubernetes 容器作为函数沙箱. 微软 Azure Functions[2]使用“Function Apps”来执行函数[70], Function Apps 可以看作是函数的容器, 运行在虚拟机之内. 阿里云函数[5]使用的是 RunD[71], 它改进了 rootfs 的实现, 精简了客户内核, 并引入了轻量级的 cgroup 设计, 实现了高密度 (384 GB 内存的节点上部署了超过 2 500 个安全容器) 和高并发 (每秒启动 200 个容器) 的性能. Nightcore[72]是一个容器化的运行时环境, 提供函数的调用和执行环境. 它从多个层面优化了函数的执

行速度, 实现了微秒级的性能开销.

在实际生产环境中, 函数之间, 尤其是同一应用程序或用户的函数所使用的资源有相当部分是重复的. 商业平台出于安全性考虑, 通常不会允许跨函数或跨应用的资源重用, 但在非商用领域, 许多研究工作通过跨函数的资源重用实现了性能提升. MXContainer[73]的理念是于让同一个函数的多个调用共用资源, 通过 I/O 请求合并, 共享初始内存状态等方法, 实现了高执行速度和吞吐量, 同时也降低了内存占用. Catalyzer[74]基于快照恢复来重用资源, 并将不必要的资源载入推迟, 实现了亚毫秒级的启动速度. SOCK[75]是一使用多种技术优化的容器系统, 它在 3 个层面对容器做出了优化, 具体将在第 3.2.1.3 节中详述.

为了提高资源利用率, 许多系统允许在一个容器内执行多个函数, 例如 SOCK[75]和 SAND[76], SOCK 实现了函数之间的进程资源共享, 而 SAND 实现了函数之间的高效通信. 微软的 Function Apps 虽然每个只能执行一个函数, 但它允许多个 Function Apps 在同一个虚拟机内运行, 同样实现了资源共享的目的. 不过, 这种共享资源的执行方式难以避免的会降低隔离性, 带来安全问题, 所以大部分的系统都仅限同一用户或同一应用程序的函数在同一个容器或虚拟机内执行, 以减小安全隐患.

上海交通大学的 Molecule[77]是建立在异构计算机硬件上的服务器无感知计算框架, 它提出了矢量化沙箱 (vectorized sandbox) 的概念, 给领域特定的加速器提供了统一的硬件抽象. 可以让不同的硬件设备 (如 NVIDIA DPU、FPGA 和 GPU 等) 都能应用于服务器无感知计算. Molecule 开发了中间层 XPU-Shim 来弥合底层异构操作系统和服务器无感知系统之间的差异, 为开发者提供了统一的开发接口, 并屏蔽了底层异构硬件的细节. Molecule 是第 1 个弥合异构硬件差异的系统, 与传统的建立在同质计算机上的系统相比, Molecule 显著提高了函数密度 (高出 50%) 和应用性能 (34.6 倍).

3.1.3 进程方案

进程是比容器更高一层的方案, 虽然本质上操作系统中的所有代码都是运行在进程中的, 不过操作系统并不会为进程提供严格的隔离环境, 如 SOCK[75]和 Cloudburst[37]虽然允许多个函数在一个容器内执行, 但它们的进程不构成沙箱环境. SAND[76]允许一个容器内运行多个函数, 在相同应用的函数间不考虑隔离, 但在不同应用函数间采用 Linux 控制组 (cgroups) 实现资源隔离. FAASM[78]是完全基于进程的函数隔离执行环境, 它使用了 WebAssembly 中的软件故障隔离 (software-fault isolation, SFI) 技术来隔离内存, 同时又保留了函数间共享内存区. 这种操作系统层面的内存共享使得在同一台主机上的不同函数不再需要通过外部存储服务来交换数据, 主机内的内存共享大大提高了函数间的通信速度, 也减少了内存占用.

3.1.4 线程方案

如果想进一步提高函数资源共享, 仍可继续提高执行环境的抽象层次. Faastlane[79]将函数的并行单位挖掘到线程级, 它对支持多线程并发的语言就使用线程执行函数, 这让它获得了极高的函数间通信效率 (延迟降低 99.95%). 但线程的安全性就更低了, 为此, Faastlane 引入了 Intel Memory Protection Keys (MPK) 技术来提供轻量级的线程级别的隔离域, 以对敏感数据提供一定的安全性保障. aWsm[80]在进程内并行执行多个函数, 它基于 WebAssembly 的沙箱环境, 提供了进程内的隔离执行环境, 并且可以在沙箱之间共享运行时, 适合于在边缘计算中提供高效的执行性能.

3.1.5 更高层方案

通常线程就是抽象层次最高的执行环境了, 但仍有研究尝试更高的层次. Photons[81]将抬高抽象层次的理念推到了极致, 它的执行本质上可以认为是把多个函数的代码直接粘贴到一个文件内执行, 实现了函数间的运行时层、句柄层, 乃至部分程序状态的共享, 其核心问题就是解决变量冲突. Photons 对 Java 代码进行改造, 使用 Javassist 库, 从字节码层面修改待执行代码, 解决了变量访问冲突的问题. 在如此高的抽象层次上, Photons 实现了极好的性能表现, 不过, 它基本不保证隔离性和安全性.

3.2 冷启动优化

服务器无感知计算的函数被设计为非常轻量级且短寿命的单元, 通过大量并行实现高效计算, 这导致函数会

被频繁的启动和停止, 而沙箱的启动延迟, 尤其是冷启动延迟, 在大量频繁的请求中会被放大到极为显著的程度, 如不加优化, 冷启动延迟会严重拖慢整个系统的性能. 现有研究主要在两个场景中进行优化: 1) 单个沙箱的启动优化, 其核心思路是尽量用热启动代替冷启动, 尽量使用已经载入的资源而不是重新载入资源. 具体而言, 通常可以从两个方向实现热启动效果: 重用已载入资源和提前载入需要的资源. 2) 多沙箱并发的启动优化, 聚焦于大量沙箱同时启动时的性能优化问题, 以提升系统的扩展速度.

3.2.1 资源重用

在启动沙箱环境时重用已经载入的资源, 可以有效地提升启动速度. 除了重用整个容器或 VM 之外, 也可以部分重用已载入的进程资源或内存块等, 各大服务商也基本都采用了不同程度的热启动机制. 不过需要注意的是, 重用代表着一定程度的数据共享, 共享会带来安全隐患, 重用越多, 性能越强, 但安全隐患也就越大. 现有的研究工作在不同的系统层次上进行了重用, 尝试在性能和安全之间取得不同程度的平衡.

3.2.1.1 基于缓存池的重用

缓存池是最常见的容器或 VM 重用方案, AWS Lambda 和 Azure Functions 都使用了缓存池策略[70], 将停止的 VM 或 Function APPs 保活 (keep alive) 一段时间, 以便下次调用时可以快速启用. 但常规缓存池的命中基本靠运气, 为了改善命中率, 可以使用更灵活的保活时间调整策略. FaasCaches[82]使用缓存策略的思路改进保活策略, 它引入 Greedy-Dual 算法, 计算函数的调用频率、初始化开销和资源占用大小来综合计算函数的“优先级”, 调用频率越高、初始化开销越大、资源占用越小的函数优先级越高, 优先级高的函数将被优先保持活跃. FaasCaches 还使用命中率曲线来调整保活策略, 对于频繁调用的函数, 可以采取较短的保持活跃时间, 以便及时更新缓存池中的函数; 对于不经常调用的函数, 可以采取较长的保持活跃时间, 以减少冷启动开销, 同时保持系统资源高利用率. Vahidinia[83]的研究则引入了更复杂的强化学习算法来学习函数的调用模式, 从而确定为空闲容器保活的最佳时间.

3.2.1.2 基于快照的重用

快照 (或检查点) 是将内存数据保存回磁盘的技术, 典型例子如 Linux 的 CRIU (checkpoint/restore in userspace) 工具, 它的作用是让单个进程休眠, 其内存数据全部转储到磁盘文件, 待需要时再从文件取回内存数据, 恢复进程运行. 基于快照恢复技术可以快速重建 OS 环境, 比从头启动 OS 要快得多, 因此是解决冷启动问题的热门方向. 现在, Firecracker[68]已经支持快照启动 (https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md#about-microvm-snapshotting), Fireworks[69]也使用了快照启动机制. 除此之外, 许多其他研究也基于快照技术提出了有价值的优化方法.

Prebaking[84]在函数第一次初始化后, 使用 CRIU 工具为函数创建快照, 并将快照数据写入到容器镜像中, 这样容器在下次启动时就可以跳过函数初始化阶段, 通过恢复镜像直接还原到上一个函数的执行状态. SEUSS[85]提供了比 CRIU 更高效的快照技术, 它基于 Unikernels 实现, 从操作系统层面直接访问页表并创建快照. SEUSS 使用了页共享机制, 每次只捕获自上次快照以来发生变化的页, 因此可以快速创建快照, 并且缩小了快照尺寸. 上海交通大学的 Catalyzer[74]在系统快照的基础上实现了更为细致的容器资源重用. 它的原则是在恢复检查点时只载入必要的资源, 只恢复必要的状态, 其他非重要的数据加载都尽量推迟. 以此优化启动过程中的各个环节, 包括系统状态恢复、函数代码加载和 I/O 操作. Catalyzer 还引入了新的操作系统原语 sfork (沙箱分支), 通过直接重用运行中的沙箱实例的状态, 进一步减少启动延迟, 此技术尤其适合于在已有沙箱的基础上创建相同编程语言的沙箱. Catalyzer 实现了亚毫秒级的启动延迟, 并在蚂蚁金服中实际应用.

Ustiugov 等人[86]的研究发现, 从快照启动的函数虽然启动快, 但运行速度却相比从头启动的函数慢了 95%, 这是因为从快照启动的函数在运行中会遇到频繁的内存缺页, 需要不断从镜像文件中随机读取, 拖慢了速度. 进一步的分析发现, 函数遇到的缺页其实只占总内存数据的一小部分. 基于这个发现, 他们提出了系统 REAP, 通过在函数第 1 次调用时记录其访问过的内存数据, 并紧密保存 (以便可以连续而不是随机读取) 到文件中, 在函数再次启动时从文件预取数据到内存, 减少了内存缺页的概率, 显著提高了快照函数的运行速度.

3.2.1.3 内存资源重用

在现有的容器或 VM 中, 或多或少的重用已有的内存资源, 是提升启动性能的常见手段. 如 Linux 的 fork 方法, 它可以创建一个与当前进程完全相同的子进程, 子进程会完全复制父进程的内存空间, 从而继承父进程的代码、数据、堆栈等资源 (但数据修改是独立的). fork 方法可以快速复制进程且不会浪费资源, 是启动新函数的合适方案. 使用 fork 方法的典型工作如 SOCK[75], 它从 3 个层面对容器做出了优化: (1) 对底层 Linux 系统进行精简化修改. (2) 引入安卓中的 Zygote[87]来管理进程 (进程用于执行函数), 使用 fork 方法创建新进程, 使用树形结构来缓存和管理进程, 以最大程度地重用已经载入的库. 图 6 是 SOCK 的进程管理方式, 图中圆表示进程, A–E 代表库, 箭头代表 fork 操作. 当一个新的函数需要启动一个新的进程时, 可以根据所需要的库列表, 在树中寻找最接近的节点, fork 出新的进程, 再载入缺失的库, 最后再将新的进程也放入树中. 这种方法可以大大减少库的重复载入, 提高启动速度. (3) 设计了 3 级缓存机制, 第 1 级重用句柄和运行时, 第 2 级重用 Zygote 进程, 第 3 级重用各种 Python 库, 为此, SOCK 提前内置了一批最流行的 Python 库. 在多种机制的共同作用下, SOCK 获得了显著的性能提升.

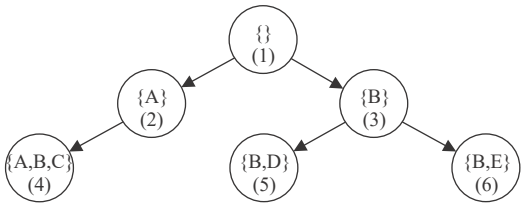

图 6 SOCK 按照树形结构 fork 进程, 以保证库的最大程度重用

MITOSIS[88]是一种操作系统原语, 它提出了一种“远程 fork”的操作, 通过 RDMA (remote direct memory access, 远程直接内存访问) 跨节点地 fork 出新的容器. MITOSIS 并不会把父容器的数据全部从网络传输到新节点中, 而是只传输一个包含基本系统配置和内存页表信息的“压缩描述符”, 新节点基于这些描述信息创建新容器, 待新容器在运行中发生内存缺页时, 再从远程主机读取内存页面. 这种方法可以实现超快速的跨节点容器创建 (1 s 超过 10 000 个), 不过它创建出的容器在运行时会频繁地出现内存缺页, 需要高效的网络性能来保障其运行速度.

Medes[89]并引入了一种介于冷暖之间的新沙箱状态, 称为去重状态. Medes 对所有去重状态的沙箱内存进行分块并计算哈希值, 并在整个集群内进行内存去重. 去重后的沙箱内存占用会大大缩小, 从而可以在内存中容纳更多沙箱. 待需要执行函数之前, 去重沙箱通过网络取回缺失的内存块并恢复为暖沙箱状态, 恢复过程要快于加载新沙箱, 从而提高了冷启动速度.

