**Serverless Computing Summerize**

目录

[摘要 1](#_Toc535143686)

[第1章 介绍 2](#_Toc535143687)

[第2章 性能测试 5](#_Toc535143688)

[2.1 资源调度 5](#_Toc535143689)

[2.1.1 可扩展性和实例放置 5](#_Toc535143690)

[2.1.2 冷启动和VM配置 6](#_Toc535143691)

[2.1.3 实例生命周期和闲置实例回收 6](#_Toc535143692)

[2.2 性能隔离 7](#_Toc535143693)

[2.2.1 CPU利用率 7](#_Toc535143694)

[2.2.2 I/O吞吐量和网络吞吐量 8](#_Toc535143695)

[2.3 短暂存储 8](#_Toc535143696)

[2.4 本章小结 9](#_Toc535143697)

[第3章 冷启动 11](#_Toc535143698)

[3.1 冷启动过程 11](#_Toc535143699)

[3.2 冷启动测试 12](#_Toc535143700)

[3.3 冷启动优化 13](#_Toc535143701)

[3.3.1 代码优化 13](#_Toc535143702)

[3.3.2 保持热启动 14](#_Toc535143703)

[3.3.3 SOCK 17](#_Toc535143704)

[3.4 本章小结 18](#_Toc535143705)

[第4章 短暂存储 20](#_Toc535143706)

[4.1 短暂存储的问题 20](#_Toc535143707)

[4.2解决方案 21](#_Toc535143708)

[4.2.1 Pocket 21](#_Toc535143709)

[4.2.2 Selecta 23](#_Toc535143710)

[4.4本章小结 26](#_Toc535143711)

[第5章 思考与总结 28](#_Toc535143712)

[参考文献 30](#_Toc535143713)

# 摘要

serverless 是一种兴起的应用程序部署体系结构，其中应用程序的资源供应以及可扩展性由第三方服务管理。本文首先评估了目前几个主流无服务器平台的一些评估与测试。本文重点分为了两部分内容，一部分描述了无服务器的冷启动问题，描述关于冷启动进行的测试，并寻找可以优化相关缺陷的方案，另一部分描述了无服务器的短暂存储问题并根据该问题介绍了一些优化方案。本文最后给出了一些思考。

# 第1章 介绍

Serverless完全向租户隐藏服务器管理，租户对应用程序运行配置的访问很少，这允许租户专注于开发他们的函数——专用于特定任务的小型应用程序。一个函数通常在具有受限资源（如CPU时间和内存，AWS Lambda根据CPU内存分了好几个套餐）的专用函数实例（容器或其他类型的沙箱）中执行；Serverless通常由一到多个函数组成，这些函数是独立、小型、无状态的组件，它们通常由某种脚本语言编写的一小段代码指定。Serverless强调function-as-a-service（FaaS），与传统的infrastructure- as-a-service（IaaS）平台中的Virtual Machine（VM）不同，只有在调用函数时才会启动函数实例，并在处理请求后立即进入休眠状态。租户按每次调用收费，无需支付未使用和闲置的资源[1]。

以 Serverless 服务平台之一的AWS Lambda为例，租户只需要考虑怎么用代码提供价值即可，甚至连可扩展、蓝绿部署等一系列的问题都不用考虑，Amazon 优秀的运营工程师已经帮助租户打造了这一系列的基础设施。并且与传统的 AWS 服务一样，如 Amazon EC2 云服务器，它们都是按流量算钱的。如果要对一个运行的函数收费，考虑到的只有运行时间、CPU、内存占用、硬盘这几个条件。可针对于不同的需求，提供不同的 CPU 是一件很麻烦的事。对于代码来说，一个应用占用的硬盘空间几乎可以忽略不计。当然，这些应用会在你的 Amazon S3 云存储上有一个备份。于是，诸如 AWS 采用的是运行时间 + 内存的计算方式，具体收费标准如表1所示：

|  |  |  |
| --- | --- | --- |
| 内存 (MB) | 每个月的免费套餐秒数(s) | 每 100ms 的价格 (USD) |
| 128 | 3,200,000 | 0.000000208 |
| 192 | 2,133,333 | 0.000000313 |
| 256 | 1,600,000 | 0.000000417 |
| … | … | … |
| 1024 | 400,000 | 0.000001667 |
| … | … | … |

表1：AWS Lambda的套餐规格

在运行程序的时候，AWS 会统计出一个时间和内存，如下所示：

*REPORT*

*RequestId: 041138f9-bc81-11e7-aa63-0dbab83f773d*

*Duration: 2.49 ms*

*Billed Duration: 100 ms*

*Memory Size: 1024 MB*

*Max Memory Used: 20 MB*

其中 Memory Size 即是租户选用的套餐类型，Duration 即是运行的时间，Max Memory Used 是租户运行应用时占用的内存。根据租户Max Memory Used的数值及应用的计算量，可以很轻松地计算出我们所需要的套餐[2]。

因此，Serverless computing的弹性和细粒度计费优势吸引了更广泛的应用，如：交互式数据分析。在Serverless computing平台上运行分析工作负载的关键挑战是在任务之间有效地共享数据。以刚刚提到的交互式数据分析为例，它的特点就是一个不断循环的过程，交互式数据分析首先输入一些离散的目标，得到一些问题，根据得到的问题分析自己的数据问答这些问题，然后生成新的问题并重复之前的动作。与由响应事件触发器而执行的单个任务组成的简单事件驱动应用程序相比，分析通常由多个阶段组成，并且需要在任务阶段之间共享中间结果。

在一些传统的分析框架（例如，Spark，Hadoop）中，缓冲本地存储中的中间数据，并直接通过网络在任务之间交换数据。相比之下，Serverless computing框架通过要求task无状态来实现高弹性和可扩展性。换句话说，任务的本地文件系统和子进程仅限于任务本身的生命周期。此外，由于Serverless平台不公开对任务的调度和放置控制，因此难以直接在任务之间进行通信。于是，解决任务间通信的自然方法是将中间数据存储在公共的远程存储服务中，这种存储称为Ephemeral Storage，即短暂存储。将任务之间交换的数据称为短暂数据。在Serverless分析作业中有多种存储选择可用于这种数据共享，当然不同选项都提供不同的成本，性能和可扩展性权衡。像Amazon的S3（Simple Storage Service）这样的托管对象存储服务为提供商管理的存储资源提供了按需使用的容量和带宽，虽然它主要用于长期数据存储，但也可用于短暂数据。 而像Redis和Memcached这样的 in-memory key-value 存储以DRAM的高成本来提供高性能，它们还要求用户管理自己的存储VM。

显然，Serverless平台具有这几个优点：（1）运行方便；（2）价格合理，减少租户运营成本，降低开发成本；（3）实现快速上线，更快的部署流水线，更快的开发速度；（4）自动扩展，适应微服务架构，系统安全性更高。但同样的，serverless带来了相应的几个问题：

