华 中 科 技 大 学

研究生课程考试答题本

考生姓名 郑永昌

考生学号 M202073486

系、年级 计算机学院 2020 级

类　　别 硕 士 生

考试科目 数据中心技术

考试日期 2020 年 12 月 26 日

评 分

题 号	得 分	题 号	得 分

总 分：	评卷人：

注意：1、无评卷人签名试卷无效

2、必须用钢笔或者圆珠笔阅卷，使用红色。用铅笔阅卷无效

两种无服务器计算框架及无服务器计算框架的定制

郑永昌

(华中科技大学, 计算机科学与技术学院, 武汉 中国 085400)

摘 要 近年来，一种新型云计算模式——无服务器计算（Serverless Computing，又名为“函数即服务”，Function as a Service, FaaS）得到了迅速发展。无服务器计算开辟了一种新的构建和量化应用程序和服务的方法，通过允许开发者打破传统的基于单机服务器的应用程序的束缚，能够使用粒度更细的云函数，这一方法得以实现。开发者只用关注函数逻辑的开发，而那些众所周知极其枯燥乏味的维护工作，比如供应、量化及管理云函数运行的后端服务器，交给云服务提供商来做就好。而且，无服务器计算拥有自动伸缩性和精细的计费粒度的优点。

本文介绍了两种在2020年ACM云计算研讨会SoCC上发表的无服务器计算框架，分别被命名为“悟空”和“光子”，以及一个无服务器计算框架定制化工具“ServerlessBench”。“悟空”通过将中心调度器的任务细分给大量的Lambda执行器，可以让任务并行调度，从而极大地提高了服务性能，减少了调度时的资源抢占，使得任务调度数据局部化，而且具有自动资源弹性和更低的成本优势。

“光子”是一个精简的无服务器计算框架，它加强了工作负载的并行性，来集中管理同一个运行时环境中相同函数的多个实例，减少了大量相同云函数调用时产生的运行时环境等应用程序状态的冗余性。并行调用可以透明地共享运行时环境和应用程序运行时状态，而没有牺牲运行安全性作为代价。“光子”每次调用减少了25%到98%的函数内存占用，同时相比于当今其它无服务器平台没有性能损失。“光子”还能减少30%的总内存利用率和52%的冷启动总数。

“ServerlessBench”是一个开源的定制化无服务器计算平台的开发套件，它包含了能够探索无服务器计算的个性化指标的许多测试用例，例如数据流通效率，启动延迟，无状态过载，以及性能隔离。它可以用来评估大多数主流的无服务器计算平台，包括AWS Lambda，OpenWhisk，Fn，以及新研发的无服务器模板。这些模板形成了设计指导大纲，可以帮助设计者优化无服务器平台，帮助应用开发者设计最适合这个平台的函数。

关键词 无服务器计算框架；“悟空”；“光子”；“ServerlessBench”

中图法分类号 TP274

Two Kinds of Serverless Computing Framework and Characterizing

Zheng Yongchang

Abstract In recent years, a new cloud computing model called serverless computing or Function as a Service (FaaS) has emerged. Serverless computing enables a new way of building and scaling applications and services by allowing developers to break traditionally monolithic server-based applications into finer-grained cloud functions. Developers write function logic while the service provider performs the notoriously tedious tasks of provisioning, scaling, and managing the backend servers that the functions run on. Nevertheless, Serverless computing promises auto-scalability and cost-efficiency in “pay-as-you-go” manner.

This paper proposes two kinds of serverless computing frameworks which were named Wukong and Photons separately and an open-source benchmark suite called “ServerlessBench” for characterizing serverless platforms. The key insight of Wukong is that partitioning the work of a centralized scheduler across a large number of Lambda executors, can greatly improve performance by permitting tasks to be scheduled in parallel, reducing resource contention during scheduling, and making task scheduling data locality-aware, with automatic resource

elasticity and improved cost effectiveness.

Photons is a redundancy-reduced framework leveraging workload parallelism to co-locate multiple instances of the same function within the same runtime. Concurrent invocations can then share the runtime and application state transparently, without compromising execution safety. Photons reduce function's memory consumption by 25% to 98% per invocation, with no performance degradation compared to today's serverless platforms. We also show that our approach can reduce the overall memory utilization by 30%, and the total number of cold starts by 52%.