XFaaS[90]是 Meta 公司内部使用的服务器无感知系统, 为了提高冷启动速度, XFaaS 在集群内为各种编程语言设置了运行时持续运行的节点, 它将相同语言的函数都调度到这些节点上运行, 并且在节点之间共享 JIT 编译数据, 从而加速 JIT 代码的编译. 为了避免在高负载时服务节点性能不足, XFaaS 使用了多种方案, 包括在一个进程中执行多个函数, 推迟执行某些不紧急的函数, 为每个函数定义配额, 并对超出配额的函数限流, 以及允许调用者指定函数的未来执行开始时间, 这样可以帮助平台预测未来的负载, 从而预分配资源. 在多种手段下, XFaaS 声称“消除了冷启动时间”. 考虑到 XFaaS 是内部平台, 面向内部任务, 对安全的要求更低, 负载的可预测性也更强, 因此可以实现更好的优化效果.

3.2.2 资源预载入

除了重用已有资源外, 预载入资源也是一个常见的方案. 预载入需要提前预测程序未来的执行情况, 当然这种算法不可能绝对准确. 为了预测程序的执行, 主要有基于控制流的预测方法和基于统计的预测方法.

3.2.2.1 基于控制流的预测

ORION[91]通过分析 DAG 控制流和函数的执行时间分布, 预测函数的请求时间, 提前预热容器; SpecFaaS[92]通过分析控制流和数据依赖来预测即将被执行的函数, 并提前准备数据和载入函数, 实现不同运行阶段的函数并行

载入或执行, 类似于处理器中的“超级流水”执行方式, 提高了程序速度并降低了延迟. Kraken[93]除了可以解析程序的 DAG 结构, 还可以适应动态变化的 DAG 关系. Kraken 动态分析程序的运行时行为, 检测函数依赖关系变化, 从而动态更新 DAG 图. 基于程序的 DAG 结构, Kraken 计算函数的调用概率, 给函数赋予权重, 再考虑负载情况, 为函数提前准备容器.

3.2.2.2 基于统计的预测

Icebreaker[94]通过统计函数运行记录中的周期性来预测函数未来的启动时间; Vahidinia[83]的研究则使用 LSTM 模型来预测未来的函数调用时间; Fifer[95]也使用了 ML 模型来预测未来的请求到达概率; Atoll[96]通过分析函数的请求到达模式来推测未来可能的最大负载量; 而 Aquatope[97]则使用贝叶斯优化算法来预测函数被调用的概率, 从而提前预热容器.

3.2.2.3 混合预测方法

Xanadu[98]综合利用了控制流预测和机器学习预测两种方法, 对于显式的执行链, 通过对代码的静态分析获取执行路径, 对于动态的执行路径, Xanadu 则使用生成模型, 基于工作流的运行历史数据来训练模型, 推测程序在未来每次执行中最可能的执行路径, 进而提前为路径上的函数部署资源. 基于统计的生成模型虽然也无法绝对准确的预测程序未来的执行路径, 但在实验中也可以达到 80% 以上的正确率.

Pagurus[99]结合了重用和预载入的方法, 在预载入容器的时候重用现有的资源. Pagurus 的核心思路是让不同的函数可以共用容器, 从而让新函数可以利用其他老函数尚未关闭的容器 (暖容器) 的资源. 但不同的函数有不同的库需求, 为保证不同函数的容器可以通用, Pagurus 分析函数之间的依赖包差异来计算函数的“相似性”, 选择与新函数尽量相似的老函数容器并预载入缺少的包, 从而提高了容器的预载入速度并节省了资源.

微软提出的混合直方图策略 (hybrid histogram)[100]结合了多种方法, 在一般情况下, 使用空闲时间分布直方图来确定保活窗口和预热窗口的长度; 当算法判断直方图对应用程序的行为代表性不强时, 使用常规保活策略, 即固定保活窗口并关闭预热窗口; 对于一些很少调用或行为非常不规律的应用程序, 无法用直方图描述其行为, 则使用时间序列分析来预测下次调用时间, 以便调整预热窗口和保活窗口的大小. 在 Azure Functions 中的实际应用中, 该策略显著降低了冷启动频率.

3.2.3 并发启动优化

在使用容器沙箱方案的服务器无感知集群内, 需要快速启动大量容器时, 容器镜像的分发速度会显著制约容器置备速度, 传统的单节点分发方式会在容器注册节点处产生严重的瓶颈效应. 对此, 阿里云[5]的 FaaSNet[101]使用平衡二叉树的结构来管理 VM (VM 内运行容器), 它为每个函数 (指函数定义, 非运行时函数) 创建一个平衡二叉树, 将分配给该函数的 VM 全部加入树中作为节点, 形成一个快速可扩展的容器分发网络. 被请求的容器镜像将先传给树的根节点, 而后从根节点向叶子节点逐级传递, 直至到达目标 VM. 这种树形结构使得镜像数据在 VM 间的相互传递, 大大减轻了容器注册节点的下载压力. 在 VM 数量变化时, FaaSNet 会自动保持二叉树的平衡. 这个树形容器分发网络显著提升了镜像分发速度, 提高了大量容器置备速度, 减少了函数的冷启动延迟.

3.3 对比与小结

沙箱环境的优化是提高整个系统性能表现的关键环节. 表 2 总结了本节提到的所有系统并做出了比较.

在各种沙箱方案中, 商业平台主要都采用虚拟机方案, 这是因为他们对安全性的要求都很高, 当前容器和其他沙箱方案对安全性的考虑都不多, 通常只做基本的内存隔离, 并不考虑复杂的攻击防御. 在更高等级安全保障的前提下实现更高性能的沙箱方案是有实际应用价值的研究方向, 其中, WebAssembly 是颇具前途的运行时环境, Faasm[78]和 aWsm[80]都使用 WebAssembly 实现了高抽象层次的隔离方案. 关于 WebAssembly 的评估, Kjorveziroski[23]的研究综述具有较好的参考价值.

在开源平台中, 容器方案是主流选择, 即使是进程或线程方案也基本都是运行在容器内, 因此容器的启动速度是沙箱性能表现的关键. 当前, 基于快照 (检查点) 的启动方案效果显著, 是提升容器冷启动速度的热门方向, 其中 Catalyzer[74]已经实现了亚毫秒级的启动速度, 对大部分服务器无感知应用而言, 这个延迟已经不至于造成性能瓶

颈. 不过, 基于快照的方案仍有一些需要解决的问题, 值得在未来的研究中关注:

(1) 从快照启动的容器虽然启动快, 但运行慢[86], 因为许多运行资源需要临时从镜像中读取. 一些研究将资源压缩[85], 或跨节点重用[89], 或推迟载入[74], 更会加剧这个问题. 如何高效创建和使用快照, 综合考虑启动速度和运行速度并找到合适的平衡点, 是值得研究的问题.

(2) 高效的启动往往倚仗大量的资源缓存, 这对服务商并不友好. 如何在保障执行性能的同时, 提高缓存的命中率, 降低集群的资源占用, 是需要关注的问题.

在冷启动优化方面, Golec 等人[24]的研究综述具有较好的参考价值.

表 2 本节概述的所有系统及比较

系统和工作	技术类型	主要技术内容
Firecracker[68]	VM沙箱	亚马逊的精简虚拟机系统
Fireworks[69]	VM沙箱	基于Firecracker[68], 使用JIT编译和快照加速的虚拟机系统
RunD[71]	容器沙箱	阿里云的精简虚拟机系统, 改进rootfs, 引入轻量级cgroup
Nightcore[72]	容器沙箱	利用现有的OS抽象来实现容器级隔离, 微秒级开销
MXContainer[73]	容器沙箱	在同一函数的多个调用之间共用处理器、内存、外部存储和I/O带宽资源
Catalyzer[74]	容器沙箱	基于快照加速的容器系统, 并推迟非重要的数据加载, 亚毫秒级启动
SOCK[75]	容器沙箱	精简Linux, 使用树形结构缓存进程资源, 内置Python库
SAND[76]	服务器无感知平台	使用cgroups对不同应用的函数实现进程间隔离
Molecule[77]	异构硬件沙箱	提出了矢量化沙箱, 为异构硬件提供同一的编程抽象
Faasm[78]	进程沙箱	使用WebAssembly中的SFI技术来隔离内存
Faastlane[79]	线程沙箱	使用线程并行执行函数, 利用Intel MPK实现隔离域
aWsm[80]	进程内沙箱	使用WebAssembly沙箱环境提供进程内隔离, 亚毫秒延迟
Photons[81]	运行时内沙箱	在进程内将运行时也共享的函数执行方案
FaasCaches[82]	容器保活策略	根据函数优先级决定保活时间
Vahidinia等人[83]	容器保活策略	用强化学习算法决定容器保温时间
Prebaking[84]	快照优化技术	将快照写入到容器镜像中以加快启动初始化速度
SEUSS[85]	快照系统	比CRIU更高效的快照技术
REAP[86]	服务器无感知平台	帮助快照提前载入可能缺页的内存, 优化快照运行速度
MITOSIS[88]	远程fork原语	通过RDMA实现快速远程fork容器, 实际内存数据待缺页时再从远程读取
XFaaS[90]	服务器无感知平台	将相同编程语言的函数调度到一起执行, 以消除冷启动延迟
Medes[89]	服务器无感知平台	基于快照启动, 并在整个集群内实现内存块重用
ORION[91]	容器预热技术	分析控制流和函数执行时间分布, 提前预热容器
SpecFaaS[92]	容器预热技术	分析DAG控制流和函数执行历史, 实现函数预载入
Kraken[93]	资源管理框架	分析DAG控制流实现函数预载入, 并支持动态更新的DAG
Icebreaker[94]	容器预热技术	通过分析函数运行记录中的周期性来实现预载入
Atoll[96]	服务器无感知平台	分析请求到达模式推测未来负载量, 并预分配资源
Fifer[95]	资源管理框架	使用ML模型预测未来的请求到达概率, 并预分配资源
Aquatope[97]	资源管理框架	使用贝叶斯优化预测未来的函数调用概率, 并预分配资源
Xanadu[98]	资源管理框架	综合使用控制流预测和机器学习推测函数的执行, 并预分配资源
Pagurus[99]	容器管理策略	让不同的函数可以共用容器以提高启动速度
Hybrid Histogram[100]	资源管理策略	综合使用直方图、常规保活策略和时间序列分析来实现预载入
FaaSNet[101]	容器镜像分发系统	使用平衡二叉树管理VM以构造高效可扩展的容器镜像分发网络

4 I/O 和通信优化

在服务器无感知环境下, 由于本地存储受限和函数间通信被禁止, 持久数据只能通过外部存储服务存取, 函数

通信也只能从外部服务中转. 这些数据传输的效率会显著影响应用程序的运行速度[102], 也是服务器无感知计算系统性能优化的一个重要问题. 本节将从 I/O 和通信两个方面综述相关的研究工作.

4.1 I/O 优化

I/O 可以在两个位置上进行优化, 一是上传下载数据的客户端, 缓存是客户端最常见的 I/O 优化办法; 二是存储服务端, 通过直接改进存储系统, 可以从根本上提升存储服务的吞吐速度.

4.1.1 基于缓存的优化

在 I/O 客户端, 通过设置缓存来缓解数据传输瓶颈是自然的思路, 缓存可以放置在不同的位置, 不同位置的缓存适应不同的任务类型, 也有不同的性能表现.

外部缓存是对系统影响最小的方法, 由于服务器无感知架构 BaaS (backend as a service, 后端即服务) 的结构, 服务之间都是松散的组合在一起, 使得外部服务的加入十分简单. Locus[103]聚焦于优化 Serverless 大数据分析任务中的 Shuffle 性能, 传统的 Shuffle 需要极其频繁的读写 S3 服务, 不但拖慢了速度, 而且会产生巨额的服务费用 (S3 读写收费), 直接导致大规模分析成为不可能. Locus 引入了内存数据库 AWS ElastiCache (Redis) 作为中间缓存, ElastiCache 读写速度快且按时间和容量计费. 由于 ElastiCache 费率较贵, Locus 仅将 Shuffle 过程中的数据交换部分转移到 ElastiCache 中暂存, 持久存储依然保存在 S3 中. 这样不但提升了 Shuffle 操作的速度, 而且大大节约了成本, 增强了服务器无感知环境下的 MapReduce 分析能力.

外部缓存始终需要通过网络访问, 并且容易出现网络瓶颈问题. 将缓存放在集群内部可以有效地解决这个问题. Faa$T[104]直接在服务器无感知集群内构建分布式缓存系统, 该缓存系统将缓存数据的存储和读写都分散开来, 消除了单函数访问大量数据时的瓶颈效应, 并且可以根据负载情况动态伸缩.

外部缓存和集群内缓存毕竟需要网络传输, 最直接的缓存方案是本地缓存, 操作系统层的缓存可以提供更高的访问速度. Cloudburst[37]在每个 VM 内设置了缓存空间, 并且进一步让缓存空间内的对象与外部存储中的原对象自动保持同步, 从而实现了便利的跨函数数据共享 (同时与一个外部存储对象同步).

