软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(2):488−510 [doi: 10.13328/j.cnki.jos.007146] [CSTR: 32375.14.jos.007146]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

FineFlow: FaaS 工作流部署优化与执行系统*

刘 璐[1,2], 高浩城[1,2], 陈 伟[1,2,3,4], 吴国全[1,2,3,4], 魏 峻[1,2,3,4]

[1](中国科学院 软件研究所, 北京 100190)

[2](中国科学院大学, 北京 100049)

[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)

[4](中国科学院大学南京学院, 江苏 南京 211135)

通信作者: 陈伟, E-mail: wchen@otcaix.iscas.ac.cn

摘 要: FaaS (function-as-a-service, 函数即服务) 工作流由多个函数服务编排而成, 通过对多个函数的协调控制来实现复杂的业务应用. 当前 FaaS 工作流系统主要基于集中式的数据存储实现函数间的数据传递, 导致 FaaS 函数间的数据传输开销大, 显著影响应用性能. 在高并发情况下, 频繁的数据传输还会产生严重的网络带宽资源争用, 导致应用性能下降. 针对上述问题, 基于函数服务间的细粒度数据依赖分析, 提出一种基于关键路径的函数部署优化方法, 设计了依赖敏感的数据存取与管理机制, 有效减少函数间数据传输, 从而降低 FaaS 工作流应用执行的数据传输时延和端到端时延. 设计实现了 FaaS 工作流系统 FineFlow, 并基于 5 个真实 FaaS 工作流应用开展实验评估. 实验结果表明, 相比于基于集中式数据存储函数交互机制的 FaaS 工作流平台, FineFlow 能够有效降低 FaaS 工作流应用的数据传输时延: 最高降低 74.6%, 平均降低 63.8%; 平均降低应用端到端执行时延 19.6%. 特别地, 对于具有明显细粒度数据依赖的 FaaS 工作流应用, 相比于现有的基于数据本地性的优化方法, FineFlow 能够使数据传输时延和端到端时延进一步分别降低 28.4% 和 13.8%. 此外, FineFlow 通过减少跨节点的数据传输, 能够有效缓解网络带宽波动对 FaaS 工作流执行性能的影响, 提升应用性能受网络带宽影响的鲁棒性.

关键词: FaaS 工作流; 函数即服务; 服务器无感知计算; 数据本地性; 有向无环图; 关键路径; 部署优化

中图法分类号: TP311

中文引用格式: 刘璐, 高浩城, 陈伟, 吴国全, 魏峻. FineFlow: FaaS工作流部署优化与执行系统. 软件学报, 2025, 36(2): 488–510. http://www.jos.org.cn/1000-9825/7146.htm

英文引用格式: Liu L, Gao HC, Chen W, Wu GQ, Wei J. FineFlow: FaaS Workflow Deployment Optimization and Execution System. Ruan Jian Xue Bao/Journal of Software, 2025, 36(2): 488–510 (in Chinese). http://www.jos.org.cn/1000-9825/7146.htm

FineFlow: FaaS Workflow Deployment Optimization and Execution System

LIU Lu[1,2], GAO Hao-Cheng[1,2], CHEN Wei[1,2,3,4], WU Guo-Quan[1,2,3,4], WEI Jun[1,2,3,4]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

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

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

[4](University of Chinese Academy of Sciences, Nanjing, Nanjing 211135, China)

Abstract: A function-as-a-service (FaaS) workflow, composed of multiple function services, can realize a complex business application by orchestrating and controlling the function services. The current FaaS workflow execution systems achieve data transfer among function services mainly based on centralized data storages, resulting in heavy data transmission overhead and affecting application performance significantly. In the cases of high concurrency, frequent data transmission will also cause serious contention for network bandwidth

* 基金项目: 国家重点研发计划 (2021YFB2600301); 中国科学院软件研究所重大项目 (ISCAS-ZD-202302)
收稿时间: 2023-05-22; 修改时间: 2023-09-23; 采用时间: 2024-01-04; jos 在线出版时间: 2024-04-12
CNKI 网络首发时间: 2024-04-17

© 中国科学院软件研究所 http://www.jos.org.cn

resources, resulting in application performance degradation. To address the above problems, this study analyzes the fine-grained data dependency between function services and proposes a critical path-based FaaS workflow deployment optimization method. In addition, the study designs a dependency-sensitive data access and management mechanism to effectively reduce the data transmission between function services, thereby reducing the data transmission latency and end-to-end execution latency of FaaS workflow applications. The study implements a FaaS workflow system, FineFlow, and conducts experiments based on five real-world FaaS workflow applications. The experimental results show that FineFlow can effectively reduce the data transmission latency (the highest reduction and the average reduction are 74.6% and 63.8%, respectively) compared with the FaaS workflow platform with the centralized data storing-based function interaction mechanism. On average, FineFlow reduces the latency of the end-to-end FaaS workflow executions by 19.6%. In particular, for the FaaS workflow application with fine-grained data dependencies, FineFlow can further reduce its data transmission latency and the end-to-end execution latency by 28.4% and 13.8% respectively compared with the state-of-the-art work. In addition, FineFlow can effectively alleviate the impact of network bandwidth fluctuations on application performance by reducing cross-node data transmission, improving the robustness of application performance influenced by the network bandwidth changes.

Key words: FaaS workflow; function-as-a-service (FaaS); serverless computing; data locality; directed acyclic graph (DAG); critical path; deployment optimization

交通基础设施数字化和智慧城市等众多领域发展催生了大量数据密集型应用需求, 如传感器数据处理、图像与视频分析识别等. 同时, 服务器无感知 (serverless) 计算作为一种新兴的云计算服务模式在众多领域中逐步得到应用. 服务器无感知计算主要由函数即服务 (function-as-a-service, FaaS) 和后端即服务 (backend-as-a-service, BaaS) 组成. 其中, FaaS 在 serverless 架构中扮演着最为核心的角色, 故 FaaS 又称为服务器无感知函数, 现有研究工作常用 FaaS 来指代 serverless[1−4]. 服务器无感知计算使开发者能够以函数即服务 (FaaS) 的方式编写和运行应用程序, 开发者只需要编写代码实现函数功能, 无须关心基础架构资源运维和部署运行环境. 但是, 由于单个函数服务难以满足实际场景中的复杂业务需求, 应用系统的开发实现需要使用基于多个函数服务编排而成的 FaaS 工作流[5,6], 通过对多个函数服务的协调控制来实现复杂的业务功能. 业界和学术界一致认为, 包括图像处理、科学计算在内的几类复杂应用程序最终将在 FaaS 平台上广泛应用[7].

FaaS 工作流是一系列有序的、相互协调的工作任务集合, 通常每个任务对应一个函数服务. FaaS 工作流可以描述业务的执行过程, 通常通过有向无环图 (directed acyclic graph, DAG) 进行流程定义, 由执行引擎实现任务的按序自动执行. FaaS 工作流平台借助 FaaS 的特性和系统提供的编排能力, 通过函数服务的复用和演化帮助应用开发团队提高效率, 降低运维成本. 目前各大云服务提供商均提供了 FaaS 工作流平台, 如 AWS Step Functions[8]、Microsoft Durable Functions[9]、Google Workflows[10]、阿里巴巴 serverless 工作流[11]等.

然而, 已有研究发现 FaaS 工作流平台在应用的调度管理方面存在较大时间开销[12]. 如图 1 所示, 在当前主流的 FaaS 工作流平台 AWS Step Functions[8]和 IBM Cloud Functions[13]上, 工作流应用的总体运行时间远远超过函数执行实际时间. 这部分时延严重影响了 FaaS 工作流应用的执行效率和服务质量. FaaS 工作流执行过程中的时延可以进一步细分为数据传输时延、调度时延、冷启动时延和计算时延等. 如图 2 所示, FaaSFlow[2]分析了 6 个工作流应用 (从左到右依次分别是视频处理、文件处理、数据处理和 3 个不同领域的科学工作流应用的简称) 执行过程中存在的调度和数据传输时延. 结果显示, 在 FaaS 工作流执行过程中, 除函数服务初始化过程中引入的冷启动时延外, 数据传输时延相对于函数调度时延占比更高, 平均约占系统总体开销的 95.1%. 尤其对于视频和图像处理这样的数据密集型应用来说, 较大的数据规模导致函数间数据传输开销更加显著.

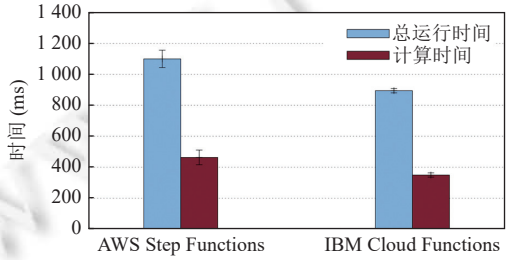


图 1　基于主流商用服务器无感知平台的 FaaS 工作流执行时间

© 中国科学院软件研究所 http://www.jos.org.cn

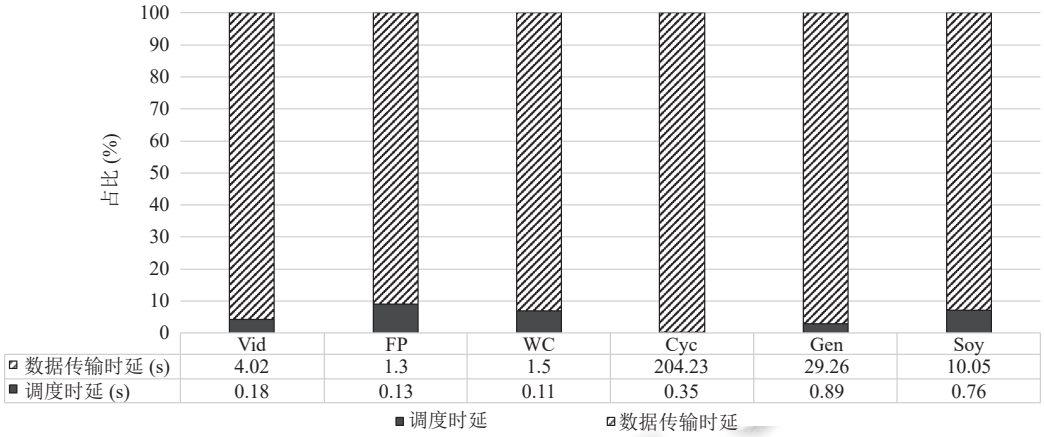


图 2 FaaS 工作流中数据传输时延和调度时延的占比分析

针对数据传输问题, 一方面, 当前主流的 FaaS 工作流平台提供了相应的策略和方式来实现函数之间的数据访问. 例如, 函数可以通过消息代理 (一种基于发布/订阅架构的消息代理系统) 或共享存储 (一种远程网络存储服务器) 进行数据交互[14–16]. 但是, 函数之间的数据访问缺乏统一的编程接口, 开发者必须自己选择并实现有效的数据存储和访问方式[17], 这增加了 FaaS 工作流应用开发和部署的复杂性.

另一方面, 已有相关研究工作利用数据本地性来进行优化[2,12,18–21], 主要采用基于应用粒度和函数粒度的分析方式, 将工作流应用整体放置在同一节点, 或通过分析函数之间的依赖关系, 将存在数据依赖的函数放置在同一节点, 以此消除在工作流执行过程中数据跨节点传输产生的开销. 但是, 已有工作缺乏对应用程序细粒度数据依赖的分析, 会增加数据传输网络带宽资源争用, 带来额外开销.

综上, 当前 FaaS 工作流的执行面临着以下挑战.

(1) 构成 FaaS 工作流的函数服务间数据传输开销大. 在工作流执行过程中, 函数之间的交互存在大量数据传输, 即前驱函数的输出结果将作为后继函数的输入. 当前 FaaS 工作流平台一般采用集中式存储, 数据在远程存储节点和本地函数执行节点之间频繁传输, 导致了网络开销和时延问题.