**不适合长时间运行应用**：Serverless 在请求到来时才运行。这意味着，当应用不运行的时候就会进入 “休眠状态”，下次当请求来临时，应用将会需要一个启动时间，即冷启动。如果租户的应用需要一直长期不间断的运行、处理大量的请求，那么可能就不适合采用 Serverless 架构。在这种情况下，采用 EC2 这样的云服务器往往是一种更好的选择。因为 EC2 从价格上来说，更加便宜。

**冷启动时间**：如上所说，Serverless 应用存在一个冷启动时间的问题。第二章本文将详细展示serverless平台下一些性能测试结果并在第三章深入探讨冷启动问题。

**完全依赖于第三方服务**：当决定使用某个云服务的时候，也就意味着可能走了一条不归路。在这种情况下，只能将不重要的 API 放在 serverless 上。当已经有大量的基础设施的时候，serverless 对于租户来说，并不是一个好东西。当采用 serverless 架构的时候，就意味着和特别的服务供应商绑定了。假设某个租户使用了 AWS 平台的服务，那么该租户再将服务迁到 Google Cloud 上就没有那么容易了。首先他们需要修改一下系列的底层代码，能采取的应对方案，便是建立隔离层。这意味着，在设计应用的时候，就需要隔离 API 网关，隔离数据库层等等。这些也将带给租户一些额外的成本，可能带来的问题会比解决的问题多。

**不透明的API**：完全依赖于第三方服务的另一个问题是，便于使用的API背后，是不透明的复杂的基础设施和管理系统。这就会导致租户有理由对供应商提供的服务表示担忧：比如平台说不同函数实例之前是隔离的，那么，隔离的质量如何？供应商的服务器在受到DDoS等攻击时展现的安全性怎么样？程序性能如何？

针对这些问题，本文首先在第二章描述了各个服务器平台的性能测试结果，包括资源调度（2.1节）、性能隔离（2.2节）和短暂存储的性能（2.3节）；随后，本文针对serverless平台的关键瓶颈——冷启动问题进行了细致的描述和评估（3.1和3.2节），根据冷启动问题进行了一些思考并罗列了目前的一些解决方案，并对冷启动的延迟进行了一些思考；最后，本文介绍了serverless中短暂存储的问题，和冷启动延迟一样，短暂存储问题也会影响serverless供应商和租户的经济效益，针对这个问题介绍了两个解决方案并进行了自己的一些思考。

# 第2章 性能测试

为了研究流行的serverless平台的资源管理机制和效率的快照，为开发人员提供了构建更可靠平台的性能基准和设计选项，并帮助租户改进无服务器平台的使用。Liang Wang等人对AWS Lambda，Azure Functions和Google Cloud Functions这三个无服务器平台进行了大规模的测试。通过在这三个服务中启动了50,000多个函数实例，以表征其架构，性能和资源管理效率。Serverless的特性使得serverless分析应用程序十分适用于serverless应用场景，而serverless分析应用程序的关键性能之一是任务之间的通信和数据交换。为了研究无服务器任务之间进行通信和数据交换，Ana Klimovic等人在AWS Lambda平台上通过使用三种不同的serverless分析应用程序（并行软件构建，MapReduce排序和视频分析）进行了大量的测试。2.1、2.2和2.3节分别描述了他们的实验结果。

## 2.1 资源调度

下文展示了三个无服务器平台在实例冷启动延迟、生命周期、可扩展性等方面，实例和VM是如何被调度的[1]。

### 2.1.1 可扩展性和实例放置

正如这些厂商声称的那样，响应需求变化的弹性自动缩放是serverless模型的“卖点”之一，因此需要衡量平台扩展的程度。

在支持并发执行方面，AWS是三项服务中最好的。N个并发调用总是产生N个并发运行的函数实例。 AWS可以轻松扩展到200个（最大测量并发级别）新函数实例。在实例放置方面，AWS Lambda似乎将它为bin-packing问题，采取了一种贪心策略，尝试在现有活动VM上放置新的函数实例以最大化VM内存利用率，即实例内存大小之和除以3,328（测量到的最大VM聚合内存）。

Azure文档声明它将自动扩展到最多200个实例，用于基于Nodejs的单个函数，并且每10秒最多可以启动一个新的函数实例。 但是，在测试中，无论如何更改调用之间的间隔，最多只能看到10个函数实例同时为单个函数运行。 所有请求都由一小组函数实例处理。 并发运行的实例都不在同一个VM上。 因此，似乎Azure不会尝试在同一个VM上共同定位相同函数的函数实例。

Google未能提供所需的可扩展性，即使Google声称HTTP触发的函数会迅速扩展到所需的调用率。 通常，即使对于低并发级别（例如并发数为10），也只有大约一半的预期实例数量可以同时启动，而其余的请求仍在排队中。

### 2.1.2 冷启动和VM配置

冷启动时间是无服务功能在一段时间没有使用后响应第一个请求所需要的时间。研究结果表明，要维持性能不变对所有的供应商来说都是一个挑战。云供应商一般会不间断地运行一组一般性的 worker（即 worker pool）。第一个进站的请求获得其中一个实例，该实例负责处理这个请求。实例在处理完第一个请求后保持运行状态。不过，保持运行的时间长短因供应商不同而不同。在冷启动比较中本文描述了对AWS，Azure和Google研究的两种冷启动事件：（1）在新的VM上启动的函数实例；（2）在已有的VM上启动的函数实例。直观上第一种比第二种有更长的冷启动延迟，因为第一种可能涉及启动一个新的VM，但是实验证明第一种只比第二种稍长一点。

租户希望函数更新之后的请求应该由新的函数代码处理，特别是在更新对安全性要求较高时，但是我们在AWS上发现，请求有可能被旧版本的函数处理。本文称这种情况为不一致的函数使用。使用冷启动来表示启动新函数实例的过程。对于无服务器平台，冷启动可能涉及启动新容器，设置运行时环境以及部署函数，与重用现有函数实例（热启动）相比，这将花费更多时间来处理请求。因此，冷启动可以显着影响应用程序响应性，进而影响用户体验。具体实验结果见3.2节。

### 2.1.3 实例生命周期和闲置实例回收

将函数实例保持活动的最长时间定义为实例生命周期。 租户当然更希望长寿命，因为他们的应用程序将能够更长时间地保持内存状态（例如，数据库连接）并且受到更少冷启动的影响。为了有效地使用资源，serverless提供商关闭空闲实例以回收分配的资源。我们定义实例在关闭之前可以保持空闲的最长时间作为实例最大空闲时间。