ServerlessBench is an open-source benchmark suite for characterizing serverless platforms. It includes test cases exploring characteristic metrics of serverless computing, e.g., communication efficiency, startup latency, stateless overhead, and performance isolation. We have applied the benchmark suite to evaluate the most popular serverless computing platforms, including AWS Lambda, OpenWhisk, and Fn, and present new serverless implications from the study. These implications form several design guidelines, which may help platform designers to optimize serverless platforms and application developers to design their functions best fit to the platforms.

Key words serverless computing framework, Wukong, Photons, ServerlessBench

1 引言

近些年来，无服务器计算成为了云计算的新潮范例。许多公共云平台提供无服务器计算服务，包括亚马逊 AWS Lambda，IBM 云函数，微软 Azure 云函数，谷歌云函数等等，私有云平台就更多了。用户们如此青睐无服务器计算，背后有三个原因。第一，它帮助开发者集中精力于核心应用逻辑，由无服务器平台去支持基础设施相关的属性（例如自动可伸缩性），以及接管服务器管理工作。第二，用户能够在无服务器计算按需计费的模式下节省开支，例如云函数只会在请求调用的时候运行并计费。第三，用多个云函数设计开发一个应用具有模块化的优势。使用无服务器计算对云服务提供者也有好处，因为他们可以更高效地管理资源。

现已有支持并行作业处理的无服务器并行计算框架，然而，这些解决方案没有充分解决高效调度和数据局部性的性能问题，从而导致长扩展延迟、次优性能和较高的成本。为此，Benjamin Carver 等人设计并构建了一个新的无服务器并行计算框架，称为“悟空”。悟空是一个面向无服务器、去中心化、局部感知和成本效益高的并行计算框架。作者在 AWS 上对悟空进行了广泛的评价。他们的结果表明，悟空减少了多个数量级的网络 I/O，达到了 68.17 倍于 numpywren 的性能，同时降低了高达 92.96%的成本。

如今的无服务器平台分别初始化和调度每个函数调用，即使许多函数调用执行相同的代码并需要相同的环境。这种严格的隔离导致了两个主要的低效率：(a)调用之间没有内存共享，这增加了内存的总体利用率；(b)每个调用都必须初始化它的内存、自己的运行时和应用程序状态，这延长了执行时间。基于这一现象，Vojislav Dukic 等提出了“光子”：一个基于运行时和应用程序虚拟化的用于无服务器函数的超轻量执行上下文。光子提供与当今平台相同的无服务器抽象，同时允许安全运行时共享应用程序，这些应用程序使用无服务器的大规模并行性来并发运行许多相同操作的实例。光子利用运行时级别隔离为同一函数的配置调用提供自动数据分离，同时仍然使用现有的虚拟化技术，如 Docker 或 Firecracker 对不同的函数执行严格的内存隔离。作者在开源无服务器平台 Open Whisk 中实现了光子，并演示了它们的好处。实验表明，对于常见的无服务器工作负载，光子每次调用的内存消耗减少了 25%到 98%。使用来自 Microsoft Azure 无服务器基础设施的跟踪模拟，证明光子可以将

整个集群内存利用率降低30%，同时减少总冷启动数为52%。

尽管新出现的无服务器计算开始流行，对于系统软件设计者来说，设计一个高效、可伸缩和用户友好的无服务器平台仍然具有挑战性。因此，一个能够揭示无服务器计算的关键指标和描述无服务器平台的基准套件对于无服务器平台设计者和用户来说都是很有必要的。所以，Tianyi Yu 等提出了一种用于无服务器计算的通用开源基准套件 ServerlessBench，它帮助系统开发人员设计和评估他们的无服务器系统，并为应用程序开发人员构建其无服务器应用程序提供了有用的提示。Tianyi Yu 等在一个商业无服务器平台(AWS Lambda)、两个开源无服务器平台(Open Whisk 和 Fn)和一个生产中的私有云(Ant Financial)上进行了评估，然后提出了对无服务器计算特征度量的新发现。这些发现可以指导无服务器平台和应用程序的设计。

2 背景

2.1 无服务器平台

在无服务器计算中，计算单元是一个函数。应用程序开发人员将其函数发送到无服务器平台，该平台为要运行的功能提供沙盒执行环境（容器或虚拟机）。平台将无服务器功能与运行时一起脱机编译，然后初始化环境，并在请求到达时调用处理函数。函数的一个需求是无状态的——函数不应该依赖于跨请求的状态。需要无状态的一个原因是用以实现自动可伸缩性，因为无服务器平台需要根据请求流量增减函数实例数。