(2) 缺乏封装良好的数据存取统一编程接口与数据管理机制. 开发人员需要自行为函数服务的输入/输出选择合适的存取方式, 并在函数代码中通过编码实现存取逻辑. 这一方式使得函数功能逻辑与底层数据存取紧密耦合, 增加了 FaaS 函数的开发和维护难度.

针对上述问题, 本文提出了基于细粒度数据依赖分析的 FaaS 工作流部署优化方法, 其核心在于充分利用数据本地性, 有效减少跨节点数据传输带来的网络开销和时延, 从而降低 FaaS 工作流执行的端到端时延.

本文的主要贡献包括以下 4 点.

● 提出了基于细粒度数据依赖的 FaaS 工作流划分与部署优化方法. 方法针对函数服务间数据传输开销大的问题, 分析函数服务间参数粒度的数据依赖关系, 采用基于关键路径的 FaaS 工作流划分策略实现函数部署优化, 通过同一节点内函数间交互数据的本地存储方式减少跨节点的数据传输规模, 优化工作流执行的端到端时延.

● 设计了依赖敏感的数据存取与管理机制. 机制针对当前数据存取与函数功能逻辑紧密耦合的问题, 向函数服务提供统一的数据访问编程接口. 该机制基于函数间细粒度的数据依赖分析进行数据存取位置的自动决策, 并对本地存储数据进行生命周期管理, 实现内存资源的有效利用. 因此, 函数开发人员无须考虑具体的数据存取逻辑和方式, 实现函数功能与数据访问的有效分离.

● 设计实现了 FaaS 工作流系统 FineFlow, 包括划分调度器、执行引擎和存储管理机制, 可支持具有并行 (parallel)、选择 (choice)、并行循环 (foreach) 等复杂逻辑结构的工作流应用的优化部署和调度执行.

● 基于 5 个真实 FaaS 工作流应用负载进行了系统测试. 从系统对应用执行时延的优化、网络带宽利用率的提高等方面验证了方法和策略的有效性, 并评估了方法策略对系统吞吐量与响应时间的影响以及组件带来的额外

© 中国科学院软件研究所 http://www.jos.org.cn

开销.

本文第 1 节介绍 FaaS 工作流相关的基本概念. 第 2 节介绍相关研究工作. 第 3 节介绍本文设计实现的 FaaS 工作流系统. 第 4 节基于 5 个真实工作流应用对本文提出的方法进行实验评估, 以验证本文方法的有效性. 第 5 节对本文进行总结.

1 基本概念

1.1 FaaS 及 FaaS 工作流

FaaS 是一种云计算服务模型, 开发者以函数的方式来编写应用程序, 开发者只需要编写代码实现函数功能, 无须考虑如何管理服务器资源或运行环境, 从而实现了更为高效和灵活的开发方式. 函数服务通常基于容器或虚拟化技术进行部署和运行, 并通过运行实例的自动伸缩来应对负载的变化. 目前, 主流 FaaS 平台可支持多种编程语言, 例如 Java、Python、Node.js 等. 但是, 单一的函数往往难以满足实际场景中的复杂业务需求, 由此引入了 FaaS 工作流的概念. FaaS 工作流可以表示为有向无环图 (directed acyclic graph, DAG), 是一系列有序的、相互协作的函数服务集合组成的应用, 它的流程定义可以描述业务应用的执行过程以及函数间的相互关联, 使执行引擎能够基于工作流定义实现任务的自动化执行. FaaS 工作流平台是一种基于函数服务来编排、执行和管理应用的系统和环境.

以交通基础设施数字化领域为例, 其业务场景需要视频处理、图像识别和传感器数据分析处理等应用, 这些应用往往是由多个任务构成的. 例如, 在交通道路上部署的卡口摄像头, 需要对采集到的来往车辆的图像进行识别. 如图 3 所示, 构成图像识别[22]工作流应用的各个任务之间存在执行依赖关系, 每个任务对应一个函数服务, 可以通过 DAG 表示应用结构.

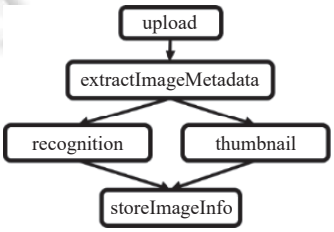
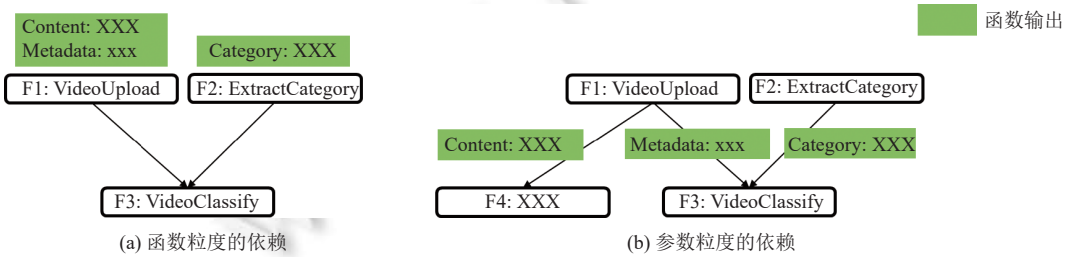


图 3 FaaS 工作流应用示例——图像识别

1.2 函数间细粒度数据依赖

众所周知, 构成 FaaS 工作流的函数间存在数据依赖. 如图 4 所示, 图 4(a) 的 F1、F3 两个函数中, F1 的所有输出会作为 F3 的输入, 则称 F1、F3 之间具有函数粒度的数据依赖关系. 另一方面, 如图 4(b) 所示, F3 只依赖于 F1 的部分输出数据 Metadata, 则称 F1、F3 之间具有参数粒度的数据依赖. 因此, 考虑到 FaaS 工作流的应用多样, 导致函数间的依赖程度不尽相同, 需要采用细粒度依赖, 即函数间参数粒度的数据依赖关系来更加准确地刻画函数服务间的依赖信息.



(a) 函数粒度的依赖 (b) 参数粒度的依赖

图 4 函数粒度与参数粒度的依赖对比

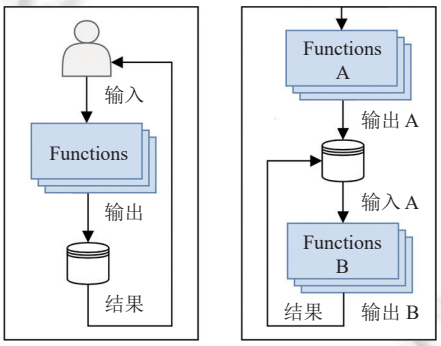
© 中国科学院软件研究所 http://www.jos.org.cn

2 相关工作

2.1 FaaS 工作流调度优化

FaaS 工作流应用执行时的系统开销主要分为两个部分, 一是 FaaS 函数本身的冷启动开销, 二是工作流执行过程中的调度开销和数据传输开销. 已有相关研究工作分别针对冷启动[3,12,23−27]、调度开销[2,28]、数据传输开销[4,17]以及其他可能存在的系统开销[29]进行了优化技术研究. 与本文工作最为相关的是 FaaS 工作流数据传输开销优化的相关工作.

如图 5 所示, 函数服务的调用主要有两种方式. 函数服务通过外部请求触发执行, 执行结果通过后端数据库保存, 然后再返回给用户, 这种方式称为外部调用, 如图 5(a) 所示. 相比之下, 函数之间的调用则被称为内部调用, 而 FaaS 工作流的实现主要依赖于内部调用. 但是, 当前 FaaS 工作流平台的设计原则是避免函数之间进行直接的、大规模的数据传输和交互, 表 1 列举了主流平台对于函数请求的数据规模限额. 因此, 函数服务间的内部调用主要通过后端数据库实现, 即前一个函数的输出结果将保存在后端数据库中, 并由下一个函数获取和使用, 如图 5(b) 所示. 然而, 基于后端数据库集中存储的函数内部调用方式存在一个不足之处, 即, 可能引入额外的数据传输开销. 由于函数执行节点和数据存储节点之间频繁交互, 工作流执行的端到端延迟可能会显著增加. 因此, 相较于传统工作流调度的多目标优化研究[30,31], 现有的 FaaS 工作流更加注重对数据的管理, 基于数据本地性进行应用中函数服务的部署优化和调度, 降低数据传输时延. 资源使用的动态波动是 FaaS 系统中的另一个特点. 在 FaaS 环境下, 函数实例副本能够自动缩放, 并且可以重用容器, 使得每个函数在动态运行时可能会存在多个实例, 这种波动性也使得 FaaS 工作流应用的资源开销难以预测. 因此, 面向 FaaS 工作流应用的调度优化更注重在一个较短的决策时间内找到一个合理的工作流调度策略, 在满足 FaaS 工作流中函数的资源需求前提下尽可能地降低工作流执行的端到端时延.



(a) 外部调用 (b) 内部调用

图 5 函数服务调用模式

表 1 主流服务器无感知平台对于函数请求大小 (包括所有请求数据) 的限额

服务器无感知平台	限额
AWS Lambda	6 MB (synchronous), 256 KB (asynchronous)
Google Cloud Functions	10 MB for data sending to functions
Microsoft Azure Functions	1 MB with single stream
Alibaba Function Compute	6 MB (synchronous), 128 KB (asynchronous)
Apache OpenWhisk	1 MB for each entity

SAND[12]将工作流整体封装到一个容器中, 以此消除因跨节点数据传输而在工作流中产生的网络开销和延迟, 但是忽略了资源争用的干扰和对函数服务部署带来的约束. Faastlane[18]和 SAND 类似, 将整个 FaaS 工作流应用部署在单个虚拟机 (VM) 中执行. 不同的是, Faastlane 考虑了资源约束, 当单个 VM 的资源无法满足整个工作流应用

© 中国科学院软件研究所 http://www.jos.org.cn

的资源需求时, 会进行工作流的拆分并使用多个 VM 来部署应用. 虽然这些方法试图减少函数服务之间的通信延迟, 但采取极端的方法强制将工作流中的所有函数部署在同一容器中执行, 会在一定程度上牺牲调度的灵活性, 并出现资源竞争导致的服务性能下降甚至不可用, 为资源分配带来了挑战. NightCore[19]假设构成应用的所有服务都可以在同一服务器上部署运行, 并通过减少同一节点上服务间的 RPC 开销来进行优化. Viil 等人[20]使用多级 k-way 图分区自动为多云环境调配科学工作流. 然而, 他们的分区算法并不适用于服务器无感知平台环境. 相比之下, FaaSFlow[2]选择通过 Worker 节点的共享内存进行同一节点上函数之间的数据传输. GlobalFlow[21]考虑了集群资源的地理分布, 将位于同一物理位置的函数分组到同一子图中, 并用轻量级函数将它们连接起来, 以增强数据本地性并减少传输延迟.

以上方法都使用了数据本地性来进行数据传输时延的优化, 但均是基于应用粒度或函数粒度的 FaaS 工作流应用划分和部署优化. 但是, 考虑到应用中函数服务间存在参数粒度的数据依赖, 因此仍然存在进一步优化的空间. 如图 6 所示, 由于参数粒度的数据依赖分析对每个参数进行单独存储, 因此, 相比于函数粒度分析直接将函数输出参数全部统一存储于本地或远端, 参数粒度的数据依赖分析做了更细致的划分和考虑, 仅将本节点内后续函数依赖的数据存储在本地, 而跨节点的数据仍存储在远端. 由于计算节点的内存资源有限, 在使用本地存储时, 若对函数进行细粒度的数据依赖分析, 仅在本地存储当前节点后续函数所需的数据信息, 则可以进一步提高内存的使用效率, 并在一定程度上进一步实现 FaaS 工作流应用资源消耗和执行时间的优化.