比 OS 级的缓存更细粒度的是函数内缓存. 单个 Serverless 函数的资源虽然有限, 但 Ran Ribenzaft 的调查[105]发现, AWS Lambda 中的函数内存平均使用率只有 27%. 究其原因, 人们为了保证程序执行, 总是会按照预估的峰值内存占用量来分配内存, 这导致函数在大部分执行时间里都被分配了富余的内存容量. OFC[106]就使用函数内空闲的内存来做读写缓存, 因此 OFC 的缓存是“机会性”的, 每次运行时缓存空间的多少有无都是不确定的, 但从期望的意义上看, 大多数时候总会有些空间可以用. 为了提前确定是否有以及有多少空闲内存可用, OFC 使用机器学习算法来推测函数即将会占用的最高内存大小, 模型的特征数据来自于函数的输入参数类型. 模型的预测不可能绝对准确, OFC 设置了监控模块, 当内存即将不足时主动释放缓存让出内存, 并改进预测模型. 虽然缓存空间较为紧张, OFC 还是取得了明显的性能提升.

4.1.2 存储系统优化

外部存储是服务器无感知计算最常用的存储方式, 许多服务器无感知程序都选择将状态数据保存到外部存储, 这对外部存储服务的性能和吞吐量提出了很高的要求. Pocket[107]是为服务器无感知计算优化的存储系统, 它同时运用多种不同 I/O 速度的存储介质来应对各种场景下的存储需求, 包括 DRAM 内存存储、闪存存储和磁盘存储, 并根据性能和成本要求将数据放置在合适的介质上. Pocket 以任务为粒度调度资源, 并支持存储资源的自动扩容. Jiffy[108]借鉴了 Pocket 多层次存储的思想, 但调度粒度更细, 它以固定大小的块为单位分配存储资源, 从而允许在任务中动态扩展内存资源, 并在秒级尺度上满足任务的瞬时需求.

Shredder[109]将传统的“数据移到计算”的模式逆转过来, 将“计算移到数据”, 允许应用程序在存储服务器上嵌入小型存储函数, 用于直接操作数据. 这种的方案减少了数据的频繁移动, 在服务器无感知计算这种需要大量并行访问数据传输的场景下, Shredder 可以显著降低系统的网络开销.

4.2 通信优化

函数间通信是限制许多应用迁移到服务器无感知平台的障碍, 主流的服务器无感知系统都不支持函数间直接

通信, 这严重限制了函数间的交互能力, 制约了许多应用的移植, 例如图计算. 许多研究都尝试改善这个问题, 试图打通函数间通信渠道并优化通信方法.

FMI[110]选择直接解决通信信道问题, 它建立了完整的服务器无感知系统消息通信解决方案. FMI 以 C++库的形式运行, 模拟 MPI 的接口设计, 支持多种信道, 包括对象存储、键值存储、内存存储以及直接通信等. FMI 甚至在 AWS Lambda 上通过 NAT 内网穿透实现了函数间的 TCP 直连. 高效的通信渠道大大提高了系统性能 (机器学习通信加速 162 倍, 成本降低 397 倍). 不过, 在 AWS Lambda 上的 TCP 连接并不稳定, 可能会出现意外的超时, 需要重新连接, 不过在用户可控的环境中, FMI 可以稳定的连接.

除了 NAT 穿透之外, AWS Lambda 中常规的通信解决方案都是通过外部中转, 例如 ExCamera[58]使用 EC2 服务器中转函数消息. 但中转毕竟效率较低, Crucial[111,112]为 AWS Lambda 函数实现了共享内存服务 (基于 EC2 服务器), 从而实现了状态保存和函数间通信, 并自动处理了各节点上共享数据对象的同步和一致性, 并支持持久化保存.

在开源平台中, Boki[113]为函数提供了共享日志服务, 通过读写和订阅共享的日志数据, 函数之间可以更简便的实现状态保存和数据交换. Boki 的日志保存在外部存储节点中, 系统自动处理了日志的排序和一致性问题, 同时在函数节点上设置了日志缓存来优化读写性能, 实现了高吞吐量和低延迟的数据互通. Cloudburst[37]直接打通函数间的通信渠道, 通过交换函数的唯一 ID, 系统可以为函数之间打开点对点的 TCP 通道. SPRIGHT[114]在主机内设置了共享内存空间, 并通过共享内存进行通信, 实现了零拷贝的消息传递. 诺基亚贝尔实验室开发的 SAND[76]进一步优化了大量函数间的通信效率. SAND 重新设计了 Serverless 函数之间的通信机制, 实现了一套 2 级消息总线系统, 即主机内部第 1 级, 主机外部第 2 级, 主机内的函数间通信直接使用内部总线, 无需通过网络, 只有在跨主机通信时才需要使用外部总线, 通过网络发送, 如图 7 所示. SAND 尽量将同一应用的函数都放置在一台主机上, 使它们尽量集中, 以减少跨主机通信. 实验显示, SAND 相比 OpenWhisk 可以提升 43% 速度.

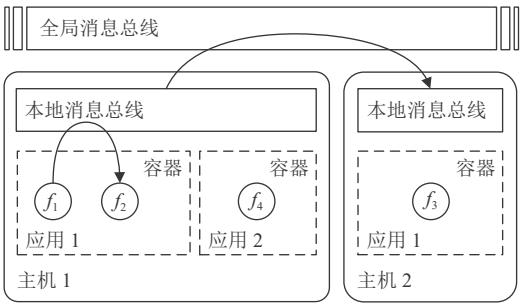

图 7 SAND 的应用管理和通信模型

Faasm[78]在提供进程隔离环境的同时使用了操作系统层的内存共享, 允许不同的函数直接共享同样的内存区块. 这使得同一台主机上的不同函数之间不再需要通过外部存储服务来交换数据, 大大提高了函数间的数据传输速度, 同时也降低了内存占用. SONIC[115]则同时使用了 3 种数据传输方式: 1) 虚拟机存储, 通过调度需要传递数据的函数到同一个虚拟机上, 然后本地互传; 2) 直接发送, 通过网络跨虚拟机传输数据; 以及 3) 远程存储, 即通过类似 AWS S3 形式的共用外部存储服务来交换数据. SONIC 综合使用 3 种传输方式, 通过分析函数输入数据的大小、网络带宽、DAG 中当前边 (代表数据传输) 所处的图结构等因素, 动态选择最合适的数据传输方式, 有效降低了传输延迟和运行性价比.

MXFaaS[73]通过减少请求次数的方法来优化 I/O 和通信效率, 它将同一函数的多个存储请求按访问键值合并处理, 多个远程函数调用按目标函数合并处理, 从而减少了 RPC 次数, 降低了 I/O 和通信请求量.

4.3 对比与小结

表 3 总结了本节提到的所有系统并做出了比较. I/O 和通信瓶颈是服务器无感知应用程序中最显著的性能瓶

颈, 它是服务器无感知计算为了高效伸缩而做出的权衡结果. 现有研究一方面集中于基于缓存进行优化, 另一方面则试图构建和优化函数间的通信渠道. 在商用系统如 AWS Lambda 中, 函数间直接通信受到限制, 通常只能通过中转节点解决通信问题, 通过外部缓存解决数据吞吐问题, 优化效果有限. 非商用系统会有更大的优化自由度, 可以通过设置缓存, 或者构建函数间直接通信信道等方法, 但这些优化操作或多或少会影响 Serverless 函数的弹性或隔离性, 未来的研究可以关注在优化数据传输后保障函数群的伸缩性.

表 3 本节概述的所有系统及比较

系统和工作	基于FaaS系统	技术类型	主要技术内容
Locus[103]	AWS Lambda	外部缓存	使用ElastiCache缓存优化大数据分析中的Shuffle操作
Faa$T[104]	Azure Functions	集群内缓存	设置在服务器无感知集群内的分布式缓存系统
Cloudburst[37]	—	主机内缓存	在VM上设置共享缓存空间, 打通函数间的TCP通道
OFC[106]	OpenWhisk[7]	函数内缓存	设置在函数内的“机会性”内存缓存
Pocket[107]	—	存储系统	同时运用多种存储介质应对不同需求的存储系统
Jiffy[108]	—	存储系统	比Pocket[107]更细调度粒度的存储系统
Shredder[109]	OpenLambda[8]	传输优化	将计算移到数据, 在数据端实现简单计算, 减少数据移动
FMI[110]	AWS Lambda	传输优化	服务器无感知消息通信解决方案, 支持多种信道
Crucial[111,112]	AWS Lambda	共享内存	基于EC2的AWS Lambda共享内存服务
Boki[113]	Nightcore[72]	共享日志	为函数提供共享日志服务, 借此还可以实现函数间通信
SPRIGHT[114]	Knative[11]	主机内缓存	在VM上设置共享内存空间, 通过共享内存实现零拷贝消息传递
SAND[76]	—	函数间通信	通过二级总线系统实现高效的函数间通信
Faasm[78]	Knative[11]	共享内存	在进程沙箱的基础上, 提供操作系统层的内存共享
SONIC[115]	OpenLambda[8]	传输优化	综合使用虚拟机存储、直接发送和远程存储3种传输方式交换数据
MXFaaS[73]	OpenWhisk[7], Knative[11]	存储和传输优化	将同一函数的多个请求合并处理以降低请求量

分布式计算任务的通信和数据读写模式和任务类型高度相关, 为不同类型的任务设计不同的缓存和通信系统会有显著的收益. 因此, 在服务器无感知计算的 I/O 和通信优化方面, 对特定类型的任务和场景进行针对性优化, 挖掘每种类型任务的性能提升空间, 是具有明显潜力的研究方向.

5 资源调度优化

资源调度是指为任务分配合适资源的过程, 整个流程包括资源准备、分配、调度和缩放操作. 服务器无感知计算平台的资源管理和调度是一个具有挑战性的问题, 在平台层面, 函数任务的高并发和高动态性需要调度策略做出对应的调整和适应, 在用户层面, 更灵活细粒度的计费方式也让用户层的精细调度有了必要性. 本节将从面向平台和面向用户两个层面综述相关的研究工作.

5.1 面向平台的调度

面向平台的调度技术和传统集群的调度技术基本相通, 只是需要针对服务器无感知计算的负载特点进行优化. 所有资源调度的根本目标都可以归结为两个, 减少资源占用和提高任务性能. 前者可以通过提高资源的利用率实现; 后者可以基于负载均衡或任务调度, 也可以借助不同类型的计算资源, 应对不同的场景.

5.1.1 提高资源利用率

提高资源利用率是资源调度的首要目标, 其实现的途径可以归结为两条: 通过合适的任务 (函数) 调度和资源分配, 提高现有资源的利用率, 减少多余资源的占用; 控制整体资源的伸缩, 在保证性能的情况下, 及时释放不需要的资源. 当然也有一些研究工作同时使用了两种方法.

Fifer[95]聚焦于降低资源的超额分配, 通过在一个容器内执行尽量多的函数来节省容器资源. 为了将更多函数放进一个容器执行, Fifer 使用 ML 模型来推测函数的执行时间, 推算出用户的 SLO 时间和预测实际执行时间之间

的“松弛时间”, 在松弛时间耗尽之前, 将函数排队积压, 直到等待的函数数量达到一个合适的批处理大小后再交给容器执行, 从而提高容器的利用率. Libra[116]通过性能分析预测函数的实际资源需求, 将用户过多分配的资源回收起来, “赠与”资源不足的函数使用, 从而提高整体资源利用率.

强化学习 (DRL) 算法是实现自动化资源调度的热门方向. Mampage 等人[117]使用强化学习来调度函数到合适的虚拟机上, 其状态空间由集群节点的资源使用统计和函数的预计负载特性组成, 以函数调度为动作, 根据应用性能和提供商成本优化目标定义奖励函数, 训练 DQN (deep Q-learning) 模型并实现函数的自动调度. Zafeiropoulos 等人[118]使用强化学习实现了基于指标的调度模式, 可以保持 CPU 使用率、内存使用率、吞吐量或延迟等指标在预定义的范围内. 算法以 CPU 和内存占用率、延迟时间、吞吐量等指标作为状态空间, 以扩展 CPU 和内存为动作, 以请求成功比例和延迟构造奖励函数, 训练 Q-learning 模型, 实现资源的自动伸缩. SFSchlr[119]在状态空间中纳入了函数的包和数据依赖关系信息, 训练 DRL 模型来控制函数和容器的调度, 模型会鼓励函数被安排在已载入相似包和数据的容器上, 从而提高启动速度. 在多租户环境下, 传统的单智能体强化学习算法会由于环境不稳定和相互干扰导致性能下降甚至收敛失败. 为了更好地适应多租户环境, SIMPPO[120]引入了多智能体强化学习算法, 每个智能体代表一个函数 (指函数定义, 非运行时函数) 进行训练, 使用增量在线学习来实现调度策略的自适应学习, 从而可以在运行时动态地学习和更新资源管理策略, 更好地适应动态环境和不确定性 (例如新增或更新函数).

Hermod[121]综合使用了 3 种分布式调度技术: 早期绑定、混合负载平衡和处理器共享调度. 早期绑定在低负载时使用, 在函数调用到达时立即分配给可用的节点, 而不是等待调度决策; 混合负载平衡是指在低负载时将函数集中调度给已经有可用容器的节点, 以实现高度的资源共享和低延迟, 在高负载情况下, 将函数调度给已经有热容器的节点, 以减少冷启动的影响; 处理器共享调度是指在节点让多个任务共享处理器资源. Hermod 在不同的场合使用不同的调度技术, 兼顾了性能和资源利用率.