2.2 函数启动

在现有的无服务器平台中，函数实例可以使用冷启动或热启动处理请求，这取决于是否有可用的空闲沙盒。函数的第一次执行总是从冷启动开始，它需要准备函数映像（例如 Docker 映像）、创建沙盒（例如 Docker 容器）和加载函数代码。冷启动通常会导致较长的延迟。当函数实例完成执行时，沙箱可能会暂停指定的时间，以便为后续相同的函数请求提供不间断服务，即热启动。

2.3 函数组成

无服务器应用程序开发人员倾向于将复杂的应用程序解耦为松散耦合的无服务器函数的组成，以追求细粒度的可伸缩性和模块化开发部署。有两种功能组合类型：序列函数链和嵌套函数链。通常，平台提供的或第三方协调器（例如 IBM Action Sequences 和 AWS Step Functions）负责执行序列函数链。序列链中的每个函数通过将结果数据传输到序列中的下一个函数来完成它们的工作。在嵌套链中，函数调用嵌套函数并等待嵌套函数的结果才能返回。这些组成方法提供了不同的灵活性和性能特点，因此应根据实际应用需要仔细选择。

2.4 无服务器计费

无服务器计算中的现收现付方式计费对无服务器用户来说是一个非常有吸引力的因素。执行账单（无服务器计算中的主要成本）可以用 C*时间*资源计算，其中 C 是特定于平台的常量，时间是具有毫秒的函数执行时间级粒度（AWS Lambda, IBM 云函数和Google 云函数中是 100ms，Azure 函数中是 1ms），资源通常表示提供给函数的内存和 CPU 资源（在少数情况下，如 Azure 函数，它表示平均内存使用量）。现有的无服务器平台，包括 AWS Lambda、Google 云函数和 IBM 云函数，通常按照与充足内存的一定比例为函数分配 CPU 计算资源。

2.5 一般约束和限制

服务提供商对使用云资源进行限制，以简化资源管理。以 AWS Lambda 为例：用户捆绑配置 Lambda 的内存和 CPU 资源。用户可以以 64MB 增量选择 128MB - 3008MB 之间的内存容量。然后，Lambda 将按配置的内存量的比例线性分配 CPU 功率。每个 Lambda 函数最多可运

行 900 秒，

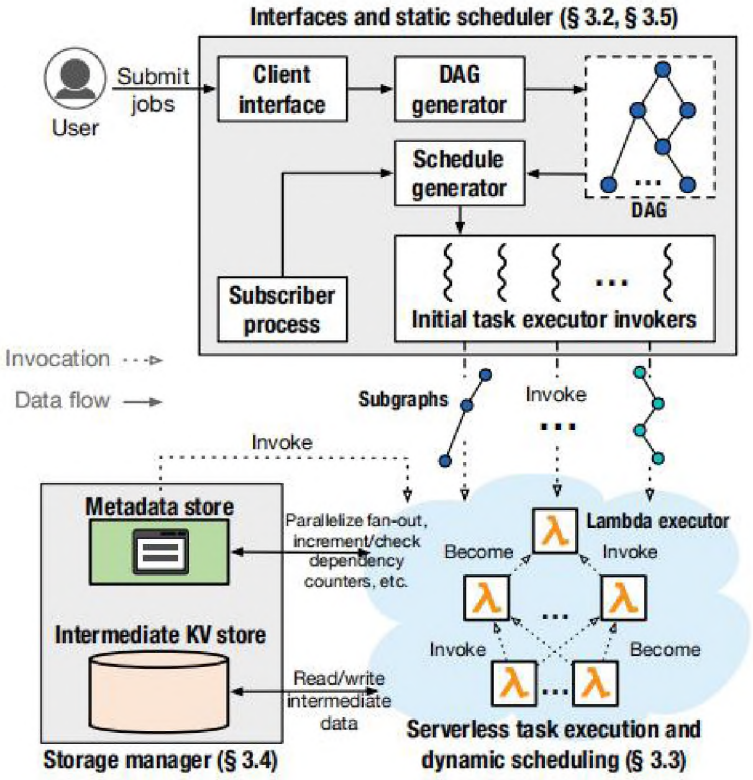

图 1 悟空架构图

当达到时限时将被强制停止。此外，Lambda 只允许出站 TCP 网络连接，并禁止入站连接和 UDP 协议。

3 “悟空”的设计

3.1 高度抽象设计

图 1 为悟空的高度抽象设计。该设计包括三个主要组件：运行在 Amazon EC2 上的静态调度生成器、Lambda 执行器池和存储集群。

悟空的调度是分散的，采用静态调度和动态调度相结合的方法。静态调度是 DAG 的子图。每个静态计划被分配给一个单独的执行器。执行器使用动态调度来强制执行静态调度中任务的数据依赖关系。一个执行器可以调用额外的执行器来增加任务并行性，或者集群任务来消除它们之间的任何通信延迟。执行程序将中间任务执行结果存储在一个弹性的内存中键值存储(key-value storage, KVS)集群中，和与作业相关的元数据则存放在我们称之为元数据存储(MDS)的独立 KVS 中。