在更长的生命周期和更短的空闲时间之间存在权衡，因为维护更多空闲实例会浪费VM内存资源，而更少的准备服务实例会导致更多冷启动。

Azure函数实例的生命周期显着长于AWS和Google。在AWS中，所有设置的实例生命周期中值为6.2小时，最大值为8.3小时。 AWS中的主机VM通常寿命更长：观察到的最长VM内核正常运行时间为9.2小时。当请求频率增加时，实例寿命反而趋于变短。Google总体生命周期偏短，更大的内存往往具有更长的生命周期。例如，当每五秒调用一次时，Google的128MB和2,048 MB内存的90％实例的生命周期分别为3-31分钟和19-580分钟。因此，对于在繁重工作负载下具有小内存的函数，Google似乎积极地启动新实例而不是重用现有实例，这可能会增加冷启动的性能损失。

对于空闲实例的回收，在AWS中，一个实例通常可以保持不活动状态最多27分钟。 如果VM上存在活动实例，则实例可以保持非活动状态较长时间。通过每10秒发送一个请求，在给定的VM上保证一个实例处于活动状态，会发现：AWS仍然采用相同的策略来回收相同函数的空闲实例，但是观察到一些空闲实例在这种情况下可能会闲置1-3个小时。而在Azure中，我们无法找到一致的最大实例空闲时间。 我们通过在不同的时间重复了几次实验，发现最大空闲时间分别为22,40和120分钟。 在Google中，实例的空闲时间可能超过120分钟，120分钟后，18％的实验中实例仍然保持活跃状态。

## 2.2 性能隔离

本次研究中还研究了三个无服务平台的性能隔离。性能隔离主要分为两个方面，一方面是CPU利用率，另一方面是I/O和网络。

### 2.2.1 CPU利用率

不同供应商对于如何根据特定的 CPU 需求来调整内存都存在差异，例如，AWS 给配备较高内存的实例提供更多的 CPU 周期。Google 也采用了类似的策略，而 Azure 对于 CPU 分配的策略则不同，“4-vCPU 的虚拟机将分配更多 CPU”。测试方法是测量函数使用time.time()（Python）或Date.now()（Nodejs）连续记录1,000 毫秒的时间戳数据，实例的CPU利用率定义为这1000毫秒时间戳记录的分数，即在1,000毫秒的时间里使用了多少时间的CPU。

根据AWS，函数实例的CPU功率与其预先配置的内存成比例。但是AWS没有详细说明如何为实例分配CPU时间。所以论文进一步研究了如何在共存实例之间分配CPU时间，检查结果发现运行的实例公平地共享CPU，因为它们具有几乎相同的CPU利用率。 因此，AWS尝试仅基于函数内存为实例分配固定数量的CPU周期。

Google采用与AWS相同的机制来根据函数内存分配CPU周期。在Google中，随着函数内存的增加，实例CPU利用率的中值从11.1％到100％不等。 对于给定大小的memory，不同实例的速率的标准偏差非常低。 Azure的CPU利用率差异较大（14.1%-90%），中值为66.9%，即使为实例分配了相同数量的内存也是如此。4-vCPU虚拟机上的实例往往获得更高的CPU份额，而1-vCPU VM和2-vCPU VM上实例的利用率分布实际上是相似的; 但是，当共存函数实例增加时，1-vCPU VM上的实例的CPU利用率下降得更厉害。

### 2.2.2 I/O吞吐量和网络吞吐量

在AWS中，虽然聚合I/O和网络吞吐量保持相对稳定，但随着共存实例的增加，每个实例在I/O和网络资源中的份额都会减少。在Azure中，实例的I/O和网络吞吐量也随着colevel的增加而下降，并且由于来自其他coreident实例的争用而波动。更有趣的是，资源分配是根据函数实例恰好安排在哪种类型的VM来区分的。 在Google中，随着函数内存的增加，测量的I/O和网络吞吐量都会增加。但是Google在具有相同内存大小的不同实例测量的网络吞吐量会发生显着变化。例如，在2,048 MB函数实例中测量的网络吞吐量在0.2 Mbps和321.4 Mbps之间波动。论文发现关于吞吐量有以下两种情况：（1）所有实例吞吐量在给定时间段内波动，与内存大小无关；（2）单个实例暂时受到吞吐量下降的影响。情况（1）可能是由于网络状况的变化，而情况（2）可能是GCF租户实际上共享主机所以收到资源争用的影响。更具体的测量方法和测试结果可以参考论文详细内容[1]。

以上的测试结果说明，AWS和Azure无法在共存实例之间提供适当的性能隔离，因此争用可能会导致性能显着下降。 AWS将来自同一帐户的函数实例打包到VM上，意味着扩展函数会将相同的函数放在同一个VM上，从而导致资源争用和延长执行时间，会造成更长的冷启动延迟），Azure具有类似的问题。 Azure另外一个问题是VM之间的争用在帐户之间产生，还为服务攻击的交叉租户降级提供了可能性。

## 2.3 短暂存储

Ana Klimovic等人在AWS Lambda平台上通过使用三种不同的serverless分析应用程序（并行软件构建，MapReduce排序和视频分析）进行了大量的测试。使用AWS Lambda作为serverless平台，使用三种不同类型的存储系统来分析应用程序性能：基于磁盘的托管对象存储服务（Amazon S3）、基于DRAM的内存中的键值存储（ElastiCache Redis）和基于Flash的分布式存储系统（带有ReFlex Flash后端的Apache Crail ），以下是他们的测试结果和分析：

**对延迟敏感的作业**：三种应用程序中，只有并行软件构建显示对存储延迟的一些敏感性，因为访问的大多数文件都低于100 KB。实验结果表明，与S3相比Redis存储的延迟更低，最多可达100个并发lambdas；但是随着增加并发性，S3和Redis的运行时会收敛，因为作业最终会在AWS Lambda上受到计算限制。

**并行性有限的作业**：虽然serverless 平台允许用户通过启动许多并发lambda来利用高应用程序并行性，但是具有固有受限并行性（例如由于lambda之间的依赖性所导致）的作业可能经历lambda资源瓶颈（例如，memory、计算及（或）网络带宽限制）而不是存储瓶颈。软件构建过程的第一阶段具有高并行性，因为每个文件都可以独立地进行预处理、编译和组装。但是，后续存档和链接文件的lambdas取决于早期阶段的输出。无论存储系统如何，作业需要大约30秒才能完成，这是因为优化I/O不会影响具有特别高的计算延迟的lambda，这正是难点所在。

**吞吐量密集型作业**：MapReduce排序是一种具有充分并行性的I/O密集型应用程序。使用S3作为输入/输出文件，并将性能与S3、Redis、CrailReFlex作为短暂存储进行比较。将短暂数据存储在远程DRAM（Redis）或远程Flash（Crail-ReFlex）中可以提供类似的端到端性能，因为在存储集群中提供了足够的带宽，并且瓶颈变成了lambda CPU的使用率。随着我们增加lambda的数量，性能会呈线性扩展。视频分析是另一种具有充分并行性的应用程序。Redis和CrailReFlex相比，从S3读取和写入短暂数据会增加执行时间。由于Flash上的读写干扰，Crail-ReFlex在第二阶段的读取时间比Redis高。这是因为第一阶段的一些lambdas会更早完成和启动第二阶段。 因此，对于某些第二阶段lambda的I/O读取会干扰仍在运行的第一阶段lambdas的写入请求。