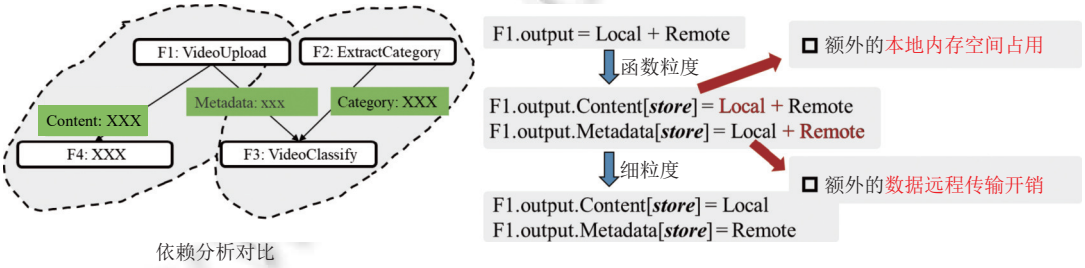


图 6 函数粒度和参数粒度的数据依赖分析对比

综上, 细粒度的函数依赖分析能够影响和优化函数调度策略与数据存取的规模. 在划分调度时, 如果采用细粒度的函数依赖分析, 尽可能地将存在较大规模数据依赖的函数部署到同一节点, 可以利用数据本地性来优化 FaaS 工作流端到端执行时延, 同时避免不必要的本地数据存储带来的额外资源开销.

2.2 FaaS 函数间的数据访问机制

在数据密集型 FaaS 工作流应用中, 函数间的数据访问开销成为工作流平台面临的一个重要挑战[7,32,33]. 当前, FaaS 函数之间的数据通常通过远程存储 (例如 S3[15]) 的方式进行存取访问, 开发人员需要在函数中编码实现远程数据的存取操作. 虽然通过远程存储可以清晰地将计算和存储资源分离, 但对于数据密集型应用而言, 这将会显著增加性能开销[33]. 例如, Pu 等人[34]使用 S3 远程存储在 AWS Lambda 上运行了具有 100 TB 数据规模的 CloudSort 应用基准测试, 发现相比于在 VM 集群上进行本地数据存储和访问, 其开销增加了多达 500 倍. SONIC[17]运行了一个具有简单顺序结构的机器学习工作流应用, 结果显示, 远程存储的使用导致数据传输开销占了端到端时延的 75% 以上.

为了减少数据远程传输的开销, 现有工作提出了各种策略, 例如实现数据交换 Operator 以此优化对象存储[33,34]、使用基于内存的存储 (例如 ElastiCache Redis) 替换基于磁盘的对象存储 (例如 S3) 或组合不同的存储介质 (例如 DRAM、SSD、NVMe) 以匹配应用需求[14,35]. 然而, 这些方法仍需要通过多次网络通信来传递数据.

SONIC[17]依据 AWS Lambda 平台的特性, 将函数间的数据访问机制分为 Direct-Passing、VM-Storage、Remote storage 这 3 种, 每种数据访问机制在延迟、成本和可扩展性上均有差异, 且没有一种单一的机制可以适用于所有 FaaS 工作流应用. 基于此, 该工作设计了相关策略来帮助工作流函数自动选择最优的数据访问方式. 但其仅基于 Lambda 商用云环境进行实现, 且并没有考虑函数间可能存在的细粒度数据依赖.

© 中国科学院软件研究所 http://www.jos.org.cn

将函数串联起来的工作流应用会逐步累积函数服务间的交互延迟. 为此, Pheromone[4]重点关注 FaaS 函数之间存在的细粒度数据依赖, 提出一种数据驱动的新执行范式. 该方法将子函数的触发与数据相关联, 而不是函数执行结束的状态, 从而使工作流的执行更加高效. 然而, 这种方法会大大改变 FaaS 函数的原始执行机制, 并不符合当前函数服务的编程模型, 会产生新的学习门槛.

Lambdata[36]为了更加具体地定位函数间的执行依赖和数据依赖关系, 设计了新的工作流定义方式. 开发人员可在工作流定义时明确声明每个函数的输入和输出数据的目的, 以帮助系统执行数据优化策略. 然而, 这种方法并没有提供一个统一的编程接口来表达和简化函数间的数据访问方式, 而且其性能与 Apache OpenWhisk[37] 系统本身高度相关.

综上, 当前 FaaS 函数间的数据访问机制仍然存在两个方面的问题: 首先, 大多数系统采用远程集中式的数据存储方式, 且与具体的系统和实现技术紧密相关; 其次, 即使有相关工作提出了基于本地的数据存储策略, 也仍然需要开发人员根据具体的数据存储位置在函数服务中进行相应的存储和访问功能逻辑的编码实现, 与函数服务的功能逻辑紧密耦合, 增加了函数的实现代价. 因此, 需要提供本地和远端存储相结合的数据存储与管理机制, 且通过统一的编程接口实现函数逻辑与数据存取的分离, 使开发人员在函数实现过程中无须考虑具体的数据存取位置和方式.

3 FineFlow 函数服务工作流系统

本文设计并实现了 FineFlow, 其总体架构如图 7 所示. FineFlow 主要包括 3 个组件. (1) 划分调度器. 其输入为工作流与计算节点信息, 输出为划分调度结果, 并通过和计算节点交互把工作流函数部署在计算节点上. 其实现了基于细粒度数据依赖分析的 FaaS 工作流划分调度方法. 方法以工作流函数间数据传输时延为优化目标, 根据每个计算节点上的可用资源约束和每个函数对之间传输的数据传输时延为依据, 进行基于关键路径的 FaaS 工作流应用子图划分, 并将同一子图内的函数服务部署在相同节点. (2) 工作流执行引擎. 划分调度后, 其负责接受工作流执行请求, 并通过和计算节点交互触发节点上部署的函数执行, 内部实现了 FaaS 工作流执行的状态管理机制. 它负责跟踪工作流目前所处的执行步骤, 以及识别步骤的输入/输出数据, 根据执行约束状态, 触发调用相应函数服务. (3) 自适应数据存储服务. 工作流执行时, 其为函数执行的输入提供参数, 并存储函数输出的参数结果. 其内部实现了数据依赖敏感的数据存取与管理机制. 它为函数服务提供统一的编程模型, 根据函数间的数据依赖关系进行数据存取位置的自适应决策, 选用适当的数据存储方式 (本地内存, 或远程存储) 来支持函数间通信, 并对本地存储数据进行生命周期管理, 实现函数功能逻辑与数据管理的解耦. 除上述组件外, FineFlow 采用开源函数服务平台 OpenFaaS 作为函数服务的基本运行支撑环境. FineFlow 与 FaaS 运行环境分离, 通过服务接口实现对函数服务的调用, 因此也可以采用其他开源或商业平台作为函数服务的运行环境.

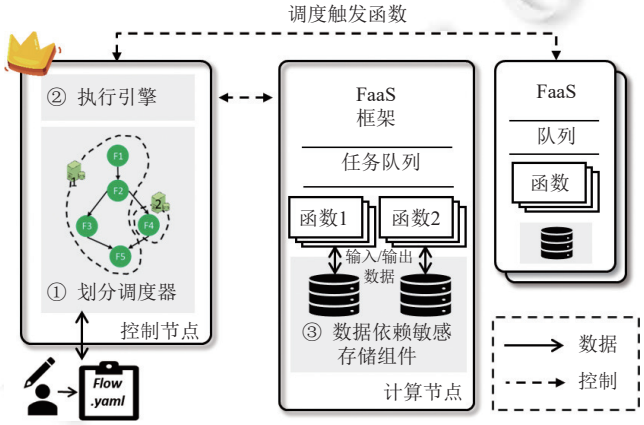


图 7 FineFlow 系统架构

© 中国科学院软件研究所 http://www.jos.org.cn

3.1 划分调度器

如图 8 所示, 划分调度器通过解析用户工作流定义文件得到含有细粒度数据依赖信息的工作流模型, 并将其与计算节点的资源模型作为输入, 采用基于关键路径的子图划分调度算法, 实现函数到节点的映射, 从而实现对 FaaS 工作流应用的划分调度优化.

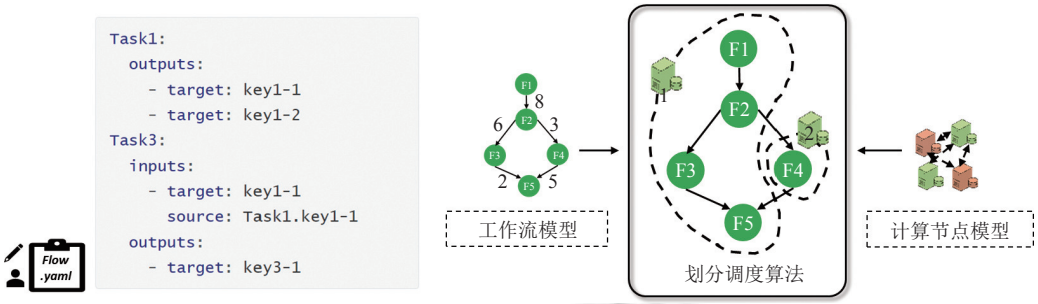


图 8 FineFlow 工作流划分调度器

3.1.1 FaaS 工作流及资源建模

FaaS 工作流划分调度需要考虑计算节点的资源供给、函数任务的资源需求, 并重点对函数任务之间的细粒度数据依赖信息进行刻画.

(1) FaaS 工作流建模

以图 9 所示的图像识别工作流[22]为例, 本文将对工作流中每个函数的执行时间、函数任务之间细粒度的数据依赖关系、函数任务之间的数据传输大小以及任务的资源需求进行建模. 实际运行时工作流中的每个函数服务可以拥有多个实例, 导致工作流实例的实际运行时状态可能与其静态定义不一致, 因此需要对 FaaS 工作流应用的静态和动态信息进行全面刻画.

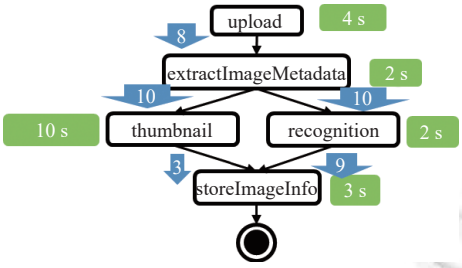


图 9 FaaS 工作流模型

假定 FaaS 工作流应用 $Workflow$ 由 n 个函数构成, 则可表示为 $Workflow=\{func_1,func_2,\ldots,func_n\}$. 对于其中任意函数 $func_j$ $(1\leqslant j\leqslant n)$, 若其输出参数有 O_j 个, 则可表示为 $func_j^{\text{output}}=\{output_1,\ldots,output_{O_j}\}$; 若其输入参数有 I_j 个, 则可表示为 $func_j^{\text{input}}=\{input_1,\ldots,input_{I_j}\}$.

针对函数 $func_j$ 的刻画主要包括 3 方面信息: a) 资源需求; b) 依赖信息; c) 运行时信息. 具体建模信息如下.

a) 资源需求

函数 $func_j$ 的资源需求描述的是在不考虑运行时多实例副本的情况下, 单个函数实例对于 CPU 和内存资源的需求情况, 其中,

CPU: $func_j^C$ 表示函数需要的 CPU 核心数, 单位 vCPU, $func_j^C>0$.

内存: $func_j^M$ 表示函数需要的内存容量, 单位 GB, $func_j^M>0$.

当前对于函数服务资源配置的主流方式是以内存配额为基准, 其他资源, 如 CPU 和网络带宽等, 采用相应的

© 中国科学院软件研究所 http://www.jos.org.cn

比例进行设置, 例如, 将函数服务的 CPU 和内存资源按照 1:2 的比例进行配置. 需要说明的是, 函数运行时的资源分配还与其实际运行的实例数相关, 即运行时函数 $func_j$ 各个维度的资源配置为资源需求与实例数的乘积.

b) 依赖信息

函数服务主要存在函数依赖于参数依赖两个层次的依赖信息.