3.2 静态调度

悟空用户向 DAG 生成器提交 Python 计算作业，DAG 生成器使用 Dask 库将作业转换为 DAG。静态调度生成器从 DAG 生成静态调度。对于具有 n 个叶节点的 DAG，n 个静态调度被生成。叶节点 L 的静态调度包含所有任务节点，这些节点可以从 L 和所有边缘进入和离开这些节点。任务节点的数据包括任务的代码和任务输入数据的 KVS 键。使用从 L 开始的深度优先搜索(DFS)很容易计算 L 的调度。当一个最终结果存储在 Redis 中之后，会向静态调度程序的定期进程发送消息，然后 Lambda 执行器会通知静态调度程序。当收到消息时，静态调度程序将下载最终结果并自动返回给用户。

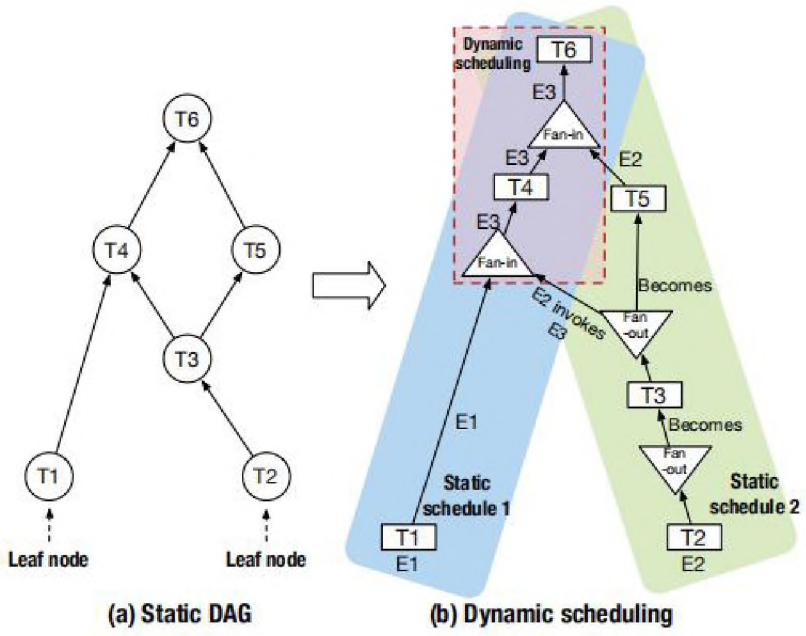

图 2 静态 DAG 和动态调度

图 2(a)显示了具有两个叶节点的 DAG。图 2(b)显示了从这个 DAG 生成的两个调度：调度 1（蓝色）和调度 2（绿色）。

静态调度包含三种类型的操作：任务执行、扇入和扇出。为了简化我们的描述，当 DAG 任务 Tx 后面紧跟着 Ty，而 Tx(Ty)没有扇出（扇入）时，作者在静态调度的 Tx 和 Ty 之间添加了一个平常的扇出操作。这个扇出操作有一个从 Tx 向内的边缘和一个到 Ty 的向外的边缘，即没有实际的扇出。在图 2(a)和(b)中，DAG 任务 T2 和 T3 就是这种情况。

扇入任务 T 可能取决于将由不同的执行器执行的任务，例如图 2 中的任务 T4。 下面描述的动态调度技术确保了 T 的数据依赖得到了满足，并且 T 仅由一个执行器执行。还请注意，静态调度不会将其任务映射到处理器；此映射当一个执行器函数实例被静态调度时，由 AWS Lambda 平台在执行时自动完成的，并由 AWS Lambda 放置在 VM 上。

3.3 扇出操作的动态调度

对于扇出操作有两种情况：

情况 1：n 个（其中 n>1）扇出边缘中没有一个是扇入边缘。然后 E“成为”其中之一的执行器。例如 T，通过执行 T，E“调用”其他扇出任务的执行器。

情况 2：扇出边中的一个或多个也是扇入边。例如，图 2(a)中的扇出节点 3 具有扇出边缘，也是节点 4 的扇入边缘。选择 E 的“become”边缘是基于扇出边缘所针对的任务的即时可用性。如果没有任务目标依赖项得到满足，那么没有任务目标可以立即执行，并且没有一个扇出边缘可以选择为 E 的“become”边缘（扇入边缘具有扇入操作，将在下一步执行）；否则，为可用的目标任务选择一个扇出边缘作为“become”边缘。