## 2.4 本章小结

本章对无服务器的关键性能描述了大量的测试结果，延迟问题成了性能测试中的一项关键因素。根据Liang Wang等人的测试结果，在资源调度方面，冷启动是各个厂商策略选择的首要因素。冷启动可以显著影响应用程序的响应性，进而影响用户体验；此外实例生命周期和闲置实例回收的策略选择主要取决于冷启动，I/O吞吐量和网络吞吐量也可能会因为冷启动问题导致不必要的资源争用和延长执行时间。高冷启动意味着冷启动的时间可能比函数运行花费的时间还要长，特别是，如果正在编写一个低延迟的交易应用程序，在这个时候由于高冷启动使用FaaS平台显然不是一个明智的选择。

另一方面，serverless的特性吸引了分析应用程序。本章关于无服务器的短暂存储性能进行了描述。Ana Klimovic等人的研究测试为期望的短暂存储属性提出了建设性的意见。为了支持serverless平台上的数据密集型分析，他们的分析推动了短暂存储服务的设计，该服务支持存储容量和吞吐量的自动和细粒度分配。对于他们研究的三种不同应用，吞吐量比延迟更重要，并且为闪存存储在性能和成本方面提供了良好的平衡。

在本文接下来的两章中，讲对冷启动和短暂存储两个问题进行细致的描述。在第3章中，本文首先介绍冷启动过程和冷启动成因，并描述了冷启动在三大主流无服务商的表现情况。接下来，本文将描述一些针对高冷启动问题的几个优化方案。在第4章中，本文首先介绍根据2.3节的测试结果所展示的目前主流无服务商平台短暂存储的问题期望的短暂存储特性，并介绍了一些现有的解决方案。

# 第3章 冷启动

本文在第二章解释了冷启动是无服务器平台中影响性能的一项关键瓶颈，这一节中，本文将继续深入冷启动，分析冷启动的成因（3.1节）以及针对冷启动更细致的实验结果（3.2节），最后着重讲解了一些关于冷启动问题的思考和优化（3.3节），包括代码优化、热启动和SOCK——一种适用于serverless优化的容器，同时在4.3.2节介绍的Selecta工具也提到了一些关于冷启动优化的方法。

## 3.1 冷启动过程

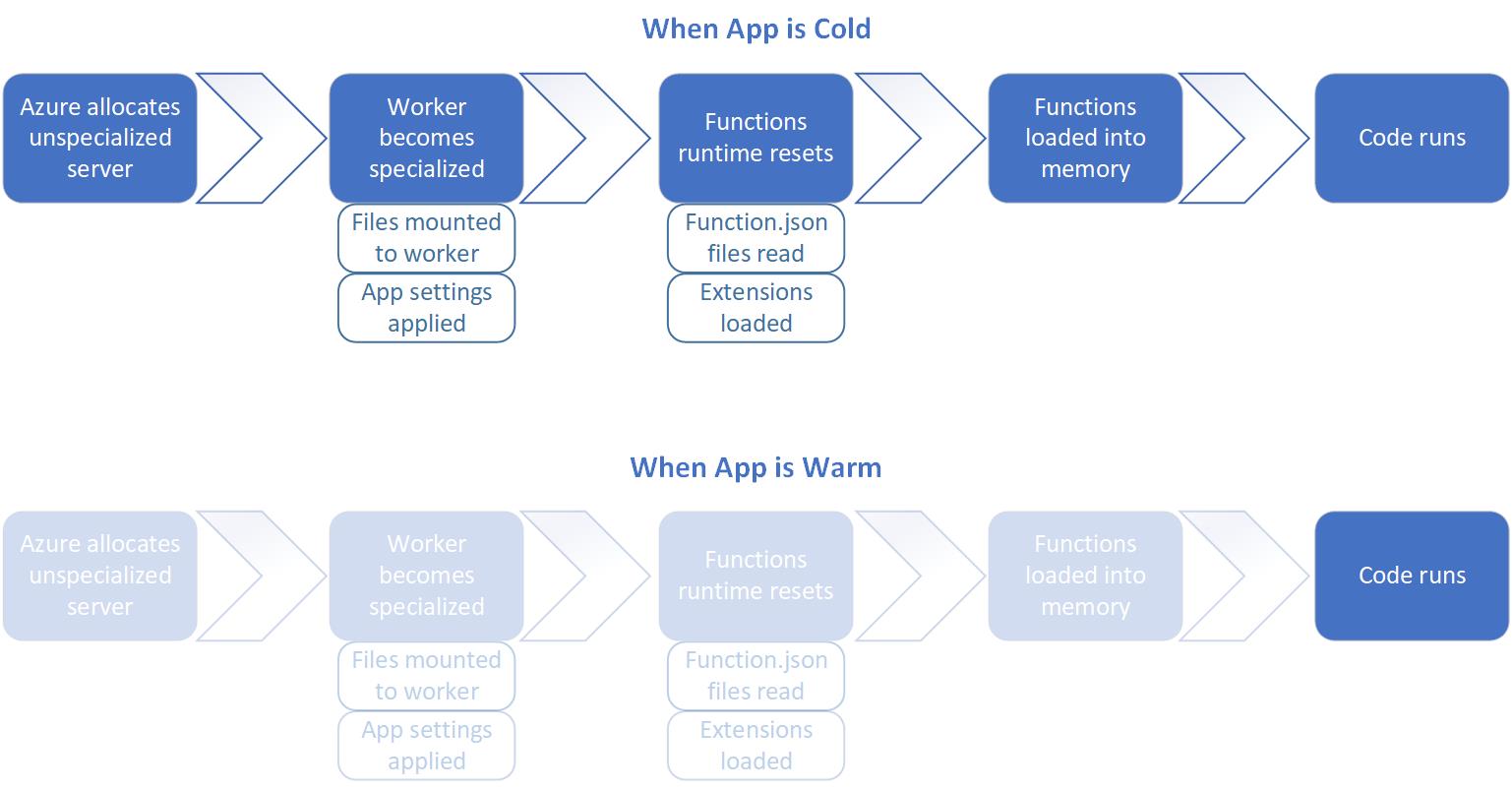
Serverless平台的自动配置和自动可扩展性是这些FaaS的重要功能：无需管理，云提供商将根据实际传入负载为用户提供相应的基础架构。这种动态配置的一个缺点就是冷启动，通常，一段时间未使用的应用程序需要更长的时间来启动和处理第一个请求。云提供商保留了许多通用的worker。每当无服务器应用程序需要扩展时，无论是0到1个实例，还是从N到N1个，运行时都会选择一个备用worker并配置其服务于指定的应用程序。此过程需要时间，因此应用程序事件处理的延迟会增加。图1描述了“冷启动”的一系列过程[3]：