5.1.2 保持负载均衡

负载均衡致力于将任务均匀分配到资源, 避免因资源利用倾斜导致的任务性能下降和资源闲置.

Sequoia[122]将函数链视为整体作为调度单位, 并在函数链内引入并发限制和队列管理等措施, 来保证函数链内的资源分配公平, 最终实现 QoS (quality of service, 服务质量) 目标. Atoll[96]将集群划分为多个“工作池”, 每个工作池由一个半全局调度器独占管理. 在工作池内, 每个函数都关联了一个延迟截止时间 (SLO 目标之一), 半全局调度器将按照最少剩余松弛时间优先算法来调度任务; 在工作池外, 多个负载均衡器负责将任务调度给合适的工作池, 并监控各个工作池的负载以扩展或收缩工作池的数量. 这种分组管理的模式在保证了高效调度的同时也避免了调度器瓶颈的出现.

5.1.3 提高任务性能

SFS[123]对 Linux 默认的调度器 CFS (completely fair scheduler) 进行改进. 由于 CFS 对工作负载一无所知, 经常在短函数之间进行切换, 而 SFS 使用两级调度来考虑短函数的优先级, 对短函数使用 FIFO, 对长函数继续使用 CFC, 从而最小化函数的平均执行时间. Cypress[124]通过将请求批处理和重新排序来最小化资源消耗并保证 SLO 合规. Cypress 基于函数的输入参数大小来估计函数的执行时间, 基于该时间估计, 一方面更好的估计每个容器能容纳的请求数量, 将请求合并批处理时可以减少容器的数量, 另一方面重新排序请求队列, 让具有更高执行时间的请求先被执行, 从而尽可能多的满足请求的 SLO. 此外, Cypress 还使用了链式预测 (chained prediction) 方法来预测下游函数的输入大小分布, 以便在不同的请求负载和输入大小分布下提前扩展容器, 从而提高 SLO 合规率.

数据本地性一直是分布式计算中影响性能的关键因素之一, 通过尽量将函数放置在同一容器或 VM 内, 可以有效地增加数据本地性, 加快函数的启动速度, 提高函数通信效率, 并可以共享操作系统缓存. XFaaS[90]将相同语言的函数都调度到相同节点上运行, 并且在节点之间共享 JIT 编译数据, 从而加速 JIT 代码的编译. Palette[125]在编程接口中引入“颜色”的概念, 由用户将需要在同一实例上执行的函数 (如社交网络对相同帖子的处理任务, 或 DAG 任务中连续执行的任务链等) 标记为同一颜色, Palette 会将同色函数放在同一 VM 实例上执行, 从而使这些函数可以共享缓存并加快数据传输. CH-RLU[126]使用一致性哈希算法在服务器之间分发函数, 以确保在服务器数

量伸缩变化时, 函数被重新分配的概率被降低到最低, 从而让一个函数尽可能地保留在同一服务器上执行, 避免数据的重复加载.

此外, 提前为任务准备资源也是常见的优化方案, 不过这方面的研究在第 3.2.2 节已经概述, 此处不再赘述.

5.1.4 异构资源调度

许多研究考虑服务器无感知计算在不同性能、不同计算模式、不同硬件, 甚至不同地理位置的计算资源之间的调度问题, 这些跨不同类型资源的调度技术可以在不同的场景下实现针对性的优势.

Icebreaker[94]通过混合使用高性能和低性能节点来提高函数的启动速度并同时控制整体计算成本. 它一方面收集函数历史运行数据, 使用模型分析函数的运行模式, 并预测函数未来的调用频率和概率; 另一方面考虑服务节点的加速比和内存使用情况, 评估其“效用分数 (utility score)”, 而后将高概率和高频率运行的函数调度到更高性能的节点上并进行预热, 从而提高函数的启动速度.

服务器无感知计算弹性好, 计费精细, 但费率高, 因此对用户而言, 通常在应对突发、低负载请求时划算, 持续、高负载请求时不划算. 对此, 阿里云的 Amoeba[127]提出将任务在 IaaS 和 FaaS 之间切换部署, 它通过监控任务的资源消耗和集群的资源争用情况, 预估将任务切换部署后的 QoS, 以决定切换部署的时机, 实现在低负载时候部署到 FaaS, 在高负载时切换到 IaaS, 从而一方面为用户节省成本, 另一方面也为服务商节省集群资源 (高负载下 FaaS 的反复启动会浪费很多资源). Mashup[128]在服务器无感知集群和 VM 集群之间调度任务, 通过对比任务在两个平台上的执行时间决定合适的放置策略. MArk[53]在包含 VM 和硬件加速器 (如 GPU) 在内的多种资源类型之间进行调度, 其中服务器无感知计算被用于应对偶发的请求高峰, 利用它快速伸缩的性能来填补传统 VM 集群无法及时扩容响应的时间空隙, 通过略高一点的价格保证了 SLO. Dorylus[45]同时使用传统 CPU 集群和服务器无感知计算来执行 GNN 训练, 它将其中的图计算任务交给 CPU 集群实现, 张量计算任务交给 Serverless 函数负责, 兼顾了系统性能和伸缩性.

GPU 调用一直是商业服务器无感知服务中缺少的功能, 在开源领域, 除了 Molecule[77]外, DGSF[129]也是一个支持 GPU 调用的系统. 与 Molecule 不同的是, DGSF 实现了运行时层 (CUDA) 的 GPU 虚拟化, 从而使多个函数可以共享使用远程 GPU. 为了实现这一点, DGSF 使用了多种技术, 包括 API 远程优化 (如批处理、预初始化)、负载平衡、实时迁移等. DGSF 是第一个实现了 GPU 共享服务的服务器无感知系统. 之后, FaST-GShare[130]针对深度学习推理任务, 实现了 GPU 的时空多路复用, 提高 GPU 利用率的同时还保证了服务水平目标 (SLOs).

Skippy[131]针对边缘计算场景优化容器调度技术, 它可以综合考虑各种因素, 包括节点的性能、硬件 (如 GPU) 和软件环境, 带宽和相互之间的地理距离等, 在移动数据和移动计算之间进行启发式权衡.

随着服务器无感知计算提供商的增多, 联邦计算的概念也开始进入到服务器无感知计算领域. 联邦计算是指将计算任务分发到多个不同的服务节点上进行并行处理的模式, 这些节点可以跨不同的云服务, 从而进一步增强系统的并行度和鲁棒性. 现有的工作主要集中于多个云之间的调度问题, 主流方法是基于各种性能指标预测函数在每个云上的性能表现来做出调度决策. 如 FaaSt[132]使用基于列表的调度方法, 在执行时首先遍历每个函数在列表内每个 FD (函数部署, 即云服务) 上的运行性能, 而后考虑各个 FD 的完成时间、往返时间和并发度, 为每个函数选择预期性能最好的 FD 进行部署. MPSC[133]在边缘云上实现跨服务商的调度, 基于监控各服务商的性能状态来调度应用程序, 它还允许用户自定义调度策略, 以实现个性化的调度模式.

5.2 面向用户的调度

对于服务器无感知计算的用户来说, 他们无需也无法接触底层的资源分配策略, 不过, 平台通常会允许用户在一定程度内调节函数的资源分配, 例如 AWS Lambda 允许用户调整函数的内存大小 (vCPU 数量和内存绑定调节), 从 128 MB 到 10 GB 自由选择, 并行函数数量从 0 至 1 000 可选 (普通账户), 平台将根据函数的实际执行时间和资源量分配量计费, 这就给了用户一定的资源调度自由度. 需要注意的是, 服务器无感知计算虽然具有高并发和高弹性的优点, 但却不一定更便宜, 当前的商业服务器无感知服务的单位时间定价普遍超过性能相近的 VM, 长时间运行的价格必然比不上高利用率的集群, 此外服务器无感知计算还可能产生其他服务的费用, 如 S3 读写, DynamoDB

使用等. 这就需要用户合理的安排计算流程, 分配计算资源, 以便在利用服务器无感知计算优势的同时控制计算成本. 当前, 面向用户的资源调度技术主要有两个目标, 提高性能和降低成本, 当然, 也有一些研究会兼顾两个目标.

5.2.1 提高性能

服务器无感知计算的资源配置虽然简单, 但仍不代表用户就能完美设置, 函数资源的过分配与欠分配都是常见的现象. Libra[116]在系统层回收过分配资源并“赠与”欠分配的函数; Freyr[134]在用户层帮助用户回收和补充资源, 它使用深度强化学习算法推测其实际需要的资源量, 对资源分配过多和过少的函数进行重分配.

如果将应用场景缩小到特定的任务类型, 那么就可以根据任务本身的特点进行优化. Caerus[135]致力于提高数据分析任务执行的并行度, 相较于传统的基于阶段 (Stage) 的调度方法, Caerus 使用了更细粒度的基于步骤 (Step, 可以包含多个函数) 的调度方式, 使用更细粒度的“步骤” (从执行计划中获取或由用户从代码指定) 来描述任务内部的依赖关系, 通过使用流水线方式执行步骤, 挖掘出任务内部的并行潜力. 北京大学的 Ditto[136]同样针对数据分析类任务, 它将阶段捆绑成组进行调度. Ditto 使用基于实测数据拟合出来的时间模型, 预测阶段在不同并行度下的执行时间, 并通过调整各个阶段的并行度, 平衡 DAG 中不同路径的执行时间, 从而最小化关键路径的执行时间. 实验证明, Ditto 的性能超过了 Caerus. StepConf[137]也使用了调整并行度的思路, 它将函数输入数据切分为多个子集, 调用函数内多个进程同时执行函数逻辑, 同时也并行启动多个函数来处理不同的数据部分, 实现了函数间和函数内的并行, StepConf 主要针对视频处理任务. 需要注意的是, 以上优化技术只能针对特定的任务类型, 而不适用于其他任务类型.

FaaSLight[138]通过将非必要的代码推迟载入来加快函数启动速度. 它通过静态代码分析构建代码的调用图, 从图中识别出不可或缺代码和其他代码, 将其他代码分离并压缩为轻量级文件, 待需要时再按需加载. FaaSLight 完全在用户层工作, 完全等效地修改函数代码, 并且可以简单地适配各个服务器无感知平台.

5.2.2 降低成本

Akhtar[139]的研究中发现, 在 AWS Lambda 平台上运行的任务, 总体费用并不是一直随着资源分配的上升而上升的. 这是因为在一定范围内, 随着内存分配的增加, 虽然单位时间的费用增加了, 但函数的运行时间却缩短了, 反而减少了总费用. 经过观察发现, 费用随内存大小变化的曲线存在一个低谷, 而低谷的位置即不在最小资源处也不在最大资源的方向, 如图 8 所示. 许多研究都致力于寻找这个低谷的位置.

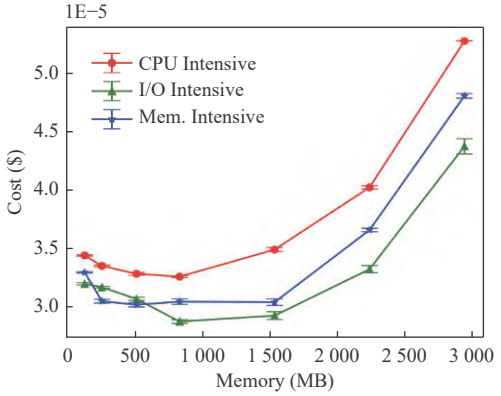

图 8 AWS Lambda 上各类型任务的运行费用和内存分配之间的关系[139]

贝叶斯优化是一种寻找黑盒函数极值的常用算法, 也是可以用于成本管理的有效方法. 该算法通过监控任务的执行效果, 不断调整资源配置, 反复尝试、逼近, 就可以找到可能的价格 (局部) 最低点位置. COSE[139,140]和 SMLT[40]都使用了贝叶斯优化算法来寻找最佳的资源配置方案, 其中 COSE 的算法可以适用于 AWS Lambda 和其他商业平台以及私有云框架, 而 SMLT[40]可以基于服务级别目标 (SLO) 进行优化.

贝叶斯优化需要在任务执行中不断尝试以寻找最佳配置, 尝试的时间会导致一定的资源浪费, 因此一些研究

使用了预训练的性能模型. StepConf[137]在任务执行前使用曲线拟合的方法, 离线地预测函数的性能和成本. 在任务正式执行时, 同样使用启发式算法, 基于实时负载情况和 SLO 设置, 找到最佳的内存大小配置并动态调整. Sizeless[141]不需要针对任务进行离线测试, 它预训练了一个性能预测模型, 该模型包含大量典型函数的性能表现特征. 基于该模型, Sizeless 可以仅依靠函数在一个内存配置下的执行性能就预测出在其他配置下的表现, 从而更快的为用户找出最佳的内存配置, 不过这也失去了一些在运行中动态调整内存配置的灵活性. λDNN[44]预训练了一个针对 DNN 任务的轻量级性能分析模型, 它使用代表性的 DNN 模型在代表性的数据集上进行测试训练, 基于这些训练数据再训练出性能分析模型, 用于预测任务的执行时间, 从而能在未来的任务执行中, 从多种函数资源配置方案中找出最经济的方案, 在保证性能的前提下节省计算成本.