3.4 扇入操作的动态调度

如果任务执行器 E 执行具有 n 个 in 边缘（其中 n>1）的扇入操作，则 E 和剩下(n-1)个执行器共同参与此扇入操作，以查看其中哪一个将在扇入的 out 边缘上继续其静态调度（例如，图 2 中的节点 4）。作为被 E 执行的任务 T 的扇入操作，E 原子性地得到而且更新 KVS 中的一个值，该值跟踪在执行期间已满足的 T 的输入依赖项的

数量。这有两种情况：

情况 1：T 的所有输入依赖项都已满足。然后 E 通过执行 T 继续其静态调度；

情况 2：T 的所有输入依赖项都没有得到满足。然后 E 将 T 所需要的中间对象发送给存储管理器。 在图 2(b)中，每个扇入任务都标记为此任务的执行者。

4 “光子”

4.1 启用了光子的无服务器平台

光子是在任何无服务器平台上和任意语言运行时都可以实现的通用原语。然而，为了清晰起见，作者在运行于修改版 OpenWhisk 之上的 JVM 实现上下文中描述它们。光子被设计成具有较小的封装和快速启动的特性。为此，作者通过保留活动线程池来重新利用以前光子中已经存在的线程来执行新线程。在执行过程中，光子可以产生多个线程以及子进程，类似于今天的无服务器平台。此外，同一执行环境中的所有光子共享相同的对象堆，节省了为每个光子分配私有内存的开销，并使光子容易地共享应用程序状态成为可能。最后，所有光子也共享应用程序运行时代码缓存，这意味着在代码准备阶段产生的所有优化代码（包括代码解释、对本机代码的分析和编译）有益于所有光子，使得它们能够更快地执行。

4.2 数据隔离

光子保持现有无服务器运行时的语义：调用是隔离的，即使它们共享运行时。为了在同一运行时的多个函数之间进行数据分离，必须写入静态字段的本地特定于调用的副本。此外，对于每一个新的函数，静态初始化器必须独立运行，最后，在函数执行终止时，必须删除其本地字段副本以避免内存泄露。

函数加载器：为了支持数据分离，保留当前的无服务器抽象，并允许用户平滑地过渡到光子，我们实现了一个函数加载程序，该函数加载程序可以拦截和监听用户字节码。函数加载器自动插入适当的操作并修改对全局静态程序元素的访问。字节码转换是通过安装基于 Javassist 的类加载程序,在类加载时执行的。此类加载器将加载所有应用程序类，并确保所有静态字段都被正确隔离。请注意，作者的方法不是修改运行时——函数加载程序只是 JVM 的一个包装器，而且负责：（1）字节码转换；（2）初始化静态元素；（3）清理私有状态。

代码转换：私有状态（静态字段的本地副本）是通过静态表替换应用程序代码中的所有静态字段来构建的，给定一个唯一的函数执行标识符，静态表返回唯一的本地字段副本。除了替换所有静态字段声明外，还对所有读写进行修改以使用通过表可访问的字段的本地版本。最终静态字段被忽略，因为不可能进一步修改，因此不可能通过这些字段来破坏数据分离。

算法 1 是用于隔离私有状态的转换的简化版本。首先，引入静态表，每个静态字段有一个（第 15 行）。然后，对于类静态初始器（第 16 行）、构造器（第 18 行）和方法（第 20 行），所有静态字段访问都通过静态表（第 7 行和第 9 行)来进行。我们省略了这两个 Convert_Map_* 函数的完整实现，但这些函数分别将字段写入 o.f=v 转换为 o.f.put(photon_id, v)，将字段读出 v=o.f 转换为 v=o.f.get(photon_id)。最后，删除静态字段（第 22 行）。

运行私有初始化器：类静态初始化器由一系列指令组成，JVM 保证在创建相应类的实例，访问静态字段，或调用静态方法之前执行这些指令。通过设计，静态初始化器在应用程序执行的整个生命周期中只运行一次。因此，为了保证在同一运行时的多个函数调用的数据分离，静态初始化器需要为每个初始化器独立执行。确保所有本地版本的静态字段都已经在对应的表中声明并已正确初始化非常重要。