图1 空闲worker冷启动的整个流程

## 3.2 冷启动测试

New Relic网站在冷启动方面进行了一些简单的研究，Mikhail Shilkov对于冷启动进行了一些测试并展示了他的实验结果。Liang Wang等人研究了实例放置的冷启动延迟。

观察冷启动时间相对简单——使用来自New Relic Infrastructure的AWS Lambda集成的数据，New Relic网站进行了一些测试，尽管冷启动时间大部分情况下，可以在 50ms 以内。而这是对于 Node.js 应用来说，对于拥有虚拟机的 Java 和 C# 差距可能达到十几倍。

Mikhail Shilkov 一开始使用了Node.js进行测试。Javascript可能是迄今为止无服务器应用程序最流行的语言。 因此，比较三个云提供商在Javascript中的表现方式是有意义的。 基础测试测量“Hello World”类型函数的冷启动。函数没有依赖关系，因此部署包非常小。在测试中，AWS显然表现最好。 GCP位居第二，Azure是最慢的。Mikhail Shilkov 进一步对不同编程语言间进行了比较，Azure支持更多语言，包括Python和Java，但它们的冷启动问题十分严重[4]。

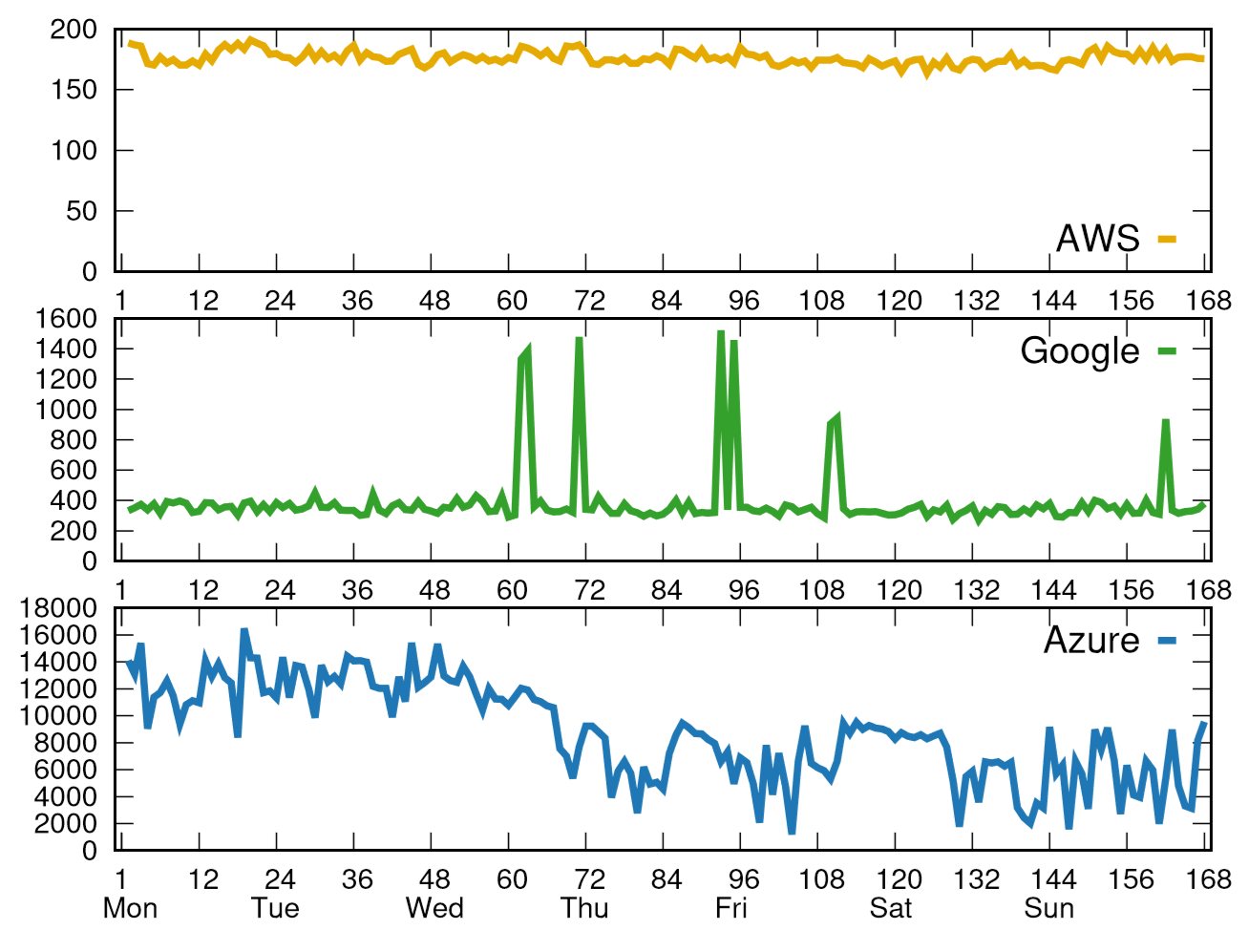
在2.1.2节介绍了Liang Wang等人对实例放置的冷启动延迟的研究。研究中发现，最小VM内核正常运行时间（来自/proc/uptime）为132秒，表明VM已在调用之前启动。因此，AWS拥有一个准备好的VM池。2.1.2节中情况（1）中的额外延迟更可能通过调度（例如，选择VM）而不是启动VM来引入。函数内存和语言影响冷启动延迟，比如Python 2.7实现了最低中位冷启动延迟（167-171 ms），而Java函数的延迟（824-974 ms）明显高于其他语言。图2展示了三种无服务平台在超过168小时内的冷启动延迟的情况。

图2 每个数据点是在给定时间内收集的所有冷启动延迟的中值

冷启动延迟通常随着函数内存的增加而降低。一种可能的解释是AWS会根据内存大小按比例分配CPU功率; 随着CPU功率的增加，环境设置变得更快。由于AWS的实例放置策略，可以在同一VM上同时启动许多函数实例。在这种情况下，随着更多实例同时启动，冷启动延迟会增加。 比如，在给定VM上启动具有128 MB内存的基于Python 2.7的函数的20个函数实例平均花费1,321 ms，这比在同一VM上启动1个函数实例（186 ms）慢约7倍。Azure和Google。Google也会按比例分配内存，但Google内存大小对冷启动延迟的影响要大于AWS。在Azure中启动函数实例需要更长的时间，尽管它们的实例总是分配1.5 GB的内存。Azure中的冷启动延迟中值为3,640毫秒。过长的延迟是由Azure发现并正在努力改进的平台的设计和工程问题造成的。