函数依赖: 表示函数之间的执行顺序, 其中 $Func_j^{\text{pre}}$ 和 $Func_j^{\text{next}}$ 分别表示当前函数服务 $func_j$ 的前驱函数集合与后继函数集合. 当 $Func_j^{\text{pre}}$ 中的所有函数执行完成时, $func_j$ 开始执行; 如果 $Func_j^{\text{next}}$ 中的所有函数仅依赖于 $func_j$, 则其中的函数在 $func_j$ 执行完毕后开始执行.

$$Func_j^{\text{pre}} = \left\{func_{j_{\text{pre}}}, \ldots\right\},\ 1 \leqslant j_{\text{pre}} \leqslant n,$$

$$Func_j^{\text{next}} = \left\{func_{j_{\text{next}}}, \ldots\right\},\ 1 \leqslant j_{\text{next}} \leqslant n.$$

参数依赖: 表示函数之间的细粒度数据依赖. 下面的表达式表示 $func_j$ 的输入 $input_\alpha$ 来自 $func_k$ 的输出 $output_\beta$.

$$func_j^{input_\alpha} = func_k^{output_\beta},\ k \geqslant 1, j \leqslant n, j \neq k, 1 \leqslant \alpha \leqslant I_j, 1 \leqslant \beta \leqslant O_k.$$

c) 运行时信息

考虑到函数服务运行时的动态变化, 需要对其运行时状态信息进行建模, 以更加准确地刻画函数运行时的资源使用情况和运行状态, 主要包括:

- $func_j^{\text{time}} = t_j\ (t_j > 0)$, 表示函数服务的执行时间;
- $func_j^{\text{scale}} = s_j\ (s_j > 0)$, 表示函数服务运行时实例副本数量;
- $func_j^{\text{map}} = p_j\ (p_j > 0)$, 特别是, FaaS 工作流中的 foreach 循环结构每次执行时可以包含同一函数的多个任务节点, 因此需要对其中的任务节点数量进行建模.

当前, 本文工作主要采用周期性的度量和取平均值的方式度量和计算上述运行时信息. 具体地, 方法将 FaaS 工作流应用前次和本次部署之间的时间间隔称为一个周期, 周期的设置方式可以是固定时间间隔, 也可以通过监测特定事件 (如响应时延超出预设阈值、请求失败数量或比例超出阈值等) 的发生来触发再次部署. 任意函数 $func_j$ 的执行时间 t_j 来源于周期内函数的平均执行时间, 函数服务运行时实例副本数量 s_j 通过计算周期内实例缩放的平均数量获得, foreach 步骤包含的任务节点数 p_j 同样表示为周期内 foreach 函数节点的平均数量.

(2) 节点资源建模

节点集群表示为 $N = \{N_1, N_2, \ldots, N_m\}$, 其中节点 i 表示为 $N_i = \{c_i, rc_i, m_i, rm_i, p_i, rp_i\}, 1 \leqslant i \leqslant m$. c_i 表示节点 CPU 总核心数, rc_i 表示节点 CPU 已被分配占用的核心数, 单位均为 vCPU. m_i 表示节点的内存总容量, rm_i 表示节点已被分配占用的内存容量, 单位为 GB. p_i 表示节点可部署的最大容器数量, rp_i 表示节点已部署的容器数量. 从上述指标可以计算得到节点当前可用于分配的内存和 CPU 容量以及可部署容器数量. w 表示计算节点到存储节点的网络带宽, 代表了数据在网络中传输的速度, 单位为 Mb/s.

3.1.2 基于细粒度数据依赖分析的工作流划分调度方法

本文提出基于关键路径的 FaaS 工作流划分调度算法. 算法的目标是在满足资源约束的条件下, 优化工作流应用执行的端到端时延, 并平衡节点的负载. 关键路径是指整个工作流从开始到结束执行过程中时延最长的路径, 由于工作流执行的端到端时延是由关键路径决定的, 算法选择了基于工作流执行关键路径的贪心划分调度策略. 该策略在满足节点资源约束的前提下, 只将对端到端时延影响最大的关键路径上的函数尽可能地划分到相同节点. 划分调度算法流程如图 10 所示, 主要包括初始放置、关键路径识别及排序、函数分组合并几个步骤, 并通过步骤 2 和 3 的不断迭代达到优化目标. 以上划分调度算法实现在划分调度器中.

(1) 初始放置

初始条件下, FaaS 工作流中的每个函数均被划分为单独的一个分组, 并根据当前节点的资源使用情况, 在满足资源的条件下对分组进行节点的随机映射.

© 中国科学院软件研究所 http://www.jos.org.cn

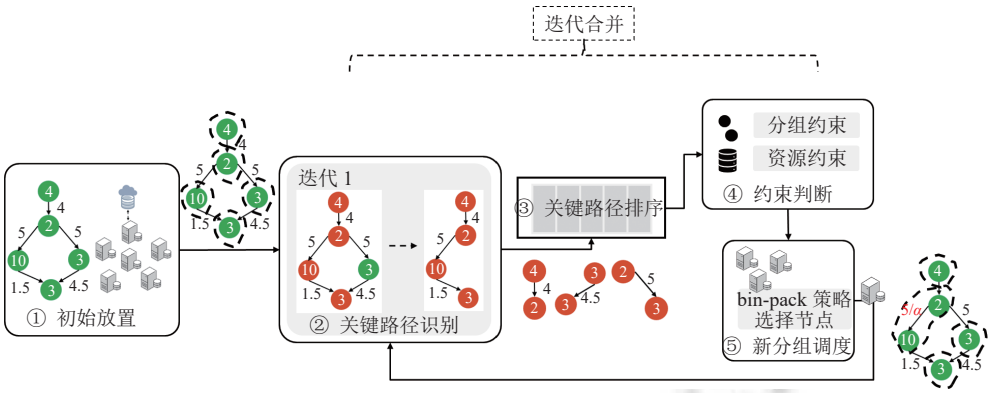


图 10　工作流划分调度流程

(2) 关键路径识别及排序

根据当前的工作流划分情况, 通过数据传输时间和函数执行时间计算得到 DAG 中每条路径的执行时间, 其中从起始节点到结束节点时间最长的路径称为关键路径. 接着对关键路径上所有的边按照数据传输时间长短倒序排列, 数据传输时间越长的边排序越靠前. 排序后得到关键路径上所有边的排序列表, 越靠前的关键边对工作流执行端到端时延的影响越大.

工作流函数 j、k 之间的数据传输时间 TT_{jk} 等于函数 j、k 之间所存在的参数依赖数据大小总和除以当前云边网络带宽, 计算如公式 (1) 所示. 其中对于函数 j 中所有依赖函数 k 的参数 α 构成 j 对 k 的参数依赖集合 A_{jk}.

$$TT_{jk}=\frac{\sum_{\alpha\in A_{jk}} func_j^{input_\alpha}}{W} \tag{1}$$

FaaS 工作流的执行路径 $path$ (路径) 由首尾相接的边 (e_{jk}, 表示函数 j和函数k 首尾连接的边) 组成, 从起始节点开始, 以结束节点终止, $path_m=\{e_{jk},\dots\}$, $1\leqslant k,j\leqslant n, j\neq k$. 任意路径 m 的执行时间 (execution time, ET) 计算公式如公式 (2) 所示. 路径 m 所包含函数构成集合函数集合 P_m, 其中包含的任一函数 $func_h$ 的执行时间为 $func_h^{\text{time}}$, 从函数的运行时信息获得.

$$ET_m=\sum_{e_{jk}\in path_m} TT_{jk}+\sum_{h\in P_m} func_h^{\text{time}} \tag{2}$$

(3) 函数分组合并

依次遍历关键边排序列表, 对关键边上两个函数所在的分组进行合并. 首先进行资源约束判断, 主要存在两个方面的约束, 一是判断节点资源是否满足合并后的分组内函数的总体资源需求, 二是判断两个函数传输的数据大小是否超出了这两个函数的内存使用限制. 如果上述的约束条件都满足, 则表示两个分组可以合并成一个分组. 为新分组选择合适的部署节点. 若分组无法合并, 则继续遍历下一条关键边. 在新分组部署节点选择阶段分为预选和优选两个阶段. 1) 预选阶段. 这一阶段主要根据节点资源进行过滤, 考虑内存、CPU 和最大可部署容器数量这 3 个资源指标, 若节点的任意一项资源不满足新分组的需要, 则过滤此节点. 2) 优选阶段. 经过预选阶段过滤后的剩余节点资源均能够满足新分组的部署资源需求. 优选采用打分制, 从预选节点里选择出得分最高的节点作为新分组部署的节点. 使用 bin-pack 打分策略 MostRequestedPriority, 即选择目前资源使用率最高的节点. 分数 $score_i$ (满分为 10 分) 计算公式如公式 (3) 所示, 其中 cpu 和 $memory$ 分别表示当前这两种资源的利用率.

$$score_i=\frac{cpu\left(10\times\frac{rc_i}{c_i}\right)+memory\left(10\times\frac{rm_i}{m_i}\right)}{2} \tag{3}$$

合并完成后两个函数的数据传输将采用本地内存, 同时对工作流的中这两个函数之间的数据传输时间, 按照内存存取速度和远程存取速度的倍数 α 进行更新 (对于 α, 本文依据实验中采集到的相同数据存储在远端和本地

© 中国科学院软件研究所 http://www.jos.org.cn

内存所需的时间比值, 设置为 20). 合并后关键边传输时间 *time* 更新为 $time/\alpha$. 接着算法回到步骤 (2) 进行新一轮的关键路径识别. 如果遍历完的关键边排序列表都没有可以成功合并的函数分组, 则划分调度结束. 算法结束后, 会将工作流函数按照划分调度的结果依次部署到相应计算节点上.

3.2 数据依赖敏感的存储组件

若两个函数合并为一组, 则表示它们之间传输的参数可以在本地存储, 因此将该参数的存储位置设为本地存储. 整个划分算法执行结束后, 划分调度器进一步分析参数依赖, 若还存在其他函数依赖该参数, 且被部署到其他节点, 则存储位置还需增加远程存储. 这一分析过程可得到各个函数输出数据的存储方式的相关配置. 执行时存储组件会依据该配置为每个参数数据进行相应的存储. 本文设计了如图 11 所示的数据依赖敏感的存储组件, 其主要职责包括:

(1) 为开发人员提供统一的编程接口, 实现机制对开发人员透明, 无须考虑实际的数据存储位置.

(2) 实现依赖敏感的存储策略, 根据函数间的细粒度数据依赖关系自动决策将数据存储在本地或远端.

(3) 管理数据生命周期, 及时回收中间无效数据占用的内存资源.

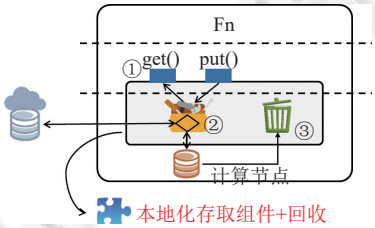


图 11 数据依赖敏感的存储组件

3.2.1 细粒度参数存取组件

如图 12 所示, 组件提供 put() 和 get() 两个数据存取统一编程接口. get 的原理是根据参数的 key 值在本地内存进行数据查找, 如果没有找到, 则从远程存储获取对应的数据, 并把 value 返回给函数. put 的原理是根据函数相关配置信息, 自动判断参数需要存在本地还是远端.