Siren[41]则使用了深度强化学习 (DRL) 算法来决定机器学习任务中资源的分配, 它以机器学习任务中每个 Epoch 所消耗的时间、损失函数值、平均内存和 CPU 占用等数据作为状态空间, 函数的并行数量和内存大小调整作为动作, Epoch 执行时间和预算消耗作为奖励函数, 不断进行迭代训练, 使模型可以做出整体性价比趋向最高的调度决策, 并在 Epoch 之间动态调整调度策略. Astrea[142]可以根据用户指定的目标, 自动安排任务的资源分配和并行度方案. 它的算法基于图论中的 DAG 模型, 分别解决了两个优化问题: 在预算约束下, 最小化作业完成时间; 在服务质量 (QoS) 约束下, 最小化运行成本. Astrea 能够平衡作业完成时间和计算成本, 从而实现更优秀的执行方案. Parrotfish[143]使用在线参数回归算法来寻找最划算的资源配置, 它支持用户设置时间和成本上的约束, 并自动求解出合适的配置. MLLess[42,43]通过缩减并行函数数量来节约机器学习训练任务的成本, 它将函数并行实例个数向模型的快速收敛时间区间倾斜, 当收敛曲线开始变平时则减少函数实例的数量, 以节省计算成本.

ORION[91]同时使用了 3 种手段以提高任务性能和资源利用率: 1) 通过性能模型和优化算法来找到每个函数的最佳资源配置; 2) 将多个相互并行的函数合并到一个资源较多的 VM/Lambda 执行, 以充分利用资源; 3) 根据对 DAG 控制流的分析, 提前预热即将被执行的函数. ORION 对特定服务器无感知平台的依赖不高, 在 AWS Lambda 和 Microsoft Azure Function 上都可以部署执行.

除了在有限的资源调度自由度上来挖掘优化潜力之外, 批处理方式也是一种有效降低成本的方案. 通过将多个函数打包为一个大函数执行, 可以降低实际的函数并发数量, 减少函数启动开销, 从而降低成本. 如 Batch[47]使用批处理模式执行机器学习推理任务; INFless[48]则使用非均匀批处理策略, 根据推理模型的资源需求 SLO 动态选择不同的批处理大小和资源配置; 而 ProPack[144]通过性能和成本建模来选择最合适的“打包程度”, 即多少个函数被打包到一起, 才能实现性能和成本的最优化.

5.3 对比与小结

表 4 总结了本节提到的所有系统并做出了比较. 当前面向平台的资源调度基本还是沿用传统集群内的调度思路和技术, 这也足以解决绝大部分的问题, 未来的研究可以关注一些仍然没有应用到服务器无感知集群内的调度技术, 例如异构资源的调度优化. 当前在支持 GPU 的服务器无感知计算系统中, GPU 和 CPU 都是独立分开调度, 在二者的协同调度方面, 仍可进一步探索, 其中值得关注的点包括但不限于如下几个方面.

(1) 任务划分和资源分配: 当前研究在任务划分上基本都是由用户指定, 以函数为单位, 划分给指定类型的容器. 对于一些复杂的函数, 可能某一部分适合 CPU 计算, 另一部分适合 GPU 计算, 如何为其分配合适的 CPU 和 GPU 资源, 或者在执行中动态调整资源分配, 是当前尚未解决的问题.

(2) 数据传输优化: 由于 CPU 和 GPU 拥有各自独立的内存, 在他们之间互传数据可能导致高昂的传输成本. 如何根据任务的特点降低数据传输频率并优化传输速度, 是协同调度中的一个关键点.

(3) 同步问题: CPU 和 GPU 承担的任务并不是相互独立的, 一些任务可能会同时使用两种资源, 在执行时要保证两种资源的计算相互同步, 减少或避免任务出现阻塞, 这也是调度方法中需要考虑的问题.

(4) 动态负载均衡: 对 CPU 和 GPU 两种硬件群的调度, 比单一硬件群的调度要更为复杂, 一些任务可能会同时使用两种硬件, 需要在这种情况下保持系统整体性能最优, 这对于负载均衡技术是一个挑战.

表 4 本节概述的所有系统及比较

系统和工作	基于FaaS系统	SLO保证	调度单位	调度策略	主要技术内容
Fifer[95]	—	√	函数	批处理	利用“松弛时间”累积批处理分组
Libra[116]	OpenWhisk[7]	×	集群资源	性能建模	回收过分配的资源并赠与欠分配的函数
Mampage等人[117]	Kubeless[12]	×	函数	DRL	使用强化学习算法调度资源
Zafeiropoulos等人[118]	Kubeless[12]	√	集群资源	DRL	使用强化学习算法控制资源伸缩
SFSchlr[119]	—	×	函数、容器	DRL	考虑包和数据依赖的DRL调度方案
SIMPPO[120]	OpenWhisk[7]	√	集群资源	DRL	使用多智能体RL控制资源伸缩
Hermod[121]	OpenWhisk[7]	×	函数	启发式	综合使用3种调度技术
Sequoia[122]	—	√	函数链	启发式	将函数链视为整体作为调度单位
SFS[123]	多平台	×	函数	启发式	使用两级调度, 对长短函数分开调度
Cypress[124]	OpenFaaS[9]	√	函数、容器	启发式	通过请求批处理和重新排序来保证SLO
Atoll[96]	—	√	函数	启发式	保证SLO和负载均衡的二级调度方法
Palette[125]	Azure Functions Host	×	函数	启发式	以“颜色”的概念增加任务本地性
CH-RLU[126]	OpenWhisk[7]	×	函数	启发式	使用一致性哈希算法分发函数
Icebreaker[94]	OpenWhisk[7]	×	函数	启发式	混合使用高性能和低性能节点进行调度
Amoeba[127]	OpenWhisk[7]	√	函数	启发式	在IaaS和FaaS之间切换部署任务
Mashup[128]	AWS	×	任务	启发式	在PaaS和FaaS之间切换部署任务
MArk[53]	AWS	√	函数	启发式	综合使用不同类型的云服务来执行任务
Dorylus[45]	AWS Lambda	×	函数	启发式	同时使用VM和FaaS进行GNN训练
DGSF[129]	OpenFaaS[9]	×	函数	—	使用虚拟层实现透明的共享GPU服务
FaST-GShare[130]	OpenFaaS[9]	√	函数	启发式	深度学习推理任务的GPU时空复用
Skippy[131]	OpenFaaS[9]	×	函数	启发式	在边缘场景调度任务到最佳的节点
FaaSt[132]	多平台	×	函数	基于列表	跨多个云选择预期性能最好的部署位置
MPSC[133]	多平台	×	函数	用户定义	允许自定义策略的边缘多云调度框架
Freyr[134]	OpenWhisk[7]	√	单函数资源	DRL	基于DRL的函数资源回收和重分配
Caerus[135]	Pywren[26] + Jiffy[108]	×	步骤 (Step)	流水线	挖掘出任务内部并行潜力并流水线执行
Ditto[136]	SPRIGHT[114]	×	阶段组	性能建模	将阶段合并成组进行调度
StepConf[137]	AWS Lambda	√	函数数量 单函数资源	性能建模	挖掘函数间和函数内的并行能力, 动态调整函数资源配置
FaaSLight [138]	多平台	×	函数代码	—	将函数的非必要代码分离并推迟载入
COSE[139,140]	多平台	×	单函数资源	贝叶斯优化	使用贝叶斯优化寻找函数最佳内存大小
SMLT[40]	AWS Lambda	√	函数数量 单函数资源	贝叶斯优化	使用贝叶斯优化寻找函数最佳内存大小和并行数量, 并满足SLO和最小化成本
Sizeless[141]	AWS Lambda	×	函数内存	ML	使用预训练的模型选择最佳内存配置
λDNN[44]	AWS Lambda	√	单函数资源	性能建模	基于性能模型选择最经济资源配置
Astrea[142]	AWS Lambda	√	函数数量 单函数资源	DAG模型	根据用户指定的目标, 兼顾时间和成本自动安排任务的执行方案
Parrotfish[143]	不限	√	单函数资源	参数回归	使用参数回归寻找最划算的资源配置
Siren[41]	AWS Lambda	×	函数数量 单函数资源	DRL	使用深度强化学习算法决定函数最佳内存大小和并行数量
MLLess[42,43]	IBM-PyWren[27]	×	函数数量	启发式	逐渐缩减函数数量以节约成本
ORION[91]	多平台	√	函数	性能建模 启发式	通过性能建模选择资源配置, 并行函数合并执行, 根据控制流提前预热函数
Batch[47]	OpenFaaS[9]	√	函数	批处理	使用批处理执行ML推理任务
INFless[48]	OpenFaaS[9]	√	函数	批处理	使用非均匀批处理执行ML推理任务
ProPack[144]	FuncX	×	函数	批处理	通过建模选择最合适的函数“打包程度”

在面向用户的资源调度方面, 商业平台的资源配置自由度很小, 调度空间不大, 不过仍可以获得一定性能收益. 未来的研究, 可以更多地聚焦于针对特定任务类型的调度策略, 通过合理安排函数的执行流程, 获得更好的特定任务运行效率和成本效率.

在资源管理方面, Mampage[25]做了针对性的研究综述, 具有较好的参考价值.

6 总结与展望

6.1 总 结

服务器无感知计算是新兴的云计算模式, 它提出了函数即服务 (FaaS) 的计算模式, 实现了高效的并发和伸缩, 并为用户屏蔽了底层资源管理的细节, 同时, 也带来了许多自由度限制. 自服务器无感知计算诞生以来, 研究人员一方面试图挖掘、发挥它在各个领域上的优势, 一方面也不断尝试改进、突破它在各个方面的限制. 性能优化一直是服务器无感知计算研究的核心问题, 本文从几个方面对现有的研究工作做出了综述.

(1) 面向典型任务的优化, 包括服务器无感知计算在面向各类典型任务类型时的适配和优化工作, 其中以机器学习类最多. 大部分的任务都可以从服务器无感知计算中收益, 但它们也或多或少都做出了功能或规模上的妥协, 这主要是因为服务器无感知计算中各种资源和通信的限制.

(2) 沙箱环境优化, 即围绕执行 Serverless 函数的隔离环境的各类优化技术, 包括各个抽象层次的沙箱方案, 以及对冷启动问题的优化研究. 在各类沙箱方案中, 抽象层次越高则性能越高, 但函数的隔离性越差, 权衡性能和安全性, 容器是当前最主流的方案. 各类冷启动研究使用了各个抽象层次的资源重用和预载入技术, 最快已经可以实现亚毫秒级的启动速度.

(3) 通信 I/O 优化, 包括针对服务器无感知计算在 I/O 和通信性能优化方面的研究, 现有方案一方面基于缓存优化 I/O 吞吐效率, 一方面也尝试各种途径构建函数间的通信渠道.

(4) 资源调度优化, 包括在平台层和用户层针对服务器无感知计算的资源或任务调度优化技术. 其中平台层的调度方法基本与传统调度技术相似, 用户层的调度自由度不大, 但仍可以节省成本和提高特定类型任务的执行效率.

本文综述了以上 4 个方面的性能优化技术, 共概括了 107 个相关技术和系统, 归纳了领域内主要面临的问题和挑战. 本文中提到所有系统及其相互之间的层次关系以及各自所解决的问题全部汇总在后文图 9 中. 部分系统同时使用了多种优化技术, 在图中以交叉框图的形式表示.

6.2 展 望

服务器无感知计算仍是非常有前途的技术模式, 无论在上层应用或是底层优化都仍有许多工作可以研究. 本文认为, 服务器无感知计算的性能优化技术未来值得关注的研究方向主要有以下几个.

(1) 更多任务类型在服务器无感知计算环境下的适配. 研究报告[145]指出, 大部分的计算任务都可以被拆解成小型任务解决, 当前迁移到服务器无感知平台的计算任务仍属少数, 仍有大量的任务类型可以迁移到服务器无感知环境, 例如图计算、各种领域计算任务等.

(2) 机器学习任务的优化. 虽然机器学习是被迁移最多的任务类型, 但其实实现的模型仍很不全面, 尤其是训练任务, 复杂模型要么无法实现, 要么规模受限, 如深度学习、图计算任务的适配都还处于初级阶段.

(3) 针对特定任务类型服务器无感知计算优化, 如特定类型数据的分析处理、特定类型的机器学习任务等, 优化目标包括但不限于沙箱优化、I/O 优化、调度优化、成本优化等. 针对特定类型的任务可以挖掘出更多的优化潜力, 有望对系统性能产生显著的提升.

(4) 异构硬件资源的支持和优化. 截至本文发稿前, 各大商业服务器无感知计算平台尚未支持 GPU 或其他硬件加速器的调用, 这可能是由于异构硬件资源调度会影响函数的伸缩性, 抑或是为了保持服务器无感知计算“轻量级”的特点. 不过, 服务器无感知计算未来必然要走向更多类型资源的服务模式. 当前 Molecule[77]和 DGSF[129]已经

可以实现较为完整的 GPU 使用支持, 不过它们都是将 GPU 和 CPU 分开调度, 而 CPU 和 GPU 协同调度的技术暂时没有出现, 值得进一步探索.