以上结果表明，当空闲Lambda收到请求时，必须加载代码，并且预期响应时间会增加。在某些情况下，这种差异可能是极端的：如果一个简单的lambda在20ms以内响应，它的闲置时间超过一个小时，则需要800ms才能响应。在Yan Cui的一次探索中，显示每次同时调用Lambda都有自己的冷启动开销。这意味着，对于不经常使用的函数而言，冷启动或许不是一个问题，但对于突然激增的函数，情况可能会变得更糟。使用无服务器的最常见原因是它可以更快地推出新函数，如果在很少使用服务或看到使用突然激增的情况下冷启动速度减慢，那是不是意味着会对性能造成无法预估的影响？

## 3.3 冷启动优化

既然已经通过测试对serverless性能有了了解，那么就可以开始解决如何构建解决方案以避免它的问题。本节从多个角度探讨了冷启动问题的优化手段。

### 3.3.1 代码优化

首先，尽量选择一些延迟较低的语言，比如通用的语言JavaScript。 上一节中几种实验结果表明，JavaScript在主流serverless平台中都表现较好，尽管理想情况是对运行的任何语言都处于较好的优化状态。希望这些语言在未来表现更好，但是现在，建议使用上面更加通用的语言。

其次，尽量编写一些轻量级代码，部署代码时，依赖项将作为文件添加到应用程序中。应用程序所需的所有代码最终都会加载到内存中，这对于更大的应用程序来说需要更长的时间。因此，如果有大量的依赖关系，将导致更长的冷启动，这是因为文件的I/O操作时间增加，以及将更大的应用程序加载到内存所需的时间更长。一些人使用JavaScript编写函数，这不仅会增加应用程序的大小，还会增加Azure文件必须处理的文件数量，从而导致进一步减速。

最后，编写一些高效代码。有时仅仅可以通过编写更有效的代码缩短冷启动时间。应尽可能多地尝试进行异步处理。如果有重量级同步调用阻止您的代码的实现，函数将无法正常运行。在此过程中，尽量减少代码启动前准备阶段的工作量，并避免使用占用太多CPU的代码[3]。

### 3.3.2 保持热启动

保持热启动似乎是冷启动优化的一个hack方案，但是它的确能在目前的冷启动下带来帮助。Yan Cui的实验表明，AWS Lambda可以将容器保持大约45分钟。 既然冷启动十分耗时，那么是否可以尽量少的让容器陷入冷启动？

WarmUP的口号是“Keep your lambdas warm during Winter”，它本质就是通过这样一个思路写得一个工具。

为了防止容器停机，以某种频率向函数发送虚拟请求，称为热身调用。当然，需要对Lambda进行必要的更改，以区分热身调用和实际调用。由于热身调用，serverless平台将保持容器warm，将永远不会遇到冷启动。但事情并非如此直截了当。

一个热身调用可以保持一个容器warm，但如果有更多的容器，如何保持容器warm？ 如果在生产中使用Lambda，则可能会有多个容器应该同时启动。需要并行进行热身调用以保持所需的容器数量。此时还有两个风险：（1）保持所有容器忙于热身调用，并且真正的调用找不到运行的地方。 这再次导致真正的呼叫冷启动；（2）一个容器中的Lambda函数可以一次捕获所有调用，这可能会在一段时间后使其他容器失效。

为了解决第一个问题，WarmUP中提出了一种方法。 如果想一直保持N个容器，WarmUP不会连续发送N个热身调用。相反，更频繁地发送N/2 个热身调用，并且在R小于N/2的情况下发送N个热身调用。例如; 如果想一次保留10个容器，可以每5分钟发送5到10个随机调用。通过这种方式，可以确保每30分钟阻止所有容器，同时保持Lambdas的整个时间。为了解决第二个问题，应该稍微等待Lambda（比如100ms，它可以在WarmUP配置），同时调度热身调用。这样，单个容器不会捕获所有请求，可以同时保持所需的容器数量。

一段时间后，应用程序上的负载会增加，这意味着需要保留更多容器以避免延迟。 在这种情况下，需要增加对Lambda容器的并行调用次数。 同样，如果流量遵循随时间减少的模式，则可能需要减少并行呼叫的数量。 在这种情况下，最好有一个比例因子来减少或增加容器的数量，比如加倍或削减一半。

Mikhail Shilkov 在他的论文[5]中也提到了在Azure Functions中通过始终保持warm来避免第一次冷启动之后的其他冷启动。Azure Functions在Azure App Service提供的实例上运行，并且每个实例都能同时处理多个请求，这些跟AWS Lambda不太一样，所以可以假设：（1）如果每20分钟发出一个请求，则第一个实例应该长时间保持warm；（2）除非现有实例太拥挤，否则同时启动不会导致冷启动；（3）当运行时间增加并且启动新的实例时，可以在后台进行此操作，仍将传入的请求转发到现有的warm实例中。一旦新实例准备就绪，就可以在不造成冷启动的情况下将请求添加到池中；（4）如果以上成立，则第一次冷启动执行之后会得到缓解。

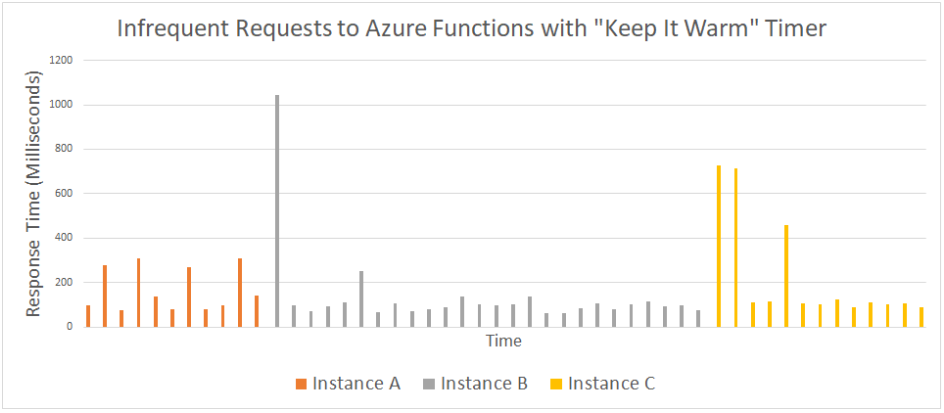
以下对一个包含两个函数（正在测试的HTTP函数、每十分钟运行一次并且只记录一行日志文本的计时器函数）的函数App进行了测试。在两天时间内通过向一个应用程序发出不常见的请求，大部分都会造成冷启动。并且，在测试过程中，即使有规律地触发计时器，Azure也有两次切换了实例为该应用服务。图3展示了测试的结果，虽然新的实例的第一个请求比后续请求慢，但是它们比正常的完全冷启动时间更快，这可能与HTTP堆栈加载有关。