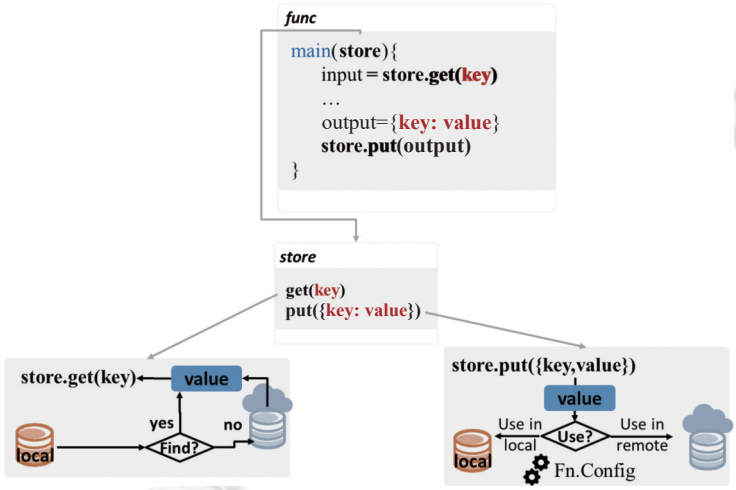


图 12 参数存取组件编程模型

综上所述, 存取组件实现了以下功能.

(1) 参数粒度依赖驱动的数据混合存储. 根据函数的划分部署结果和细粒度依赖关系, 为函数输出的每个参数选择远程或本地的适当存储方式.

© 中国科学院软件研究所 http://www.jos.org.cn

(2) 易于使用的存储访问 API. 基于统一的编程接口, 开发人员无须考虑数据存取的具体方式和代码功能逻辑的具体实现, 实现函数核心功能逻辑与数据存取访问的分离.

3.2.2 内存统一回收机制

内存统一回收机制的主要目的是实现数据生命周期管理, 回收无用内存数据, 从而减小数据本地存储给节点内存资源使用带来的额外压力. 该机制主要分析解决两个问题: a) 内存回收的时机. 本工作将在确保整个工作流实例执行完成后, 函数的输出数据不会再被使用时, 进行统一回收; b) 不同请求实例之间的数据隔离. 如图 13 所示, 为保证各请求实例的数据在回收时互不影响, 该机制通过唯一 ID 来标识不同工作流请求实例, 并对执行过程中产生的所有参数都进行 ID 标识, 实现实例级的数据相互隔离.

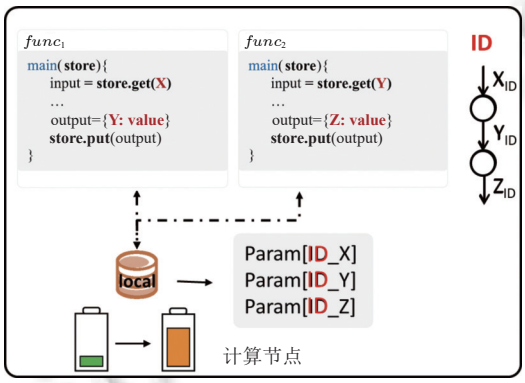


图 13 内存统一回收机制

3.3 工作流执行引擎

执行引擎维护每个工作流执行实例的运行状态, 其根据工作流实例的执行状态触发和调用相应的函数服务实例. 在执行过程中, 执行引擎通过函数服务平台的网关调用函数. 为了使工作流引擎能够根据运行状态和函数服务间的依赖触发相应的 FaaS 函数, 引擎引入了工作流元信息和运行时信息. 每个工作流调用都分配一个 CallID 来标识不同请求调用. 在每个工作流程结构中, 元信息记录工作流的数据依赖和执行依赖信息. 运行时记录函数的执行状态, 用于保证满足约束的调用触发任务.

如图 14 所示, 工作流引擎包含代理、分发器、管理器、执行器这 4 个模块. 4 个模块互相协作支持工作流实例的正确执行.

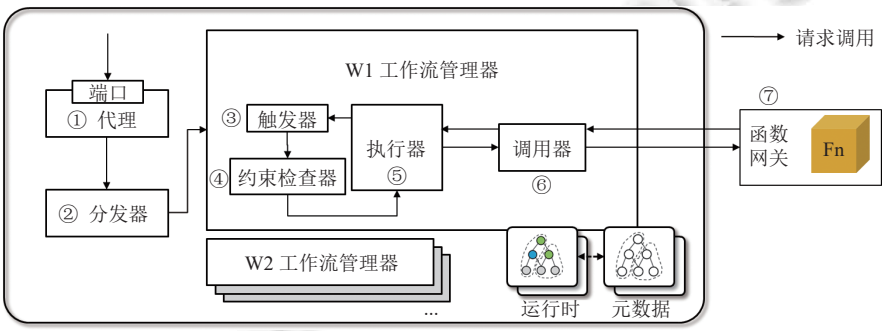


图 14 工作流执行引擎

(1) 代理. 作为工作流执行的代理负责接收工作流函数的触发请求, 当触发工作流起始函数任务时, 请求发送到代理, 代理将对请求进行处理, 代理负责注册工作流管理器.

(2) 分发器. 分发器的任务是根据请求的 CallID 以及工作流 ID 将请求交由相应工作流的管理器进行管理运

© 中国科学院软件研究所 http://www.jos.org.cn

行, 分发器也承担了部分管理器注册的任务.

(3) 管理器. 管理器是工作流执行的核心. 每个管理器有两个核心数据, 一是相应工作流的元信息以及函数间的依赖信息, 二是以 CallID 作为唯一性识别的请求执行实例的运行时信息. 管理器包括触发器、约束检查器等组件, 触发器负责根据依赖信息触发下一步函数, 约束检查器负责进行约束判断, 将当前运行状态与依赖信息进行对比, 判断当前步骤是否满足执行条件.

(4) 执行器. 执行器使用调用器, 负责和服务器无感知平台网关交互, 执行调用函数服务, 并在执行结束后获取执行成功与否状态. 如果失败, 执行器将采取指数退避重试策略, 对函数重复触发直至成功或超出时间阈值.

4 实验验证与分析

4.1 实验目的

实验主要目的在于分析评估 FineFlow 对于 FaaS 工作流应用的性能优化效果, 具体包括执行应用的时延、稳定性、吞吐量、响应时间等. 实验主要回答以下问题.

(1) FineFlow 对 FaaS 工作流应用执行过程中的数据传输时延和端到端时延的优化效果如何?

FineFlow 利用数据本地性优化数据传输时延并进一步优化工作流执行的端到端时延. 为了验证本文划分调度及数据存储方法的有效性, 本文设计了实验测量 FaaS 工作流执行过程中的数据传输时延和端到端时延, 并与现有工作进行对比.

(2) FineFlow 是否有效提升了 FaaS 工作流的执行性能?

本文分析对比 FineFlow 与基准系统 ServerlessFlow 在不同应用负载和网络环境下的应用执行性能, 如响应时间和吞吐量, 依此评估 FineFlow 提升 FaaS 工作流应用执行性能的有效性.

(3) FineFlow 的划分调度器带来的额外的性能和资源开销如何?

划分调度器在决策和调度方面的开销直接影响 FineFlow 的执行效率, 为此本文通过实验分析评估了划分调度器动态调度的性能以及运行时的内存开销.

4.2 实验设置

4.2.1 实验环境

本文采用塔式服务器和树莓派搭建了实验集群环境, 其中塔式服务器作为控制节点, 若干树莓派作为计算节点, 分别部署了 FineFlow 的相关组件和其他支撑环境相关的软件系统. 其中控制节点部署了划分调度器和执行引擎, 并作为远程集中存储节点部署 CouchDB. 计算节点部署函数服务运行支撑环境 OpenFaaS 以及支持本地存储的内存数据库 Redis 等. 表 2 展示了控制节点和计算节点的数量以及具体实验环境配置信息.

表 2 实验环境配置

配置	控制节点 (1个)	计算节点 (5个)
硬件	CPU: x86_64 Intel (R) Xeon (R) Silver 4 110 CPU @ 2.10 GHz Cores: 8, DRAM: 128 GB, Disk: 1 TB HDD	CPU: aarch64 ARM Cortex-A72 Cores: 4, DRAM: 8 GB, Disk: 64 GB SSD
软件	Operating system: CentOS Linux with kernel 6.0.0 Database: Apache/CouchDB: 2.3.1 Kubernetes version: 1.20.15 Containerd version: 1.6.9 runc version: 1.1.4	Operating system: Ubuntu Linux with kernel 5.15.0 Database: Redis: 7.0.4 Kubernetes version: 1.20.15 Containerd version: 1.6.18 runc version: 1.1.4
	OpenFaaS version: gateway: 0.23.0 faas-netes: 0.15.1	
容器	Container runtime: Python-3.7.16, Linux with kernel 5.15.0 Resource allocation: CPU (vCPU):MEM (GB) = 1:2 Function container limit: 10 for each function	

4.2.2 FaaS 工作流应用

本文从阿里云、Google、AWS 等云厂商提供的典型工作流应用样例以及 FaaSFlow 研究工作中使用的基础

© 中国科学院软件研究所 http://www.jos.org.cn

样例中选取了 5 个具有不同结构特点和应用类型的真实工作流应用作为测试负载, 具有典型性和代表性. 实验根据每个应用的功能, 使用相应类型的数据作为工作流输入, 表 3 总结了 5 个 FaaS 工作流应用的特点以及每个工作流实例所使用的输入数据大小. 5 个工作流应用的具体信息如下.

表 3 FaaS 工作流应用特点与实验数据

应用信息	VF	IR	FP	DS	DC
应用类型	视频处理	图片识别	文件处理	数据处理	数据清洗
结构特点	并行、分支循环	分支、并行	并行	并行循环	并行
节点个数	5	7	4	3	5
数据类型	.mp4	.jpg	.md	.txt	.csv
数据大小 (MB)	13.45	1	28.23	3.2	90.68

注: VF指Video-FFmpeg; IR指Illgal-Recognizer; FP指File-Processing; DS指Data-Statistics; DC指Data-Cleansing

(1) 视频处理工作流 (VF)

FFmpeg 是一套可以记录、转换数字音视频, 并将其转化为流的开源计算机程序. Video-FFmpeg 是基于 FFmpeg 的音频和视频处理应用程序. 工作流中相关函数调用 FFmpeg 对上传的视频进行并行转码. 工作流的实例结构如图 15 所示. 应用实现的源代码来自 Alibaba Function Compute (阿里云函数计算)[38].

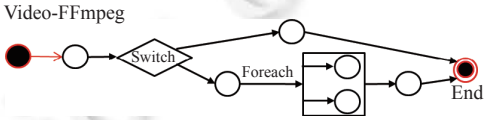


图 15 Video-FFmpeg 工作流

(2) 图像检测工作流 (IR)

Illgal-Recognizer 用于对上传图像进行风险内容识别. 它对图像进行成人与暴力违规检测, 并从上传的图像中提取文本, 最后再次检测风险性并对图像中违规风险内容进行马赛克模糊处理. 工作流的实例结构如图 16 所示. 源代码来自 Google Cloud Functions[39].

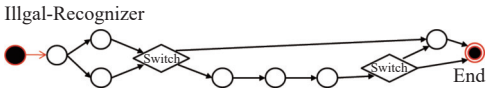


图 16 Illgal-Recognizer 工作流

(3) 文件处理工作流 (FP)

File-Processing 是一个实时文件处理应用, 文件上传将触发两个并行处理流, 一个用于将 Markdown 文件转换并保存为 HTML, 另一个用于检测文件情感. 工作流的实例结构如图 17 所示. 其源代码来自 AWS Lambda 用例[40].

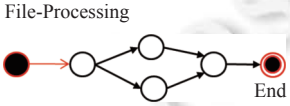


图 17 File-Processing 工作流

(4) 数据处理工作流 (DS)

Data-Statistics 是经典的 MapReduce 应用程序. 工作流的结构如图 18 所示[41].

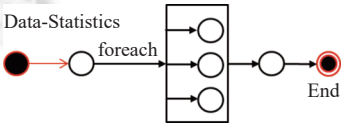


图 18 Data-Statistics 工作流

© 中国科学院软件研究所 http://www.jos.org.cn

(5) 数据清洗工作流 (DC)

数据清洗应用中包括的函数功能有消息转换、消息分割和消息富化等. 如图 19 所示, 来源于 Alibaba Function Compute (阿里云函数计算)[42].