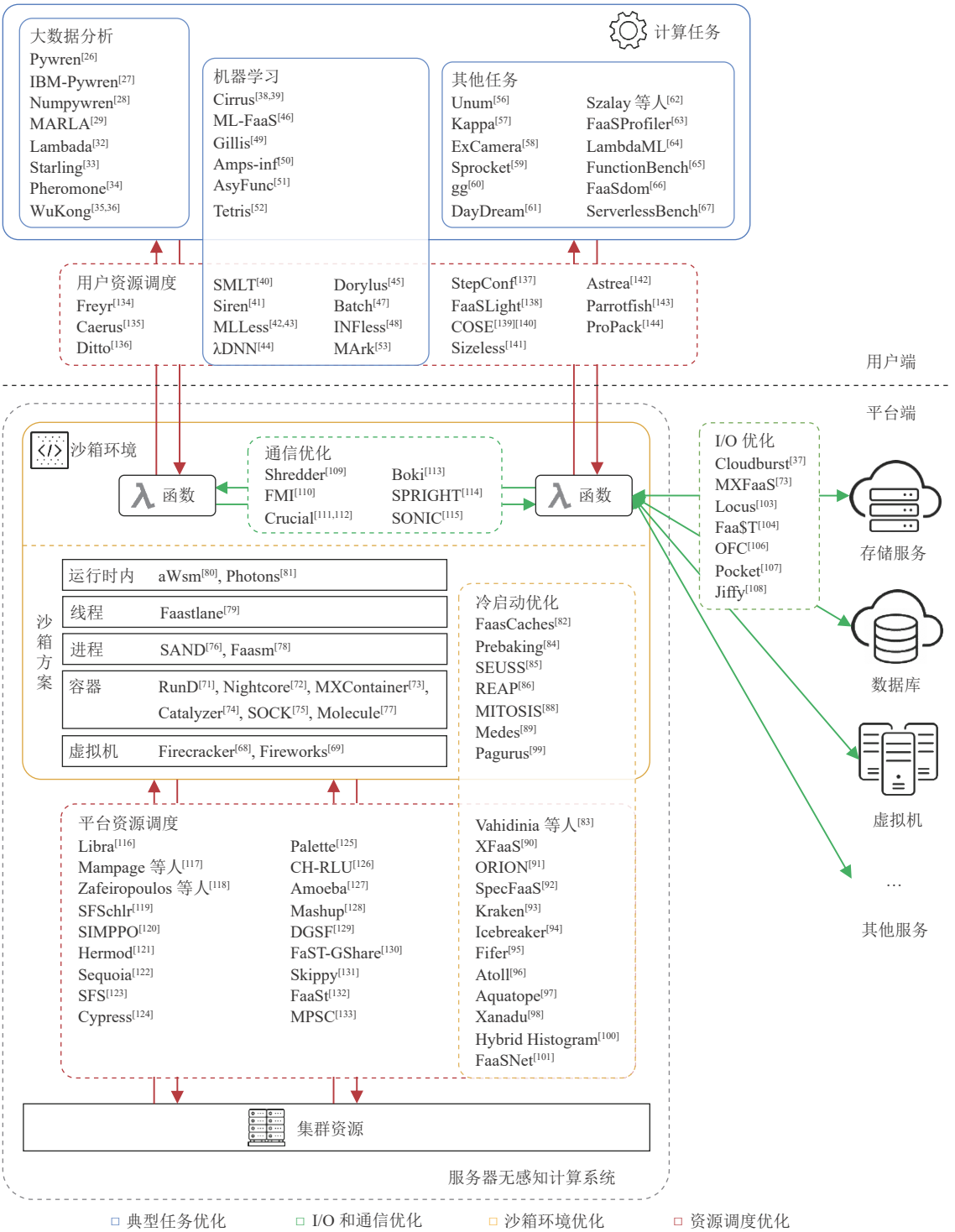

图 9 本文概述的所有系统

References:

[1] AWS Lambda. Serverless Function, FaaS Serverless. https://aws.amazon.com/lambda

[2] Microsoft Azure. Azure Functions—Serverless Functions in Computing. https://azure.microsoft.com/en-us/services/functions

[3] Cloud Functions | Google Cloud. https://cloud.google.com/functions

[4] IBM Cloud Functions. https://www.ibm.com/cloud/functions

[5] Aliyun Function Compute (in Chinese). https://www.aliyun.com/product/fc

[6] Baidu Cloud Function Compute (in Chinese). https://cloud.baidu.com/product/cfc.html

[7] Apache OpenWhisk. Open source serverless cloud platform. http://openwhisk.apache.org

[8] Hendrickson S, Sturdevant S, Harter T, Venkataramani V, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Serverless computation with OpenLambda. In: Proc. of the 8th USENIX Conf. on Hot Topics in Cloud Computing. Denver: USENIX Association, 2016. 33–39.

[9] OpenFaaS. Serverless functions, made simple. https://www.openfaas.com

[10] GitHub. Fission/fission: Fast and simple serverless functions for Kubernetes. https://github.com/fission/fission

[11] Knative. https://knative.dev/

[12] GitHub. vmware-archive/kubeless: Kubernetes native serverless framework. https://kubeless.io

[13] Passwater A. 2018 Serverless Community Survey: Huge growth in serverless usage. 2018. https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/

[14] Villamizar M, Garcés O, Ochoa L, Castro H, Salamanca L, Verano M, Casallas R, Gil S, Valencia C, Zambrano A, Lang M. Infrastructure cost comparison of running Web applications in the cloud using AWS Lambda and monolithic and microservice architectures. In: Proc. of the 16th IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid). Cartagena: IEEE, 2016. 179–182. [doi: 10.1109/CCGrid.2016.37]

[15] Jonas E, Schleier-Smith J, Sreekanti V, Tsai CC, Khandelwal A, Pu QF, Shankar V, Carreira J, Krauth K, Yadwadkar N, Gonzalez JE, Popa RA, Stoica I, Patterson DA. Cloud programming simplified: A Berkeley view on serverless computing. arXiv:1902.03383, 2019.

[16] Baldini I, Castro P, Chang K, Cheng P, Fink S, Ishakian V, Mitchell N, Muthusamy V, Rabbah R, Slominski A, Suter P. Serverless computing: Current trends and open problems. In: Chaudhary S, Somani G, Buyya R, eds. Research Advances in Cloud Computing. Singapore: Springer, 2017. 1–20. [doi: 10.1007/978-981-10-5026-8_1]

[17] Hellerstein JM, Faleiro J, Gonzalez JE, Schleier-Smith J, Sreekanti V, Tumanov A, Wu CG. Serverless computing: One step forward, two steps back. In: Proc. of the 9th Biennial Conf. on Innovative Data Systems Research. Asilomar: www.cidrdb.org, 2019.

[18] Khandelwal A, Kejariwal A, Ramasamy K. Le taureau: Deconstructing the serverless landscape & a look forward. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 2641–2650. [doi: 10.1145/3318464.3383130]

[19] Shafiei H, Khonsari A, Mousavi P. Serverless computing: A survey of opportunities, challenges, and applications. ACM Computing Surveys, 2022, 54(11s): 239. [doi: 10.1145/3510611]

[20] Li YK, Lin YY, Wang Y, Ye KJ, Xu CZ. Serverless computing: State-of-the-art, challenges and opportunities. IEEE Trans. on Services Computing, 2023, 16(2): 1522–1539. [doi: 10.1109/TSC.2022.3166553]

[21] Li ZJ, Guo LS, Cheng JG, Chen Q, He BS, Guo MY. The serverless computing survey: A technical primer for design architecture. ACM Computing Surveys, 2022, 54(10s): 220. [doi: 10.1145/3508360]

[22] Barrak A, Petrillo F, Jaafar F. Serverless on machine learning: A systematic mapping study. IEEE Access, 2022, 10: 99337–99352. [doi: 10.1109/ACCESS.2022.3206366]

[23] Kjorveziroski V, Filiposka S. WebAssembly as an enabler for next generation serverless computing. Journal of Grid Computing, 2023, 21(3): 34. [doi: 10.1007/s10723-023-09669-8]

[24] Golec M, Walia GK, Kumar M, Cuadrado F, Gill SS, Uhlig S. Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. arXiv:2310.08437, 2023.

[25] Mampage A, Karunasekera S, Buyya R. A holistic view on resource management in serverless computing environments: Taxonomy and future directions. ACM Computing Surveys, 2022, 54(11s): 222. [doi: 10.1145/3510412]

[26] Jonas E, Pu QF, Venkataraman S, Stoica I, Recht B. Occupy the cloud: Distributed computing for the 99%. In: Proc. of the 2017 Symp. on Cloud Computing. Santa Clara: ACM, 2017. 445–451. [doi: 10.1145/3127479.3128601]

[27] Sampé J, Vernik G, Sánchez-Artigas M, García-López P. Serverless data analytics in the IBM cloud. In: Proc. of the 19th Int'l Middleware Conf. Industry. Rennes: ACM, 2018. 1–8. [doi: 10.1145/3284028.3284029]

[28] Shankar V, Krauth K, Vodrahalli K, Pu QF, Recht B, Stoica I, Ragan-Kelley J, Jonas E, Venkataraman S. Serverless linear algebra. In: Proc. of the 11th ACM Symp. on Cloud Computing. Virtual Event: ACM, 2020. 281–295. [doi: 10.1145/3419111.3421287]

[29] Giménez-Alventosa V, Moltó G, Caballer M. A framework and a performance assessment for serverless MapReduce on AWS Lambda.

Future Generation Computer Systems, 2019, 97: 259–274. [doi: 10.1016/j.future.2019.02.057]
[30] ooso: Java library for running Serverless MapReduce jobs. https://github.com/d2si-oss/ooso
[31] A serverless MapReduce framework written for AWS Lambda. 2018. https://github.com/bcongdon/corral
[32] 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]
[33] Perron M, Castro Fernandez R, DeWitt D, Madden S. Starling: A scalable query engine on cloud functions. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 131–141. [doi: 10.1145/3318464.3380609]
[34] Yu MC, Cao TJ, Wang W, Chen RC. Following the data, not the function: Rethinking function orchestration in serverless computing. In: Proc. of the 20th USENIX Symp. on Networked Systems Design and Implementation. Boston: USENIX Association, 2023. 1489–1504.
[35] Carver B, Zhang JY, Wang A, Cheng Y. In search of a fast and efficient serverless DAG engine. In: Proc. of the 4th IEEE/ACM Int'l Parallel Data Systems Workshop (PDSW). Denver: IEEE, 2019. 1–10. [doi: 10.1109/PDSW49588.2019.00005]
[36] Carver B, Zhang JY, Wang A, Anwar A, Wu PR, Cheng Y. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In: Proc. of the 11th ACM Symp. on Cloud Computing. Virtual Event: ACM, 2020. 1–15. [doi: 10.1145/3419111.3421286]
[37] Sreekanti V, Wu CG, Lin XC, Schleier-Smith J, Gonzalez JE, Hellerstein JM, Tumanov A. Cloudburst: Stateful functions-as-a-service. Proc. of the VLDB Endowment, 2020, 13(12): 2438–2452. [doi: 10.14778/3407790.3407836]
[38] Carreira J, Fonseca P, Tumanov A, Zhang A, Katz R. A case for serverless machine learning. In: Proc. of the 2018 Workshop on Systems for ML and Open Source Software at NeurIPS. 2018.
[39] Carreira J, Fonseca P, Tumanov A, Zhang A, Katz R. Cirrus: A serverless framework for end-to-end ML workflows. In: Proc. of the 2019 ACM Symp. on Cloud Computing. Santa Cruz: ACM, 2019. 13–24. [doi: 10.1145/3357223.3362711]
[40] Ali A, Zawad S, Aditya P, Akkus IE, Chen RC, Yan F. SMLT: A serverless framework for scalable and adaptive machine learning design and training. arXiv:2205.01853, 2022.
[41] Wang H, Niu D, Li BC. Distributed machine learning with a serverless architecture. In: Proc. of the 2019 IEEE Conf. on Computer Communications. Paris: IEEE, 2019. 1288–1296. [doi: 10.1109/INFOCOM.2019.8737391]
[42] Sánchez-Artigas M, Sarroca PG. Experience paper: Towards enhancing cost efficiency in serverless machine learning training. In: Proc. of the 22nd Int'l Middleware Conf. Québec City: ACM, 2021. 210–222. [doi: 10.1145/3464298.3494884]
[43] Gimeno Sarroca P, Sánchez-Artigas M. MLLESS: Achieving cost efficiency in serverless machine learning training. Journal of Parallel and Distributed Computing, 2024, 183: 104764. [doi: 10.1016/j.jpdc.2023.104764]
[44] Xu F, Qin YL, Chen L, Zhou Z, Liu FM. λDNN: Achieving predictable distributed dnn training with serverless architectures. IEEE Trans. on Computers, 2022, 71(2): 450–463. [doi: 10.1109/TC.2021.3054656]
[45] Thorpe J, Qiao YF, Eyolfson J, Teng S, Hu GZ, Jia ZH, Wei JL, Vora K, Netravali R, Kim M, Xu GH. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In: Proc. of the 15th USENIX Symp. on Operating Systems Design and Implementation. USENIX Association, 2021. 459–5140.
[46] Paraskevoulakou E, Kyriazis D. ML-FaaS: Toward exploiting the serverless paradigm to facilitate machine learning functions as a service. IEEE Trans. on Network and Service Management, 2023, 20(3): 2110–2123. [doi: 10.1109/TNSM.2023.3239672]
[47] Ali A, Pinciroli R, Yan F, Smirni E. BATCH: Machine learning inference serving on serverless platforms with adaptive batching. In: Proc. of the 2020 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Atlanta: IEEE, 2020. 1–15. [doi: 10.1109/SC41405.2020.00073]
[48] Yang YN, Zhao LP, Li YM, Zhang HY, Li J, Zhao MY, Chen XZ, Li KQ. INFless: A native serverless system for low-latency, high-throughput inference. In: Proc. of the 27th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2022. 768–781. [doi: 10.1145/3503222.3507709]
[49] Yu MC, Jiang ZF, Ng HC, Wang W, Chen R, Li B. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In: Proc. of the 41st IEEE Int'l Conf. on Distributed Computing Systems (ICDCS). Washington: IEEE, 2021. 138–148. [doi: 10.1109/ICDCS51616.2021.00022]
[50] Jarachanthan J, Chen L, Xu F, Li B. AMPS-Inf: Automatic model partitioning for serverless inference with cost efficiency. In: Proc. of the 50th Int'l Conf. on Parallel Processing. Lemont: ACM, 2021. 14. [doi: 10.1145/3472456.3472501]
[51] Pei QY, Yuan YJ, Hu HC, Chen Q, Liu FM. AsyFunc: A high-performance and resource-efficient serverless inference system via asymmetric functions. In: Proc. of the 2023 ACM Symp. on Cloud Computing. Santa Cruz: ACM, 2023. 324–340. [doi: 10.1145/3620678.3624664]
[52] Li J, Zhao LP, Yang YA, Zhan KL, Li KQ. Tetris: Memory-efficient serverless inference through tensor sharing. In: Proc. of the 2022 USENIX Annual Technical Conf. Carlsbad: USENIX Association, 2022.