图3 热启动机制下Azure Functions的测试结果

当有一个热实例，但是它已经慢于处理其他请求时，并行请求会被延迟还是会由同一个热实例处理？作者通过测试一个非常轻量级的函数（图4），测试向此函数发出2-10个并行请求，记录所有请求的响应时间，测试的结果在图5中展示。

public static async Task<HttpResponseMessage> Delay500([HttpTrigger] HttpRequestMessage req)

{

await Task.Delay(500);

return req.CreateResponse(HttpStatusCode.OK, "Done");

}

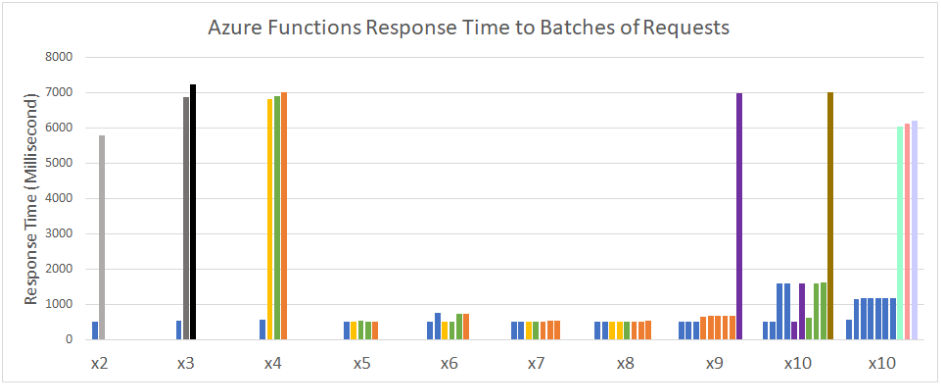
图4 并行请求测试函数

图5 并行请求测试函数的测试结果

图5中每组条形图代表同时发送的请求。然后在下一组请求发送之前大约有20秒的暂停；条形图的颜色依据处理该请求的实例，相同的颜色代表相同的实例。实验的一些观察结果：（1）在64个请求中，有11个冷启动；（2）同一实例可以处理多个同时发出的请求，例如，在最后一批中，一个实例处理了10个请求中的7个；（3）尽管如此，Azure仍然急于为多个请求启动新的实例。总共创建了12个实例，这甚至超过了每个批次中的最大请求数量；（4）其中一些实例实际上从未被重复使用（x2和x3批次中的灰色条形，x10中的棕色条形）；（5）对每个新实例的第一个请求需要支付整个的冷启动价格。在为接收的请求重用现有实例时，runtime不会在后台配置它们；（6）如果一个实例一次处理多个请求，则响应时间总是会受到影响，即使该函数是超轻量级的（Task.Delay）。

总的来说，对于具有请求频次小的低流量应用程序，可以设置每10分钟左右触发一次“warmer”的计时器函数，以防止唯一的实例被回收。然而，scale-out的冷启动是真实存在的，没有观察到任何阻止它们发生的方法。当多个请求同时进入时，我们可能会希望其中一些请求命中一个新实例并放慢速度。但是实例重用的确切算法并不全然清晰。同一个实例能够并行处理多个请求，因此在配置冷实例过程中，从routing到热实例方面存在优化的可能。

### 3.3.3 SOCK

其他优化冷启动的解决方案，比如Edward Oakes等人在论文[6]中提出了新的见解。3.1节提到过Serverless平台的自动配置和自动可扩展性是这些FaaS的重要功能，它允许程序自动向上或向下扩展处理程序的数量以适应负载，这样开发人员无需担心服务的机器数量和配置会满足不了工作负载，并且serverless平台通常非常便宜。为了提高开发人员的开发速度，三种策略（在更高的抽象级别编程、重用库、将应用程序分解为自动缩放的serverless lambda）应用而生，但是这些策略会使冷启动过程更加昂贵和频繁。诸如python和JavaScript等语言需要大量运行时间，使程序的启动速度比等效的C程序慢10倍，而重用代码会进一步增加从库加载和初始化时的启动延迟。Serverless扩大了这些成本，因为如果一个整体应用程序分解为n个serverless lambda，启动频率也同样扩大。从供应商的角度来看，避免冷启动开销非常大，如果冷启动速度更快，供应商将能够回收空闲资源，并更积极的重新平衡跨机器的负载，所以快速冷启动（fast cold start）就变得很重要。

Serverless平台通常使用容器隔离lambda，因此，容器初始化的优化是lambda冷启动问题的关键部分。在Linux中，容器化不是一个单一的内聚抽象，相反，Docker[7]等通用工具通常用于使用各种Linux机制构建容器，以分配存储、逻辑隔离资源和隔离性能。即使lambda在轻量级沙盒中执行，语言的runtimes和包依赖性也会使冷启动变慢。许多现代应用程序已经习惯于低延迟请求，例如，大多数Gmail远程过程调用都很短，在100毫秒内完成（包括互联网往返时延）。在短请求中，平均延迟为27毫秒，大约是启动Python解释器并打印“hello world”消息所需的时间。除非serveless平台提供语言和库特定的冷启动优化，否则将这些应用程序分解为独立缩放的lambda是不切实际的。

所以本文作者基于这些实际和研究，实现了SOCK（适用于serverless优化的容器），并与其他相似容器进行了比较。作者的设计目标：（1）对导入库的Python应用进行低延迟的调用；（2）高效的沙箱初始化，这样个人就可以实现高稳态的吞吐量。通过维护预初始化容器池来隐藏延迟的系统、作为大批处理的一部分并行创建多个容器的系统可以分别满足这两个目标但是不能全部满足，所以作者提出了三管齐下的解决方法：（1）为沙箱lanbda构建一个Lean容器系统；（2）推广Zygote provisioning技术扩展了大量不受信任的包的集合；（3）设计了一个三层缓存系统来减少包安装的花销。