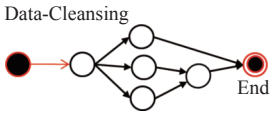


图 19 Data-Cleansing 工作流

4.2.3 实验基准

实验将 FineFlow 与以下基准系统进行比较.

(1) ServerlessFlow. 实验对比的基准是采用远程集中式存储的 FaaS 工作流系统, 本文命名为 ServerlessFlow. 如图 20 所示, ServerlessFlow 与 FineFlow 的主要不同在于, ServerlessFlow 采用和 OpenWhisk、AWS Step Functions 等 FaaS 工作流系统一致的集中式数据存储策略, 系统无本地数据存储, 数据的存取均采用远程存储.

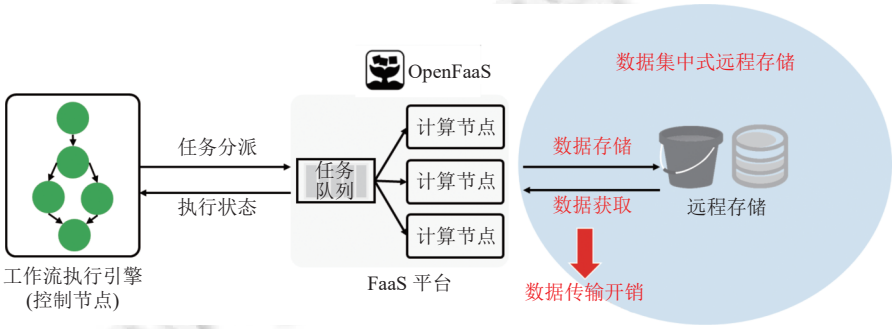


图 20 实验基准系统 ServerlessFlow

(2) FaaSFlow. 实验从当前对数据传输进行优化的研究工作中, 选取最新工作 FaaSFlow[2]作为比较对象. FaaSFlow 实现了基于函数粒度数据依赖的调度优化方法. 根据第 1.2 节的分析, 基于函数粒度的数据依赖分析会产生更多数据传输时延, 因此实验只重点分析对比了 FineFlow 和 FaaSFlow 基于不同粒度数据依赖分析对于工作流数据传输时延优化效果的差异. 实验分析和重现了 FaaSFlow 采用的函数粒度的数据依赖分析策略, 并基于 OpenFaaS 函数服务平台进行了相应的系统实现.

以上两个基准系统和本文实现的 FineFlow 之间的差别与关系总结如表 4 所示.

表 4 实验系统对比

实验系统	划分调度	数据本地性	细粒度依赖分析
ServerlessFlow	×	×	×
FaaSFlow	√	√	×
FineFlow	√	√	√

4.3 系统实验

实验分别在第 4.1 节搭建的集群环境上部署基准系统和 FineFlow, 并运行上述 5 个 FaaS 工作流应用. 通过在线下实际运行每个目标工作流应用 10 次, 收集统计工作流的历史运行时信息. 总体上, 实验通过改变工作流应用请求的生成, 实现“单应用-单实例”“单应用-多实例”“多应用-单实例”的执行负载情况. 同时, 为了避免冷启动对函数执行时延的影响, 每个函数至少保留 1 个副本.

4.3.1 工作流执行时延分析

(1) 数据传输时延

实验分别使用 FineFlow、FaaSFlow 和 ServerlessFlow 依次执行每个工作流应用, 为了避免 FaaS 应用并发执

© 中国科学院软件研究所 http://www.jos.org.cn

行带来的相互干扰, 该实验采用“单应用-单实例”的负载模式分别执行 5 个 FaaS 工作流应用, 确保任意时间最多只有 1 个工作流应用实例. 实验统计的数据传输时延为工作流应用中所有数据传输时间总和, 实验的网络带宽设置为 100 Mb/s.

实验结果如表 5 所示, 时延优化为 FineFlow 相比于 ServerlessFlow 和 FineFlow 分别减少的数据传输时延占原时延的百分比. 结果显示, 相比于没有任何优化策略的 ServerlessFlow, FineFlow 对于 FaaS 工作流应用的数据传输时延最高降低 74.6%, 平均降低 63.8%. 相较于 FaaSFlow, FineFlow 对于具有明显细粒度依赖特点的工作流应用, 如 Data-Cleansing, 其数据传输时延降低 28.4%. 由此说明, 虽然 FaaSFlow 对于数据传输时延也能够有一定程度的优化, 但是由于其对工作流执行函数采用了粗粒度的依赖分析, 导致各函数输出产生了相较于 FineFlow 更多的额外数据传输开销. 需要说明的是, 实验度量的数据传输时延是 FaaS 工作流中的所有数据传输时间总和, 而不仅仅是关键路径中的数据传输. 所以, 对于并行结构较多的工作流可能会出现总体数据传输时延优化量高于端到端时延优化量的情况.

表 5　函数工作流数据传输时延比较

数据传输时延及优化效果		VF	IR	FP	DS	DC
数据传输时延 (s)	ServerlessFlow	17.28	2.77	17.21	6.13	50.34
	FaaSFlow	6.22	1.75	4.41	2.36	21.27
	FineFlow	5.97	1.46	4.38	2.35	15.22
时延优化 (%)	相比于ServerlessFlow	65.5	47.3	74.6	61.7	69.8
	相比于FaaSFlow	4.0	16.6	0.7	0.4	28.4

(2) 端到端时延

实验同样分别使用 FineFlow、FaaSFlow 和 ServerlessFlow, 采用“单应用-单实例”的负载模式依次对每个工作流应用进行单独执行. 实验统计的端到端时延是指从工作流请求触发执行开始, 到接收到应用执行完成的响应的总体时间. 实验执行时的网络带宽同样设置为 100 Mb/s.

实验评估了 FineFlow 在降低 FaaS 工作流应用端到端时延方面的有效性.图 21 显示了 ServerlessFlow、FaaSFlow 和 FineFlow 分别执行每个工作流应用程序的端到端时延. 端到端时延由工作流中的关键路径执行时间决定. 实验结果显示, 相比于 ServerlessFlow, FineFlow 对于各工作流应用的端到端时延分别降低了 21.1%、8.4%、20.4%、31.9% 和 16.4%, 平均降低了 19.6%.

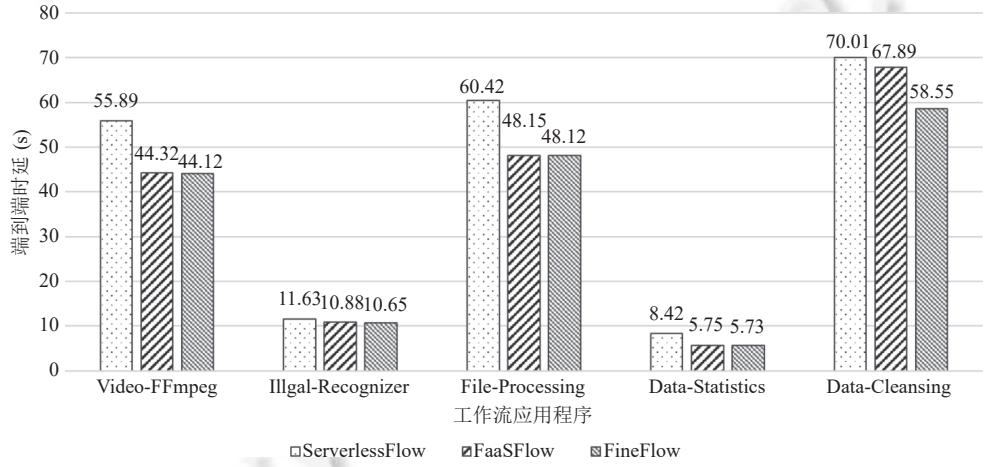


图 21　工作流单独执行的端到端时延

本文进一步地对比了 FaaSFlow 和 FineFlow 上 Data-Cleansing 端到端时延, 结果如图 22 所示. 通过分析可知, 对于明显具有细粒度数据依赖特点的工作流应用, 由于 FaaSFlow 粗粒度的依赖分析, 导致在关键路径上产生了额

© 中国科学院软件研究所 http://www.jos.org.cn

外的数据传输开销, 从而增加了工作流的端到端执行时延. 因此, FaaSFlow 对 Data-Cleansing 工作流的端到端时延仅优化了 3.0%, 远小于 FineFlow 的 16.4% 的优化效果. 相比于 FaaSFlow, FineFlow 对 Data-Cleansing 的端到端时延进一步优化了 13.8%.

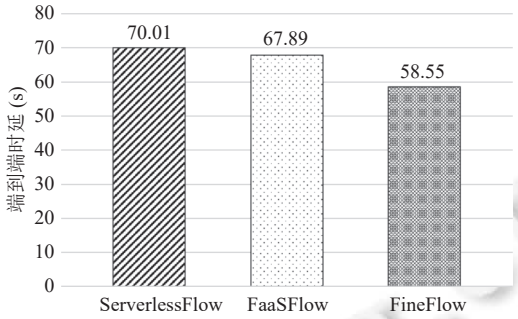


图 22 细粒度依赖工作流 Data-Cleansing 执行的端到端时延

(3) 工作流并发执行的端到端时延

实验采用了 5 个不同工作流应用并发的“多应用-单实例”的执行模式, 以分析不同工作流的并发执行对性能的影响.

图 23 显示了基于 ServerlessFlow 和基于 FineFlow 的 FaaS 工作流应用并发执行的端到端延迟. 当多个应用程序或实例在 ServerlessFlow 服务器无感知环境中并发运行时可能存在潜在的资源竞争, 尤其是有大量并行步骤的工作流应用程序时, 由于并行执行的函数更容易在相近时间内产生输出数据, 这将潜在地产生对网络带宽资源的竞争. 例如各工作流同时在 ServerlessFlow 上运行时, 其端到端时延相比于每个工作流的单独执行分别增加了 16.3%、56.5%、16.7%、88.1% 和 21.7%. 相比之下, FineFlow 可以有效缓解由于多应用并发执行时网络数据传输增加而带来的性能下降, 应用执行端到端时延分别增加 8.5%、31.1%、6.7%、11.7%、8.3%. 实验发现, 这是因为在 FineFlow 中, 经过划分调度优化后, 工作流应用程序中数据传输量大的关键路径函数被放置到了同一节点. 在这种情况下, 函数可以利用本地数据进行传输, 从而缓解网络带宽带来的瓶颈.