[53] Zhang CL, Yu MC, Wang W, Yan F. MArk: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In: Proc. of the 2019 USENIX Annual Technical Conf. Renton: USENIX Association, 2019. 1049–1062.

[54] AWS Step Functions. https://aws.amazon.com/cn/step-functions/

[55] Azure Durable Functions. https://learn.microsoft.com/azure/azure-functions/durable/durable-functions-overview

[56] Liu DH, Levy A, Noghabi SA, Burckhardt S. Doing more with less: Orchestrating serverless applications without an orchestrator. In: Proc. of the 20th USENIX Symp. on Networked Systems Design and Implementation. Boston: USENIX Association, 2023. 1505–1519.

[57] Zhang W, Fang V, Panda A, Shenker S. Kappa: A programming framework for serverless computing. In: Proc. of the 11th ACM Symp. on Cloud Computing. Virtual Event: ACM, 2020. 328–343. [doi: 10.1145/3419111.3421277]

[58] 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.

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

[60] Fouladi S, Romero F, Iter D, Li Q, Chatterjee S, Kozyrakis C, Zaharia M, Winstein K. From laptop to Lambda: Outsourcing everyday jobs to thousands of transient functional containers. In: Proc. of the 2019 USENIX Annual Technical Conf. Renton: USENIX Association, 2019. 475–488.

[61] Roy RB, Patel T, Tiwari D. DayDream: Executing dynamic scientific workflows on serverless platforms with hot starts. In: Proc. of the 2022 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Dallas: IEEE, 2022. 1–18. [doi: 10.1109/SC41404.2022.00027]

[62] Szalay M, Mátray P, Toka L. Real-time FaaS: Towards a latency bounded serverless cloud. IEEE Trans. on Cloud Computing, 2023, 11(2): 1636–1650. [doi: 10.1109/TCC.2022.3151469]

[63] Shahrad M, Balkind J, Wentzlaff D. Architectural implications of function-as-a-service computing. In: Proc. of the 52nd Annual IEEE/ACM Int'l Symp. on Microarchitecture. Columbus: ACM, 2019. 1063–1075. [doi: 10.1145/3352460.3358296]

[64] 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. Virtual Event: ACM, 2021. 857–871. [doi: 10.1145/3448016.3459240]

[65] 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]

[66] 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]

[67] 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. Virtual Event: ACM, 2020. 30–44. [doi: 10.1145/3419111.3421280]

[68] Agache A, Brooker M, Florescu A, Iordache A, Liguori A, Neugebauer R, Piwonka P, Popa DM. Firecracker: Lightweight virtualization for serverless applications. In: Proc. of the 17th USENIX Conf. on Networked Systems Design and Implementation. Santa Clara: USENIX Association, 2020. 419–434.

[69] Shin W, Kim WH, Min C. Fireworks: A fast, efficient, and safe serverless framework using VM-level post-JIT snapshot. In: Proc. of the 17th European Conf. on Computer Systems. Rennes: ACM, 2022. 663–677. [doi: 10.1145/3492321.3519581]

[70] 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.

[71] Li ZJ, Cheng JG, Chen Q, Guan EY, Bian ZZ, Tao Y, Zha B, Wang Q, Han WD, Guo MY. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In: Proc. of the 2022 USENIX Annual Technical Conf. Carlsbad: USENIX Association, 2022. 53–68.

[72] Jia ZP, Witchel E. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In: Proc. of the 26th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Virtual: ACM, 2021. 152–166. [doi: 10.1145/3445814.3446701]

[73] Stojkovic J, Xu TY, Franke H, Torrellas J. MXFaaS: Resource sharing in serverless environments for parallelism and efficiency. In: Proc. of the 50th Annual Int'l Symp. on Computer Architecture. Orlando: ACM, 2023. 34. [doi: 10.1145/3579371.3589069]

[74] Du D, Yu TY, Xia YB, Zang BY, Yan GL, Qin CG, Wu QX, Chen HB. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In: Proc. of the 25th Int'l Conf. on Architectural Support for Programming Languages and Operating

Systems. Lausanne: ACM, 2020. 467–481. [doi: 10.1145/3373376.3378512]

[75] Oakes E, Yang L, Zhou D, Houck K, Harter T, Arpaci-Dusseau AC, Arpaci-Dusseau RH. SOCK: Rapid task provisioning with serverless-optimized containers. In: Proc. of the 2018 USENIX Annual Technical Conf. Boston: USENIX Association, 2018. 57–69.

[76] Akkus IE, Chen RC, Rimac I, Stein M, Satzke K, Beck A, Aditya P, Hilt V. SAND: Towards high-performance serverless computing. In: Proc. of the 2018 USENIX Annual Technical Conf. Boston: USENIX Association, 2018. 923–935.

[77] Du D, Liu QY, Jiang XQ, Xia YB, Zang BY, Chen HB. Serverless computing on heterogeneous computers. In: Proc. of the 27th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2022. 797–813. [doi: 10.1145/3503222.3507732]

[78] Shillaker S, Pietzuch P. Faasm: Lightweight isolation for efficient stateful serverless computing. In: Proc. of the 2020 USENIX Annual Technical Conf. USENIX Association, 2020. 28.

[79] Kotni S, Nayak A, Ganapathy V, Basu A. Faastlane: Accelerating function-as-a-service workflows. In: Proc. of the 2021 USENIX Annual Technical Conf. USENIX Association, 2021. 805–820.

[80] Gadepalli PK, Peach G, Cherkasova L, Aitken R, Parmer G. Challenges and opportunities for efficient serverless computing at the edge. In: Proc. of the 38th Symp. on Reliable Distributed Systems (SRDS). Lyon: IEEE, 2019. 261–266. [doi: 10.1109/SRDS47363.2019.00036]

[81] Dukic V, Bruno R, Singla A, Alonso G. Photons: Lambdas on a diet. In: Proc. of the 11th ACM Symp. on Cloud Computing. ACM, 2020. 45–59. [doi: 10.1145/3419111.3421297]

[82] Fuerst A, Sharma P. FaasCache: Keeping serverless computing alive with greedy-dual caching. In: Proc. of the 26th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. ACM, 2021. 386–400. [doi: 10.1145/3445814.3446757]

[83] Vahidinia P, Farahani B, Aliee FS. Mitigating cold start problem in serverless computing: A reinforcement learning approach. IEEE Internet of Things Journal, 2023, 10(5): 3917–3927. [doi: 10.1109/JIOT.2022.3165127]

[84] Silva P, Fireman D, Pereira TE. Prebaking functions to warm the serverless cold start. In: Proc. of the 21st Int'l Middleware Conf. Delft: ACM, 2020. 1–13. [doi: 10.1145/3423211.3425682]

[85] Cadden J, Unger T, Awad Y, Dong H, Krieger O, Appavoo J. SEUSS: Skip redundant paths to make serverless fast. In: Proc. of the 15th European Conf. on Computer Systems. Heraklion: ACM, 2020. 2. [doi: 10.1145/3342195.3392698]

[86] Ustiugov D, Petrov P, Kogias M, Bugnion E, Grot B. Benchmarking, analysis, and optimization of serverless function snapshots. In: Proc. of the 26th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Virtual: ACM, 2021. 559–572. [doi: 10.1145/3445814.3446714]

[87] Google. Overview of memory management | Android developers. https://developer.android.com/topic/performance/memory-overview

[88] Wei XD, Lu FM, Wang TX, Gu JY, Yang YH, Chen R, Chen HB. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In: Proc. of the 17th USENIX Symp. on Operating Systems Design and Implementation. Boston: USENIX Association, 2023. 497–517.

[89] Saxena D, Ji T, Singhvi A, Khalid J, Akella A. Memory deduplication for serverless computing with medes. In: Proc. of the 17th European Conf. on Computer Systems. Rennes: ACM, 2022. 714–729. [doi: 10.1145/3492321.3524272]

[90] Sahraei A, Demetriou S, Sobhgol A, Zhang HR, Nagaraja A, Pathak N, Joshi G, Souza C, Huang B, Cook W, Golovei A, Venkat P, McFague A, Skarlatos D, Patel V, Thind R, Gonzalez E, Jin Y, Tang CQ. XFaaS: Hyperscale and low cost serverless functions at meta. In: Proc. of the 29th Symp. on Operating Systems Principles. Koblenz: ACM, 2023. 231–246. [doi: 10.1145/3600006.3613155]

[91] Mahgoub A, Yi EB, Shankar K, Elnikety S, Chaterji S, Bagchi S. ORION and the three rights: Sizing, bundling, and prewarming for serverless dags. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2022. 303–320.

[92] Stojkovic J, Xu TY, Franke H, Torrellas J. SpecFaaS: Accelerating serverless applications with speculative function execution. In: Proc. of the 2023 IEEE Int'l Symp. on High-performance Computer Architecture (HPCA). Montreal: IEEE, 2023. 814–827. [doi: 10.1109/HPCA56546.2023.10071120]

[93] Bhasi VM, Gunasekaran JR, Thinakaran P, Mishra CS, Kandemir MT, Das C. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In: Proc. of the ACM Symp. on Cloud Computing. Seattle: ACM, 2021. 153–167. [doi: 10.1145/3472883.3486992]

[94] Roy RB, Patel T, Tiwari D. IceBreaker: Warming serverless functions better with heterogeneity. In: Proc. of the 27th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2022. 753–767. [doi: 10.1145/3503222.3507750]

[95] Gunasekaran JR, Thinakaran P, Nachiappan NC, Kandemir MT, Das CR. Fifer: Tackling resource underutilization in the serverless era.

In: Proc. of the 21st Int'l Middleware Conf. Delft: ACM, 2020. 280–295. [doi: 10.1145/3423211.3425683]

[96] Singhvi A, Balasubramanian A, Houck K, Shaikh MD, Venkataraman S, Akella A. Atoll: A scalable low-latency serverless platform. In: Proc. of the 2021 ACM Symp. on Cloud Computing. Seattle: ACM, 2021. 138–152. [doi: 10.1145/3472883.3486981]

[97] Zhou ZZ, Zhang YQ, Delimitrou C. AQUATOPE: QoS-and-uncertainty-aware resource management for multi-stage serverless workflows. In: Proc. of the 28th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, Vol. 1. Vancouver: ACM, 2022. 1–14. [doi: 10.1145/3567955.3567960]

[98] Daw N, Bellur U, Kulkarni P. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In: Proc. of the 21st Int'l Middleware Conf. Delft: ACM, 2020. 356–370. [doi: 10.1145/3423211.3425690]

[99] Li ZJ, Guo LS, Chen Q, Cheng JG, Xu CH, Zeng DZ, Song Z, Ma T, Yang Y, Li C, Guo MY. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In: Proc. of the 2022 USENIX Annual Technical Conf. Carlsbad: USENIX Association, 2022. 69–84.

[100] Shahrad M, Fonseca R, Goiri Í, Chaudhry GI, 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 2020 USENIX Annual Technical Conf. USENIX Association, 2020. 205–218.

[101] Wang A, Chang S, Tian HS, Wang HQ, Yang HR, Li HB, Du R, Cheng Y. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In: Proc. of the 2021 USENIX Annual Technical Conf. USENIX Association, 2021. 443–457.