Algorithm 1 Data Separation at Class Loading Time

```
1: static_initializers ← []
2: procedure Isolate_Field_Accesses(code_block)
3:   for field_access in code_block do
4:     field ← field_access.field
5:     if isStatic(field) and notFinal(field) then
6:       if isWrite(field_access) then
7:         Convert_Map_Write(field_access)
8:       else
9:         Convert_Map_Read(field_access)
10: procedure Load_Class(class)
11:   fields_to_remove ← []
12:   for field in class do
13:     if isStatic(field) and notFinal(field) then
14:       fields_to_remove.add(field)
15:       class.addField(Map.class, field.name())
16:   Isolate_Field_Accesses(class.static_initializer)
17:   for constructor in class do
18:     Isolate_Field_Accesses(constructor)
19:   for method in class do
20:     Isolate_Field_Accesses(method)
21:   for field in fields_to_remove do
22:     class.remove(field)
23:   initializer ← Clone(class.static_initializer)
24:   for field_access in initializer do
25:     field ← field_access.field
26:       if isFinal(field) and isWrite(field_access)
then
27:       initializer.remove(field_access)
28:   static_initializers.add(initializer)
```

为此，作者将类静态初始化器克隆到每次创建新光子时都可以调用的方法中（第 23 行）。此外，作者从克隆的静态初始化器中删除了静态最终字段的初始化，因为这些初始化应该只在函数调用中初始化一次（第 27 行）。

清理私有状态：在共享应用程序运行时，我们必须清理每个函数执行的私有状态，以避免内存泄漏（即保持可访问但未使用的内存有效）。特别是，所有静态字段的本地版本都需要从各自的表中删除，以便清除已经完成的函数执行的状态。为了自动实现这个功能，作者使用了弱表。弱表是一种常见的编程语言抽象，只要对相应的键有很强的引用，就可以保持值的可达性。为了使用弱表来存储本地静态字段，每当函数执行上下文变得不可达（发生在函数执行终止时），我们依靠垃圾回收机制来自动清理无法到达的本地字段。

局限性：类加载器和字节码通常受顺序和假设的约束，如果应用程序本身安装了另一个类加载器，或者使用反射，这些假设可能会失效。在这种情况下，用户必须手动修改代码并更改对静态字段的访问。从我们的经验来看，只需要很少的修改，并且只需要较小的本地化代码编辑。在测试工作负载中，只需要对一个库进行手动更改，其中在 MinIO 库的两个类中少于 10 行的代码需要更改，因为它们使用反射来访问静态字段。

4.3 共享应用状态

与数据隔离一样，允许配置的函数调用有效地共享状态也很重要。由于共享状态将同时对于多个光子可见（对于读和写），我们提供了一组抽象来帮助开发人员管理共享状态。除了允许开发人员共享应用程序状态外，还可以使用这些原语进行协调共享资源的访问，如文件系统。我们刻意简化了这些抽象，使它们足以使光子共享应用程序状态；更先进和对开发者友好的抽象可以构建在这些抽象之上。

共享对象存储：这是共享对象的键值映射。此映射驻留在与所有并发函数调用相同的地址空间中，并可用于以很小的性能开销共享数据。

独占访问：这是一个锁定原语，可用于协调对共享对象存储的访问。例如，当需要将特定对象插入到共享对象存储中时，可以获取存储上的锁，以确保没有数据竞争。此外，还可以使用

锁定原语轻松构建细粒度的读/写锁，以进一步减少访问共享对象存储中的特定对象时的资源争用。

唯一标识符：此唯一光子标识符可用于创建私有临时状态。例如，如果函数需要在本地文件系统中使用临时文件，则光子标识符可用于标识私有文件或文件夹。

表 1 给出了一个 Java 光子函数类的简单示例。主方法包含三个参数：(a)具有函数参数的 JSON 映射，(b)用于与其它光子共享状态的对象存储，(c)标识符。 第 14 行显示如何使用光子标识符创建私有临时文件，第 16 至 23 行显示共享资源如何初始化或者取自共享对象存储的示例。为了简单起见，我们使用了 Java 中可用的同步块，但也可以使用更复杂的锁定方案（例如细粒度锁定）。 其他的语言也提供类似的锁定原语。

```
1  class MyServerlessFunction {
2  private static String run(
3     MyModel model,
4     String input,
5     File pvtfile) { ... }
6
7  public static JsonObject main(
8     JsonObject args,
9     Map<String, Object> store,
10    String photonId)
11 {
12  MyModel model;
13  String input = args.getString("input");
14  File pvtfile = new File("/tmp/" + photonId);
15
16  synchronized (store) {
17    if (!store.containsKey("model")) {
18       model = new MyModel(...);
19       store.put("model", model);
20    } else {
21       model = store.get("mode");
22    }
23 }
24
25 String result = run(model, input, pvtfile);
26 JsonObject response = new JsonObject();
27 response.addProperty("predicted", result);
28 return response;
29 }}
```

Listing 1: Example of a Photon function.