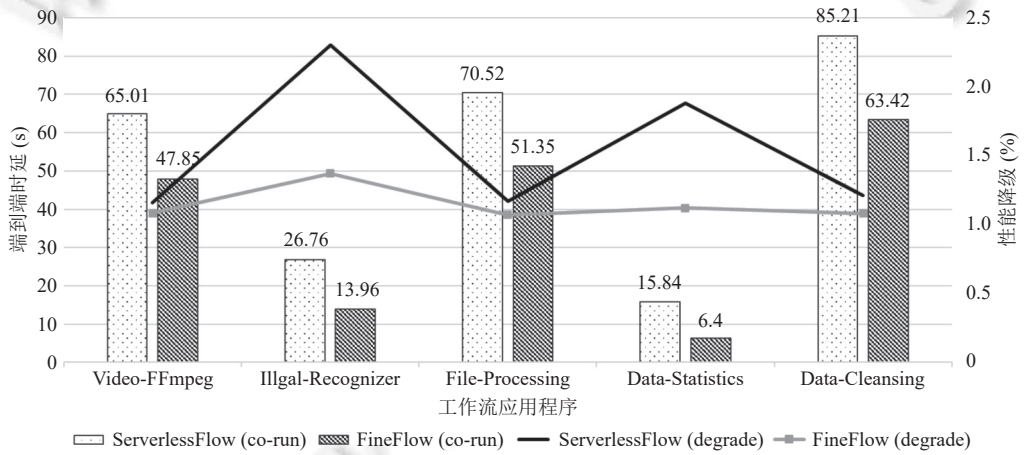


图 23 不同工作流并发执行的端到端时延

4.3.2 网络带宽影响分析

实验进一步分析了网络带宽波动对 ServerlessFlow 和 FineFlow 上工作流性能稳定性的影响. 实验选取工作流应用 Data-Statistics 进行了带宽敏感度测试, 将网络带宽设置为 100 Mb/s 到 5 Mb/s 之间的多种配置, 并对工作流

© 中国科学院软件研究所 http://www.jos.org.cn

进行 6 次/min 的调用. 实验采用“单应用-多实例”的负载模式, 系统环境内仅有 Data-Statistics 工作流的多个实例运行, 确保不存在其他工作流应用共享所设置网络带宽的情况. 实验在如图 24 所示的每种网络带宽设置下对工作流分别进行了 100 min 的持续调用, 并以这 100 min 内的 600 次调用的平均时延作为对应带宽下工作流的端到端时延. 实验结果如图 24 所示, 可以观察到, 伴随着带宽的下降, ServerlessFlow 的端到端时延有了明显增大, 即基于 ServerlessFlow 的数据密集型工作流应用对网络带宽变化高度敏感. 但是, 基于 FineFlow 的应用执行时延变化则趋于平稳, 几乎没有受到带宽的影响. 如表 6 所示, FineFlow 减少了工作流网络数据传输大小的 92.7%. 由以上实验结果可得, FineFlow 对本地数据的使用让工作流的执行时延对带宽波动的敏感度下降, 具有更好的鲁棒性. FineFlow 在其他 FaaS 工作流应用上的实验结果与针对 Data-Statistics 工作流的结果类似. 因此, 实验结果表明, 即使在带宽较小的情况下, FineFlow 仍然可以有效缓解其对数据密集型 FaaS 工作流应用执行带来的性能影响.

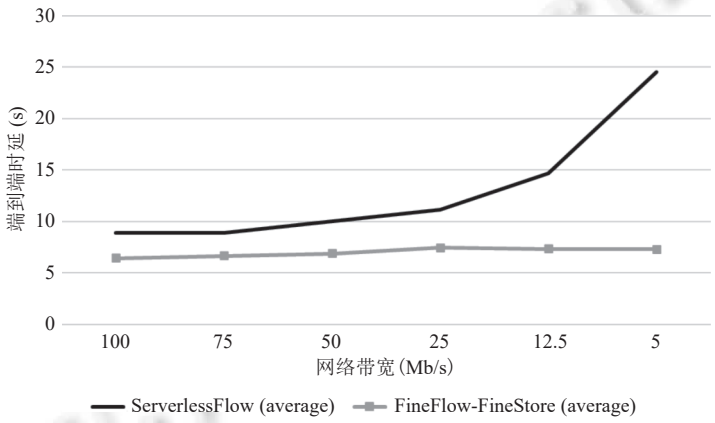


图 24 Data-Statistics 工作流不同带宽下的平均时延

表 6 Data-Statistics 工作流数据传输优化

数据传输量及优化效果		DS
数据传输量 (MB)	ServerlessFlow	4.24
	FineFlow	0.31
减少数据传输 (%)		92.7

4.3.3 不同请求负载和网络带宽下的 FaaS 工作流性能分析

该实验通过每分钟发送不同次数的工作流调用请求生成不同的测试负载, 统计 ServerlessFlow 和 FineFlow 上 FaaS 工作流的响应时间, 综合分析比较不同请求负载和网络带宽下 FaaS 工作流的性能表现.

以 Data-Statistics 工作流应用为例, 实验将网络带宽分别设置在 100 Mb/s 到 5 Mb/s 之间的 6 种情况进行测试. 实验记录 2–24 次/min 不同的请求频率 (进一步增加请求频率会导致大面积的执行超时) 下工作流执行的平均响应时间、中位数响应时间和尾部响应时间, 以此来较为全面地展示工作流的执行性能. 在 6 种不同的网络带宽设置下, 实验的每个并发测试均对工作流分别进行 100 min 调用, 并从这 100 min 的执行结果中计算得到平均响应时间、中位数响应时间和 99% 尾部响应时间. 需要说明的是, 本实验将超过 60 s 仍未响应的请求记为超时.

实验结果如图 25 所示, 与 ServerlessFlow 相比, FineFlow 系统上工作流执行的平均响应时间相对较低. 两者的平均响应时间和中位数响应时间在较低负载时的变化都趋于一致, 但伴随着负载量的上升, 尾响应时间会出现较大的差异. 相比之下, 基于 FineFlow 的应用执行表现得更加稳定. 当网络带宽减小时, ServerlessFlow 上应用的响应时间逐渐增高, 吞吐量逐渐下降, 这是因为 Data-Statistics 的函数之间存在着大量的数据传输, 当网络带宽小于 50 Mb/s 时, 并发执行的函数之间开始争夺有限的带宽资源, 导致性能下降显著.

© 中国科学院软件研究所 http://www.jos.org.cn

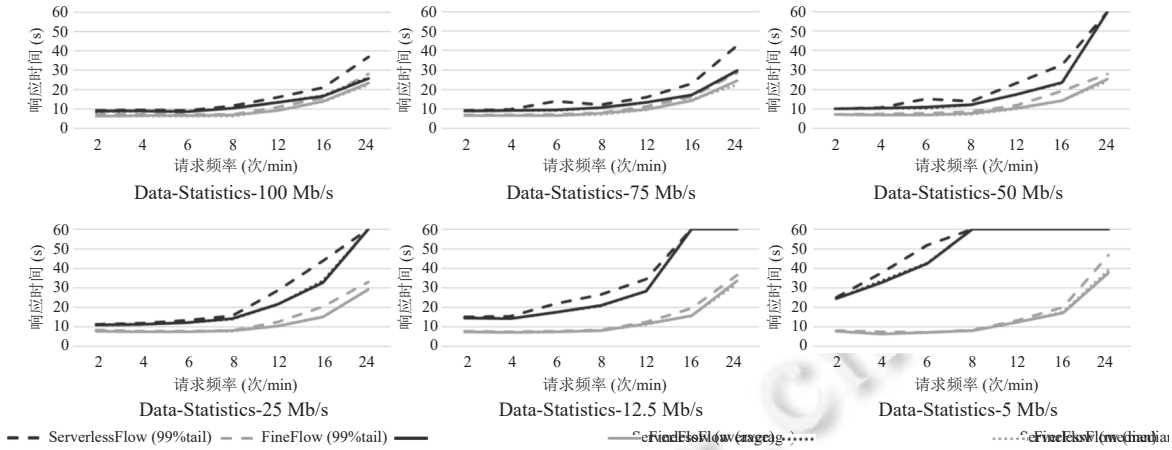


图 25 不同带宽和请求负载下 Data-Statistics 工作流的响应时间

从实验结果中可以观察发现, 在 2–8 次/min 的低访问频率下, 对于基于 FineFlow 的应用执行来说, 网络带宽的变化几乎不会对响应时间产生明显的影响. 如表 7 所示, 对比 8 次/min 下的 99% 尾时延实验结果可以看出, 伴随着带宽的减小, FineFlow 保持应用尾响应时间稳定性的优势越发凸显. 横向对比不同网络带宽下的实验结果, 在 12–24 次/min 这样相对较高的访问频率下, ServerlessFlow 在 100 Mb/s、75 Mb/s 带宽下的应用执行平均响应时间分别和 FineFlow 在 50 Mb/s、12.5 Mb/s 带宽下的应用执行平均响应时间相似. 这表明在相对高并发的情况下, 如果在执行过程中由于其他工作负载占用了大部分带宽, 使得网络带宽从 100 Mb/s 降至 25 Mb/s, 那么工作流 ServerlessFlow 的平均吞吐量至少会下降 40% 左右, 而 FineFlow 的平均吞吐量至多会下降 8% 左右. 需要说明的是, ServerlessFlow 在 5 Mb/s 带宽下的应用执行发生超时, 而 FineFlow 能够成功完成应用执行, 其 99% 的尾响应时延为 8.459 s.

表 7 8 次/min 下的 99% 尾响应时间数据对比

尾响应时延及优化效果		带宽 (Mb/s)					
		100	75	50	25	12.5	5
99%尾响应时间 (s)	ServerlessFlow	11.615	12.026	13.845	15.827	26.664	Timeout
	FineFlow	7.258	7.96	8.608	8.283	8.438	8.459
降低尾响应时间 (%)		37.5	33.8	37.8	47.7	68.4	—

注: “—”表示百分比无法计算

4.4 划分调度器的性能与资源开销分析

实验采用工作流生成器 WfGen[43], 生成 10–1 000 节点数量不等的工作流进行划分调度器的性能与资源开销实验. WfGen 来自 WfCommons 项目, 是一个支持科学工作流研究和开发的框架. WfGen 可以根据给定的任务数, 生成具有各种特征的模拟工作流, 并可配置生成工作流的执行时间长短、输入和输出数据大小等信息.

实验使用 SRASearch 工作流[44]来研究划分调度器的性能. 它是生物信息学领域的数据密集型应用. 工作流结构如图 26 所示, 4 个不同的函数功能节点分别对数据进行格式转换、对比和分析. 该工作流可生成不同节点规模的流程, 其结构语义为: 工作流只有 1 个 bowtie2-build 函数任务, 负责为数据建立索引, fasterq-dump 和 bowtie2 为可扩展的节点函数, 分别负责将 SRA 数据转换为 fastq 格式和对比数据序列, fasterq-dump、bowtie2 两者一一对应, 每 25 个 bowtie2 对应 1 个 merge 节点, 负责对其输出数据进行合并, 若 merge 数多于 1 个, 在其后会再增加 merge 节点用于最终的合并.

根据现有工作 Orion 发布的微软 Azure 云上部署工作流结构公开数据集[45], 可以发现, 其中 99.45% 的工作流节点数在 25 以下, 100% 的工作流节点数在 752 以下, 因此实验利用 WfGen 的工作流自动生成功能, 生成包括

© 中国科学院软件研究所 http://www.jos.org.cn

函数节点数量分别为 10、25、50、100、200、400、800、1 000 不等的工作流, 来分析划分调度器的可扩展性.

根据图 27 的结果分析, 随着功能节点数量 n 的增加, 划分时间开销大致遵循 $O(n^2)$, 且用时均低于 0.3 s, 节点数低于 25 的用时低于 6 ms. 实验还评估了工作流节点数量增加内存资源的使用情况. 结果发现, 调度器的内存使用量至少为 22.23 MB, 这是因为其中包含调度器的基础组件和数据. 综上所述, 对于当前 99.45% 的应用程序 (节点总数少于 25 个), 划分调度器能够表现出较高的性能.