[102] Klimovic A, Wang YW, Kozyrakis C, Stuedi P, Pfefferle J, Trivedi A. Understanding ephemeral storage for serverless analytics. In: Proc. of the 2018 USENIX Annual Technical Conf. Boston: USENIX Association, 2018. 789–794.

[103] Pu QF, Venkataraman S, Stoica I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In: Proc. of the 16th USENIX Conf. on Networked Systems Design and Implementation. Boston: USENIX Association, 2019. 193–206.

[104] Romero F, Chaudhry GI, Goiri Í, Gopa P, Batum P, Yadwadkar NJ, Fonseca R, Kozyrakis C, Bianchini R. Faa$T: A transparent auto-scaling cache for serverless applications. In: Proc. of the ACM Symp. on Cloud Computing. Seattle: ACM, 2021. 122–137. [doi: 10.1145/3472883.3486974]

[105] The New Stack. What AWS Lambda's performance stats reveal. 2019. https://thenewstack.io/what-aws-lambdas-performance-stats-reveal/

[106] Mvondo D, Bacou M, Nguetchouang K, Ngale L, Pouget S, Kouam J, Lachaize R, Hwang J, Wood T, Hagimont D, De Palma N, Batchakui B, Tchana A. OFC: An opportunistic caching system for FaaS platforms. In: Proc. of the 16th European Conf. on Computer Systems. Online Event: ACM, 2021. 228–244. [doi: 10.1145/3447786.3456239]

[107] Klimovic A, Wang YW, Stuedi P, Trivedi A, Pfefferle J, Kozyrakis C. Pocket: Elastic ephemeral storage for serverless analytics. In: Proc. of the 13th USENIX Conf. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018. 427–444.

[108] Khandelwal A, Tang YP, Agarwal R, Akella A, Stoica I. Jiffy: Elastic far-memory for stateful serverless analytics. In: Proc. of the 17th European Conf. on Computer Systems. Rennes: ACM, 2022. 697–713. [doi: 10.1145/3492321.3527539]

[109] Zhang T, Xie D, Li FF, Stutsman R. Narrowing the gap between serverless and its state with storage functions. In: Proc. of the 2019 ACM Symp. on Cloud Computing. Santa Cruz: ACM, 2019. 1–12. [doi: 10.1145/3357223.3362723]

[110] Copik M, Böhringer R, Calotoiu A, Hoefler T. FMI: Fast and cheap message passing for serverless functions. In: Proc. of the 37th Int'l Conf. on Supercomputing. Orlando: ACM, 2023. 373–385. [doi: 10.1145/3577193.3593718]

[111] Barcelona-Pons D, Sánchez-Artigas M, París G, Sutra P, García-López P. On the FaaS track: Building stateful distributed applications with serverless architectures. In: Proc. of the 20th Int'l Middleware Conf. Davis: ACM, 2019. 41–54. [doi: 10.1145/3361525.3361535]

[112] Barcelona-Pons D, Sutra P, Sánchez-Artigas M, París G, García-López P. Stateful serverless computing with Crucial. ACM Trans. on Software Engineering and Methodology, 2022, 31(3): 39. [doi: 10.1145/3490386]

[113] Jia ZP, Witchel E. Boki: Stateful serverless computing with shared logs. In: Proc. of the 28th ACM SIGOPS Symp. on Operating Systems Principles. ACM, 2021. 691–707. [doi: 10.1145/3477132.3483541]

[114] Qi SX, Monis L, Zeng ZT, Wang IC, Ramakrishnan KK. SPRIGHT: Extracting the server from serverless computing! High-performance eBPF-based event-driven, shared-memory processing. In: Proc. of the 2022 ACM SIGCOMM Conf. Amsterdam: ACM, 2022. 780–794. [doi: 10.1145/3544216.3544259]

[115] Mahgoub A, Shankar K, Mitra S, Klimovic A, Chaterji S, Bagchi S. SONIC: Application-aware data passing for chained serverless applications. In: Proc. of the 2021 USENIX Annual Technical Conf. USENIX Association, 2021. 285–301.

[116] Yu HF, Fontenot C, Wang H, Li J, Yuan X, Park SJ. Libra: Harvesting idle resources safely and timely in serverless clusters. In: Proc. of the 32nd Int'l Symp. on High-performance Parallel and Distributed Computing. Orlando: ACM, 2023. 181–194. [doi: 10.1145/3588195.

3592996]

[117] Mampage A, Karunasekera S, Buyya R. Deep reinforcement learning for application scheduling in resource-constrained, multi-tenant serverless computing environments. Future Generation Computer Systems, 2023, 143: 277–292. [doi: 10.1016/j.future.2023.02.006]

[118] Zafeiropoulos A, Fotopoulou E, Filinis N, Papavassiliou S. Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms. Simulation Modelling Practice and Theory, 2022, 116: 102461. [doi: 10.1016/j.simpat.2021.102461]

[119] Chetabi FA, Ashtiani M, Saeedizade E. A package-aware approach for function scheduling in serverless computing environments. Journal of Grid Computing, 2023, 21(2): 23. [doi: 10.1007/s10723-023-09657-y]

[120] Qiu HR, Mao WC, Patke A, Wang C, Franke H, Kalbarczyk ZT, Başar T, Iyer RK. SIMPPO: A scalable and incremental online learning framework for serverless resource management. In: Proc. of the 13th Symp. on Cloud Computing. San Francisco: ACM, 2022. 306–322. [doi: 10.1145/3542929.3563475]

[121] Kaffes K, Yadwadkar NJ, Kozyrakis C. Hermod: Principled and practical scheduling for serverless functions. In: Proc. of the 13th Symp. on Cloud Computing. San Francisco: ACM, 2022. 289–305. [doi: 10.1145/3542929.3563468]

[122] Tariq A, Pahl A, Nimmagadda S, Rozner E, Lanka S. Sequoia: Enabling quality-of-service in serverless computing. In: Proc. of the 11th ACM Symp. on Cloud Computing. Virtual Event: ACM, 2020. 311–327. [doi: 10.1145/3419111.3421306]

[123] Fu YQ, Liu L, Wang HL, Cheng Y, Chen SQ. SFS: Smart OS scheduling for serverless functions. In: Proc. of the 2022 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Dallas: IEEE, 2022. 1–16. [doi: 10.1109/SC41404.2022.00047]

[124] Bhasi VM, Gunasekaran JR, Sharma A, Kandemir MT, Das C. Cypress: Input size-sensitive container provisioning and request scheduling for serverless platforms. In: Proc. of the 13th Symp. on Cloud Computing. San Francisco: ACM, 2022. 257–272. [doi: 10.1145/3542929.3563464]

[125] Abdi M, Ginzburg S, Lin XC, Faleiro J, Chaudhry GI, Goiri I, Bianchini R, Berger DS, Fonseca R. Palette load balancing: Locality hints for serverless functions. In: Proc. of the 18th European Conf. on Computer Systems. Rome: ACM, 2023. 365–380. [doi: 10.1145/3552326.3567496]

[126] Fuerst A, Sharma P. Locality-aware load-balancing for serverless clusters. In: Proc. of the 31st Int'l Symp. on High-performance Parallel and Distributed Computing. Minneapolis: ACM, 2022. 227–239. [doi: 10.1145/3502181.3531459]

[127] Li ZJ, Chen Q, Xue S, Ma T, Yang Y, Song Z, Guo MY. Amoeba: QoS-awareness and reduced resource usage of microservices with serverless computing. In: Proc. of the 2020 IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS). New Orleans: IEEE, 2020. 399–408. [doi: 10.1109/IPDPS47924.2020.00049]

[128] Roy RB, Patel T, Gadepally V, Tiwari D. Mashup: Making serverless computing useful for HPC workflows via hybrid execution. In: Proc. of the 27th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Seoul: ACM, 2022. 46–60. [doi: 10.1145/3503221.3508407]

[129] Fingler H, Zhu ZT, Yoon E, Jia ZP, Witchel E, Rossbach CJ. DGSF: Disaggregated GPUs for serverless functions. In: Proc. of the 2022 IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS). Lyon: IEEE, 2022. 739–750. [doi: 10.1109/IPDPS53621.2022.00077]

[130] Gu JF, Zhu YC, Wang PX, Chadha M, Gerndt M. FaST-GShare: Enabling efficient spatio-temporal GPU sharing in serverless computing for deep learning inference. In: Proc. of the 52nd Int'l Conf. on Parallel Processing. Salt Lake City: ACM, 2023. 635–644. [doi: 10.1145/3605573.3605638]

[131] Rausch T, Rashed A, Dustdar S. Optimized container scheduling for data-intensive serverless edge computing. Future Generation Computer Systems, 2021, 114: 259–271. [doi: 10.1016/j.future.2020.07.017]

[132] Ristov S, Gritsch P. FaaSt: Optimize makespan of serverless workflows in federated commercial FaaS. In: Proc. of the 2022 IEEE Int'l Conf. on Cluster Computing (CLUSTER). Heidelberg: IEEE, 2022. 183–194. [doi: 10.1109/CLUSTER51413.2022.00032]

[133] Aske A, Zhao XH. Supporting multi-provider serverless computing on the edge. In: Proc. of the 47th Int'l Conf. on Parallel Processing. Eugene: ACM, 2018. 20. [doi: 10.1145/3229710.3229742]

[134] Yu HF, Wang H, Li J, Yuan X, Park SJ. Accelerating serverless computing by harvesting idle resources. In: Proc. of the 2022 ACM Web Conf. Lyon: ACM, 2022. 1741–1751. [doi: 10.1145/3485447.3511979]

[135] 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.

[136] Jin C, Zhang ZL, Xiang XY, Zou SY, Huang G, Liu XZ, Jin X. Ditto: Efficient serverless analytics with elastic parallelism. In: Proc. of the 2023 ACM SIGCOMM Conf. New York: ACM, 2023. 406–419. [doi: 10.1145/3603269.3604816]

[137] Wen ZJ, Wang YS, Liu FM. StepConf: SLO-aware dynamic resource configuration for serverless function workflows. In: Proc. of the 2022 IEEE Conf. on Computer Communications. London: IEEE, 2022. 1868–1877. [doi: 10.1109/INFOCOM48880.2022.9796962]

[138] 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]

[139] Akhtar N, Raza A, Ishakian V, Matta I. COSE: Configuring serverless functions using statistical learning. In: Proc. of the 2020 IEEE Conf. on Computer Communications. Toronto: IEEE, 2020. 129–138. [doi: 10.1109/INFOCOM41043.2020.9155363]

[140] Raza A, Akhtar N, Isahagian V, Matta I, Huang L. Configuration and placement of serverless applications using statistical learning. IEEE Trans. on Network and Service Management, 2023, 20(2): 1065–1077. [doi: 10.1109/TNSM.2023.3254437]

[141] Eismann S, Bui L, Grohmann J, Abad C, Herbst N, Kounev S. Sizeless: Predicting the optimal size of serverless functions. In: Proc. of the 22nd Int'l Middleware Conf. Québec City: ACM, 2021. 248–259. [doi: 10.1145/3464298.3493398]

[142] Jarachanthan J, Chen L, Xu F, Li B. Astrea: Auto-serverless analytics towards cost-efficiency and QoS-awareness. IEEE Trans. on Parallel and Distributed Systems, 2022, 33(12): 3833–3849. [doi: 10.1109/TPDS.2022.3172069]

[143] Moghimi A, Hattori J, Li A, Ben Chikha M, Shahrad M. Parrotfish: Parametric regression for optimizing serverless functions. In: Proc. of the 2023 ACM Symp. on Cloud Computing. Santa Cruz: ACM, 2023. 177–192. [doi: 10.1145/3620678.3624654]

[144] Roy RB, Patel T, Liew R, Babuji YN, Chard R, Tiwari D. ProPack: Executing concurrent serverless functions faster and cheaper. In: Proc. of the 32nd Int'l Symp. on High-performance Parallel and Distributed Computing. Orlando: ACM, 2023. 211–224. [doi: 10.1145/3588195.3592988]

[145] Spillner J, Mateos C, Monge DA. FaaSter, better, cheaper: The prospect of serverless scientific computing and HPC. In: Mocskos E, Nesmachnow S, eds. High Performance Computing. Cham: Springer, 2018. 154–168. [doi: 10.1007/978-3-319-73353-1_11]

附中文参考文献:

[5] 阿里云. 函数计算 FC. https://www.aliyun.com/product/fc

[6] 百度智能云. 函数计算 CFC. https://cloud.baidu.com/product/cfc.html

杨光(1984－), 男, 博士生, 主要研究领域为服务器无感知计算, 大数据, 机器学习.

王帅(1982－), 男, 博士, 副研究员, 主要研究领域为大数据系统, 分布式系统, 软件工程.

刘杰(1982－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为分布式系统, 软件工程, 大数据.

叶丹(1971－), 女, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为网络分布式系统, 软件工程.

曲慕子(1993－), 男, 博士生, CCF 学生会员, 主要研究领域为分布式系统, 机器学习, 大语言模型.

钟华(1971－), 男, 博士, 研究员, CCF 高级会员, 主要研究领域为网络分布式系统, 软件工程.