外部隔离：光子仅在运行时内提供数据隔离，而配置的调用仍然可以不受限制地共享同一个文件系统。 这种方法简化了跨函数调用的数据共享，但同时也产生了与当今平台的兼容性问题。例如，如果函数更改了全局 Linux 环境变量，则会在函数调用间形成一个竞争条件并导致正确性问题。

为了避免外部冲突和不一致，用户可以利用运行时中光子提供的独占访问原语和唯一标识符来协调对外部系统元素的更改。

5 “ServerlessBench”

Tianyi Yu 等使用 ServerlessBench 就一些无服务器计算的评价指标对几大主流无服务器计算平台进行了评估。

5.1 数据流通延迟

图 3 中的结果表明，在所有三个平台上，在传输的图像大小(0KB-30KB)的小范围内，通信延迟增长非常缓慢。 因为数据传递方法发生了变化，由于 AWS Lambda 的数据传递限制，从直接数据传递到通过 S3 的间接数据，通信延迟在 35KB 时急剧上升。将交互函数放置在同一个节点上进一步激发了数据传递效能，例如使用本地

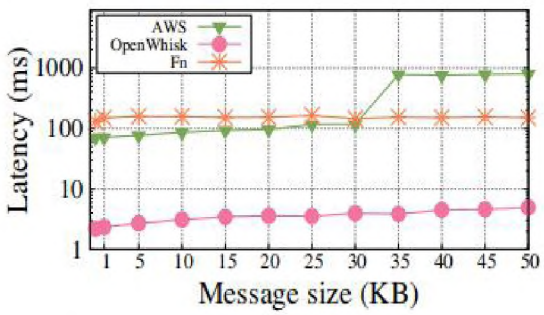

图 3 不同信息大小的数据流通延迟

消息总线或共享内存进行数据传输。因此，与函数之间没有传递数据的情况相比，我们认为在函数之间传递有限大小的数据，可能只增加可忽略的通信开销。这种观察可以通过状态传递的方式促进可能的无服务器执行优化。

5.2 启动延迟

表 2 显示了 OpenWhisk 中启动过程的每个阶段所花费的时间。在典型的冷启动中，使用镜像拉取（当函数不指定容器镜像标记或使用“最新”标签）时，镜像拉取（表 1 中的“镜像拉取”）是成本的主要来源，占总启动延迟的 81%。此外，沙箱初始化（表 1 中的“Docker Run”和“Docker init”)占用了启动延迟的 20%（或者在使用本地镜像时超过 95%，并且没有镜像开销），在此期间，Docker 将函数映像解压缩到本地存储中，准备隔离环境（例如命名空间和 cgroups），并启动语言运行时（例如 Python 和 JVM）调用的函数。热启动大约需要 135ms，这比冷启动的镜像拉取快 35 倍（或比没有镜像拉取的冷启动快 7 倍）。在热启动中，“Docker 停止”成为成本的主要来源，占启动总延迟的 71%。在这个阶段，无服务器平台用相同的函数映像唤醒一个暂停的容器来处理新的请求。Open Whisk 还提供了一个热启动机制，其中请求重用容器保持一个刚刚完成的函数实例而不暂停或者取消暂停。使用 OpenWhisk 热启动，函数启动时间可以减少到 10ms 以下。 然而，OpenWhisk 会在 50ms 后暂停函数容器，因为它会执行完成。因此热启动情况仅适用于请求在 50ms 间隔内到达，因为可重用函数实例完成的时间间隔是 50ms。

	Applications	Routing	Load balance	Msg queue	Image pull (optional)	Docker run	Docker init	Send Arg.	Sum
Cold start (ms)	Complex Java	30.7	3.0	0.8	3645.7	749.3	158.4	18.7	960.9 (+3645.7)
	Complex Python	30.3	3.0	0.9	3812.8	733.0	266.6	2.1	1035.9 (+3812.8)
	Hello Java	30.5	2.6	0.6	3609.0	744.2	159.7	18.0	955.6 (+3609.0)
	Hello Python	30.5	2.8	0.9	3763.6	741.0	259.7	1.2	1036.1 (+3763.6)
	Applications	**Routing**	**Load balance**	**Msg queue**	**Prepare container**	**Docker unpause**	**Prepare Arg.**	**Send Arg.**	**Sum**
Warm start (ms)	Complex Java	30.3	2.7	0.8	0.2	97.2	1.8	2.3	135.4
	Complex Python	30.2	2.7	0.7	0.3	96.0	1.7	3.3	135.2
	Hello Java	30.0	2.4	0.9	0.3	95.6	1.4	2.5	133.0
	Hello Python	30.1	2.6	0.8	0.3	96.7	1.7	1.8	134.4