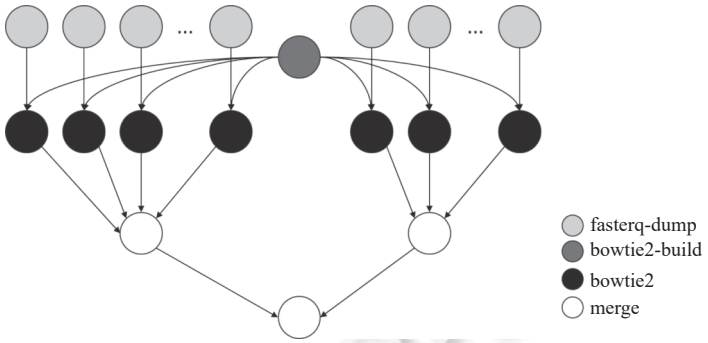


图 26 SRASearch 工作流

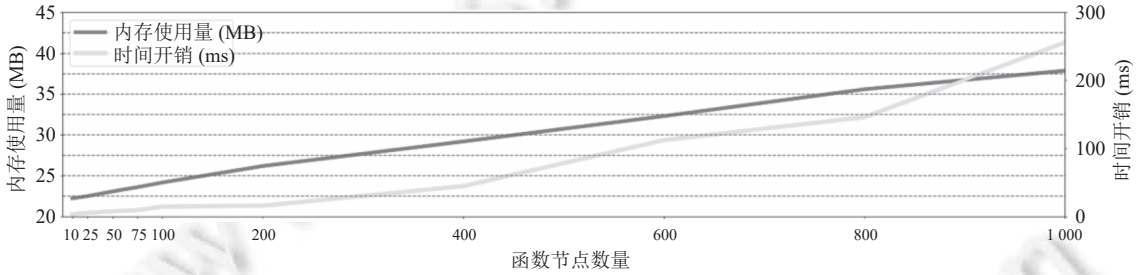


图 27 划分调度器的时间与内存开销

4.5 讨 论

由实验结果可知, FineFlow 基于细粒度数据依赖分析的 FaaS 工作流调度优化方法能够进一步优化数据传输时延和端到端响应时延, 对于网络带宽波动的影响有更好的鲁棒性. 但是, 尽管 FineFlow 验证了本文方法的有效性, 但它在以下方面仍然存在局限性和进一步优化的空间.

(1) 在 FaaS 工作流及其函数的运行时信息采集与建模方面, 当前主要采用的是基于历史平均值的统计与计算方法, 在描述和预测未来的运行时资源需求与变化的准确性方面存在局限性, 未来可基于深度学习等方法提高负载与资源消耗预测的准确度, 为 FaaS 工作流的划分调度提供更加准确的依据.

(2) 为了支撑数据本地存储和传输, 当前对于单个函数服务的部署与运行时动态伸缩限制在单一物理节点上, 从而避免函数实例分散到多个节点时在数据一致性和数据存储副本数量等方面带来的更高复杂度和资源开销. 为了进一步提高方法的适用性, 未来还需考虑如何应对上述更加复杂的场景.

(3) 应用负载和资源使用的动态变化需要 FineFlow 能够支持运行时的动态再调度, 未来将考虑通过实时监控和度量 FaaS 工作流执行延迟和节点资源消耗情况等指标来触发应用的动态调整, 以使应用的服务质量 (quality of service, QoS) 能够得到持续保障.

(4) 本文方法对于常见的具有细粒度数据依赖的工作流应用场景会产生更好的优化效果. 对于函数之间不存在细粒度数据依赖关系的工作流, 本文方法和基于函数粒度数据依赖分析的优化方法效果差异不大. 因此, 未来工作还将考虑如何有效融合上述两种情况下工作流的调度优化策略, 并进一步优化调度只存在函数粒度数据依赖

© 中国科学院软件研究所 http://www.jos.org.cn

的 FaaS 工作流应用.

最后, 本文工作主要从 FaaS 工作流的执行时延为优化目标, 未来工作可考虑从减少平台资源的总体开销和 FaaS 工作流的服务质量违背 (QoS violation) 等方面研究并提出相应的优化方法.

5 总 结

本文对目前 FaaS 工作流系统存在较大的数据传输时延的问题进行了深入研究, 提出了基于细粒度数据依赖分析的 FaaS 工作流调度优化方法, 设计了依赖敏感的数据存储策略, 并在此基础上实现了 FaaS 工作流执行系统原型 FineFlow. FineFlow 对工作流函数间的数据依赖关系进行细粒度分析和建模, 实现基于关键路径的 FaaS 工作流划分调度算法的工作流应用执行端到端时延优化. 实验结果表明, FineFlow 可以显著降低 FaaS 工作流应用的数据传输时延和应用执行的端到端时延. 对于具有细粒度数据依赖的工作流, FineFlow 的表现优于当前已有工作.

References:

[1] Liu LY, Tan HS, Jiang SHC, Han ZH, Li XY, Huang H. Dependent task placement and scheduling with function configuration in edge computing. In: Proc. of the 27th Int'l Symp. on Quality of Service (IWQoS). Phoenix: IEEE, 2019. 1–10. [doi: 10.1145/3326285.3329055]

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

[3] 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]

[4] 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. 2023. 1489–1504.

[5] Ao L, 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]

[6] Witte PA, Louboutin M, Modzelewski H, Jones C, Selvage J, Herrmann FJ. An event-driven approach to serverless seismic imaging in the cloud. IEEE Trans. on Parallel and Distributed Systems, 2020, 31(9): 2032–2049. [doi: 10.1109/TPDS.2020.2982626]

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

[8] AWS. AWS step functions introduction. 2024. https://aws.amazon.com/step-functions/?nc1=h_ls

[9] Microsoft. Azure durable functions documentation. 2024. https://learn.microsoft.com/en-us/azure/azure-functions/durable/

[10] Google Cloud. Workflows. 2024. https://cloud.google.com/workflows

[11] Alibaba Cloud. CloudFlow: Visualization, O&M-free orchestration, and coordination of stateful application scenarios. 2024. https://www.alibabacloud.com/en/product/serverless-workflow?_p_lc=1

[12] 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 USENIX Annual Technical Conf. Boston: ACM, 2018. 923–935. [doi: 10.5555/3277355.3277444]

[13] IBM. IBM cloud functions. 2024. https://www.ibm.com/products/functions

[14] Klimovic A, Wang YW, Stuedi P, Trivedi A, Pfefferle J. Pocket: Elastic ephemeral storage for serverless analytics. In: Proc. of the 13th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: ACM, 2018. 427–444. [doi: 10.5555/3291168.3291200]

[15] AWS. AWS S3: Object storage built to retrieve any amount of data from anywhere. 2024. https://aws.amazon.com/s3/

[16] AWS. Amazon ElastiCache: Real-time performance for real-time applications. 2024. https://aws.amazon.com/elasticache/

[17] 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, 2021. 973–988.

[18] 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, 2021. 957–971.

[19] 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 (ASPLOS 2021). New York: ACM,

© 中国科学院软件研究所 http://www.jos.org.cn

2021. 152–166. [doi: 10.1145/3445814.3446701]

[20] Viil J, Srirama SN. Framework for automated partitioning and execution of scientific workflows in the cloud. The Journal of Supercomputing, 2018, 74(6): 2656–2683. [doi: 10.1007/s11227-018-2296-7]

[21] Zheng G, Peng Y. GlobalFlow: A cross-region orchestration service for serverless computing services. In: Proc. of the 12th Int'l Conf. on Cloud Computing. Milan: IEEE, 2019. 508–510. [doi: 10.1109/CLOUD.2019.00093]

[22] AWS Samples. Lambda-refarch-imagerecognition. 2024. https://github.com/aws-samples/lambda-refarch-imagerecognition

[23] 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]

[24] 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, 2018. 57–69. [doi: 10.5555/3277355.3277362]

[25] 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 2020 USENIX Annual Technical Conf. USENIX, 2020. 205–218.

[26] Bermbach D, Karakaya AS, Buchholz S. Using application knowledge to reduce cold starts in FaaS services. In: Proc. of the 35th Annual ACM Symp. on Applied Computing. Brno: ACM, 2020. 134–143. [doi: 10.1145/3341105.3373909]

[27] 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 (ASPLOS 2021). New York: ACM, 2021. 386–400. [doi: 10.1145/3445814.3446757]

[28] 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 2020 ACM Symp. on Cloud Computing (SoCC 2020). New York: ACM, 2020. 1–15. [doi: 10.1145/3419111.3421286]

[29] Zhang JW, Zhou XC, Ge TY, Wang XD, Hwang T. Joint task scheduling and containerizing for efficient edge computing. IEEE Trans. on Parallel and Distributed Systems, 2021, 32(8): 2086–2100. [doi: 10.1109/TPDS.2021.3059447]

[30] Yuan YW, Bao ZQ, Yu DJ, Li WQ. Multi-scientific workflow scheduling algorithm based on multi-objective in cloud environment. Ruan Jian Xue Bao/Journal of Software, 2018, 29(11): 3326–3339 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5477.htm [doi: 10.13328/j.cnki.jos.005477]

[31] Zhou YM, Li ZJ, Ge JD, Li CY, Zhou XY, Luo B. Multi-objective workflow scheduling based on delay transmission in mobile cloud computing. Ruan Jian Xue Bao/Journal of Software, 2018, 29(11): 3306–3325 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5479.htm [doi: 10.13328/j.cnki.jos.005479]

[32] 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: ACM, 2018. 789–794. [doi: 10.5555/3277355.3277431]

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

[34] Pu QF, Venkataraman S, Stoica I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In: Proc. of the 16th Symp. on Network System Design and Implementation. Boston: USENIX, 2019. 193–206. [doi: 10.5555/3323234.3323251]

[35] 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]

[36] Tang Y, Yang JF. Lambdata: Optimizing serverless computing by making data intents explicit. In: Proc. of the 13th Int'l Conf. on Cloud Computing (CLOUD). Beijing: IEEE, 2020. 294–303. [doi: 10.1109/CLOUD49709.2020.00049]

[37] Apache. OpenWhisk. 2024. https://github.com/apache/openwhisk

[38] Alibaba Cloud. Use FFmpeg-based applications in function compute to process audio and video files. 2024. https://www.alibabacloud.com/help/zh/function-compute/latest/use-ffmpeg-in-function-compute-to-process-audio-and-video-files

[39] Google Cloud. ImageMagick tutorial (2nd gen). 2024. https://cloud.google.com/functions/docs/tutorials/imagemagick

[40] AWS Samples. Serverless reference architecture: Real-time file processing. 2024. https://github.com/aws-samples/lambda-refarch-fileprocessing

[41] Zhang SH, He BS, Dahlmeier D, Zhou AC, Heinze T. Revisiting the design of data stream processing systems on multi-core processors. In: Proc. of the 33rd Int'l Conf. on Data Engineering (ICDE). San Diego: IEEE, 2017. 659–670. [doi: 10.1109/ICDE.2017.119]

© 中国科学院软件研究所 http://www.jos.org.cn

[42] Alibaba Cloud. Use function compute to perform message cleansing. 2024. https://www.alibabacloud.com/help/en/functioncompute/latest/use-function-compute-to-perform-message-data-cleansing

[43] WfGen. Generation of realistic synthetic workflow traces with a variety of characteristics. 2024. https://wfcommons.org/generator

[44] WfCommons. Execution instances for SRA search workflow. 2024. https://github.com/wfcommons/pegasus-instances/tree/master/srasearch

[45] ICANForce. Orion-OSDI22. 2024. https://github.com/icanforce/Orion-OSDI22/tree/main/Public_Dataset

附中文参考文献:

[30] 袁友伟, 鲍泽前, 俞东进, 李万清. 云环境下基于多目标的多科学工作流调度算法. 软件学报, 2018, 29(11): 3326–3339. http://www.jos.org.cn/1000-9825/5477.htm [doi: 10.13328/j.cnki.jos.005477]

[31] 周业茂, 李忠金, 葛季栋, 李传艺, 周筱羽, 骆斌. 移动云计算中基于延时传输的多目标工作流调度. 软件学报, 2018, 29(11): 3306–3325. http://www.jos.org.cn/1000-9825/5479.htm [doi: 10.13328/j.cnki.jos.005479]

刘璐(1998－), 女, 硕士, 主要研究领域为软件工程, 服务计算.

吴国全(1979－), 男, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为软件工程, 软件测试与维护, 面向服务的计算.

高浩城(1999－), 男, 硕士生, 主要研究领域为软件工程, 服务器无感知计算.

魏峻(1970－), 男, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为软件工程, 分布式系统, 服务计算.

陈伟(1980－), 男, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为软件工程, 服务计算, 物联网.

© 中国科学院软件研究所 http://www.jos.org.cn