表 2 OpenWhisk 启动延迟

结果（图 6）表明，具有较高隔离级别的沙箱（例如，Hyper Container，gVisor）通常具有较长的启动延迟，因为沙箱运行时更臃肿。Firecracker 是一种由 AWS Lambda 设计和使用的轻量级虚拟机。除了 Node.js 应用程序之外，它在所有评估的应用程序中具有最短的启动延迟。

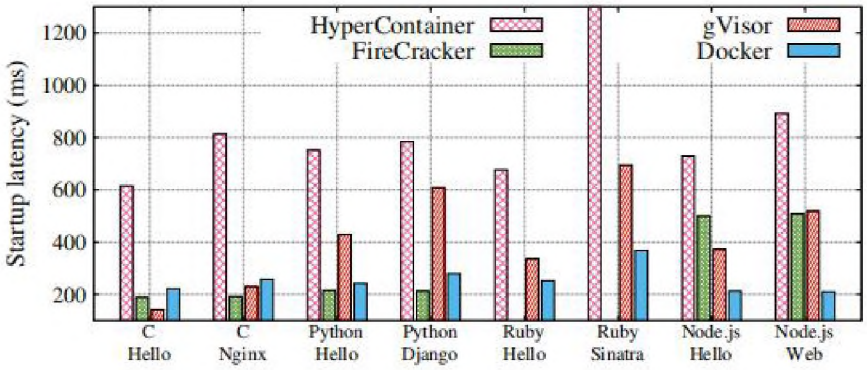

图 6 不同沙箱的启动延迟

5.3 性能隔离

作者进行了一个比较实验，以评估 DB-缓存应用程序在有或没有 CPU 密集型工作负载的情况下的性能。图 7 的结果显示，DB-cache 实例与 Alu 应用程序共同运行时有 109%的延迟。

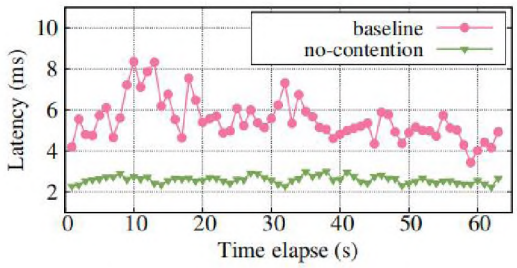

图 7 有无 CPU 争用的性能比较

6 总结与展望

无服务云计算几乎了处理所有系统管理操作，使得程序员能够更加容易地使用云。无服务计算提供了一个接口极大地简化了云编程，并且代表了从汇编语言到高级编程语言的转变。2009 年伯克利对云计算的预测，揭示了无服务计算的 motivation，描述了扩展无服务当前限制的应用程序，然后列出了发挥无服务计算潜力所遇到的阻碍和机会。就像 2009 年的论文确定了云的挑战并预测它们将得到解决，以及云会越来越被广泛使用一样，我们预测这些问题是在未来可被以解决的，并且无服务计算将成为云计算未来的主导。

未来无服务计算的两个挑战在于如何提高安全性和适应来自特殊用途处理器的成本性能改进，在这两种情况下，无服务计算的特性可能有助于解决这些挑战。物理上的共存是侧道攻击的必要条件，但是在无服务计算中很难确定函数的位置，而且很容易地进行位置随机化。云函数是使用高级语言进行编程的，例如 JavaScript、Python 或 TensorFlow，这些高级语言提升了编程的抽象层级并且更加容易进行创新，进而使得底层的硬件可以带来开销和性能的提升。

参 考 文 献

[1] Benjamin Carver, Jingyuan Zhang, Ao Wang, et al. Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing//SoCC '20, Virtual Event, USA, 2020: 1-5

[2] Vojislav Dukic. Photons: Lambdas on a diet// SoCC '20, Virtual Event, USA, 2020: 45-51

[3] Tianyi Yu, Qingyuan Liu, Dong Du, et al. Characterizing Serverless Platforms with ServerlessBench.//SoCC '20, Virtual Event, USA, 2020: 9-13