**Lean Containers：**使用绑定安装来创建容器目录；使用Unix域的Socket来进行通信；创建cgroups池来用于隔离。

**Generalized Zygote：**分支启动技术，避免重复执行相同的初始化工作和使用多个相同副本消耗多余内存的子进程。

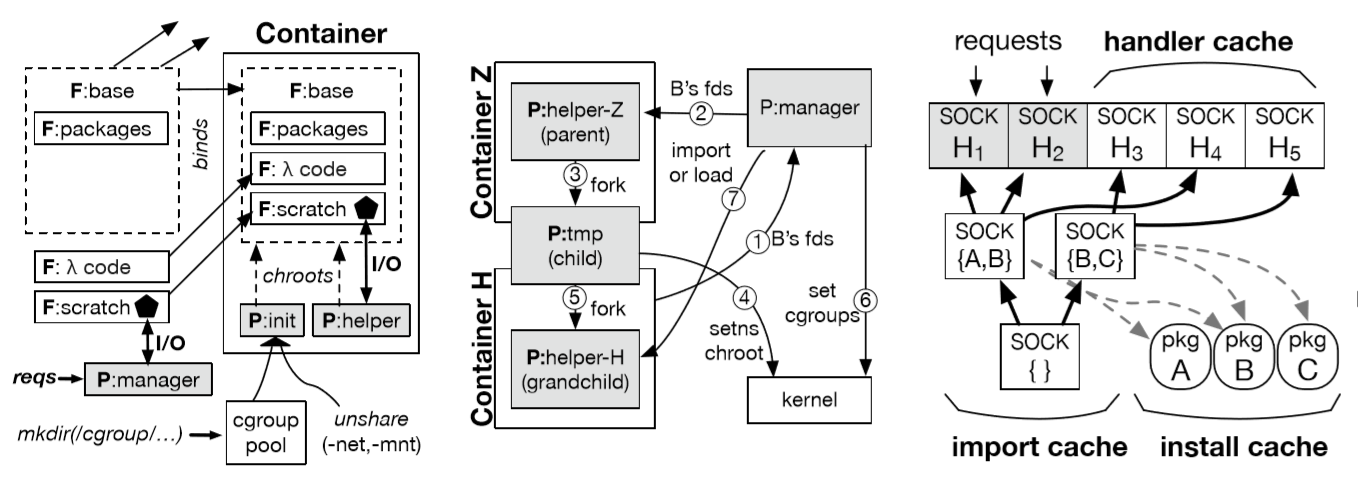
**Serverless Caching：**使用SOCK构建一个三层的换成系统；缓存导入尽可能不导入没有需求的包；监控导入缓存虽有条目的累计内存利用率，当利用率超过限制时，后台回收条目。

图6 Lean Containers、Generalized Zygote、Serverless Caching

经实验验证，基于SOCK的OpenLambda对比于基于Docker的OpenLambda有18x的速度提升，当SOCK配合了Zygotes时，这个数值增加到21x；对SOCK做的三个与包相关的优化，与没有进行优化的SOCK相比，优化后的效果至少提升了45x；预安装包有较高的使用率，但随着测试应用的数量增加，命中率下降；冷SOCK比AWS Lambda提供的快2.8x，比OpenWhisk快5.3x。

## 3.4 本章小结

Serverless在请求到来时才运行，应用不运行时会进入休眠状态，响应下次请求时会有一个启动时间，即冷启动。冷启动是serverless中一个非常重要且有必要解决的问题。从供应商的角度，优化解决冷启动问题可以有效节省serverless处理请求前准备的时间，增加供应商对资源的利用率，在以时间为计价单位的运营形式下，增加了供应商的收入；从租户的角度，租户无需为空闲的资源付费，但是减少冷启动的时间同样也能带来一定的经济节省，如第一章中表1所示，各serverless平台都是按时间收费的，更重要的是更快响应租户请求会带来更好的使用体验，从而留住租户。所以serverless平台也致力于优化解决冷启动问题，从2.1.2节的实验可以看出，Azure Functions的冷启动延迟明显高于AWS和Google，所以Azure意识到了这个问题并在努力改进，在AWS Lambda和Azure Functions的官网上都不难找到“实时文件处理”、“实时流式处理”的卖点。

本章总结了几种优化冷启动较为突出的解决方案，可以分为两种：减少冷启动时间、或者避免冷启动问题的产生。前面一种可以通过代码优化或者容器隔离来实现，减少冷启动的一部分启动延迟，但是代码优化受到开发人员的限制，容器隔离（本文中的SOCK容器）最小化了启动的成本，并且内置包的支持可以减少具有依赖关系的应用程序的冷启动延迟；保持热启动可以避免冷启动的发生，但是对serverless平台的性能配置有一定要求，所以冷启动仍然存在优化的空间。

# 第4章 短暂存储

本章在4.1节中，首先根据第二章的测试结果介绍了目前短暂存储存在的一些问题并给出了期望的短暂存储性质，并在4.2节中讨论了一些优化方案。

## 4.1 短暂存储的问题

2.3节中的测试结果表明，分布式键值存储在各种测试结果表现了良好的性能，但是用户必须负责管理存储群集规模和配置，其中包括选择要配置的适当计算，存储和网络资源。此外，更改实例类型或添加（删除）节点可能需要拆除和重新启动群集，节点需要几分钟才能启动，而服务将按小时计费。

不同存储技术（例如，DRAM、NVM、闪存和HDD）的可用性增加了为性能和成本找到最佳集群配置的复杂性。 但是，存储技术的选择至关重要，因为作业可能会表现出不同的存储延迟，带宽和容量要求，而不同的存储技术在性能特征和成本方面差别很大。每种资源配置都会导致不同的性能和成本，为作业寻找帕累托有效的存储分配非常重要，并且在多个作业中变得更加复杂。因此，一个理想的短暂存储应该具有如下性质：

**针对各种对象大小的高性能：**serverless 分析应用程序在存储，分发和处理数据方面存在很大差异。这种多样性反映在作业期间生成的短暂数据的粒度中。读/写大对象的应用程序需要高吞吐量（例如，使用500 lambdas排序100 GB需要高达7.5 GB / s的临时存储吞吐量），而低延迟对于小对象访问很重要。因此，短暂的数据存储必须为整个对象大小范围提供高带宽，低延迟和高IOPS。

**自动和细粒度扩展：**serverless 计算的关键承诺之一是动态满足应用程序需求的灵活性。serverless 框架可以即时启动数千个短期任务。因此，用于serverless 应用程序的短暂数据存储可以在几分之一秒内观察到I/O请求的风暴。一旦负载消散，存储资源应按比例缩小以提高成本效率。此外，serverless平台的用户需要一种存储服务，该服务自动管理资源并仅为其作业实际消耗的细粒度资源向用户收费，以便匹配serverless 计算已经为计算和内存资源提供的抽象。自动资源管理很重要，因为让用户去配置群集的性能与成本权衡是一种负担找到帕累托最优点是不容易的；在使用任何低成本资源分配导致次优执行时间的同时，添加资源仅增加成本而不增加执行时间是一个重点。总之，临时数据存储必须自动调整资源分配的规模，以满足应用程序I/O要求，同时最大限度地降低成本。

**存储技术意识：**除了对群集资源进行权限化之外，存储系统还需要决定使用哪种存储技术来处理哪些数据。云中可用的各种存储介质允许不同的性能成本权衡。作业的最佳存储介质选择取决于其特性。 因此，临时数据存储必须将应用程序数据放在正确的存储技术层上，以提高性能和成本效率。

**容错：**短暂数据仅在作业执行期间有价值，并且可以轻松地重新生成。因此，短暂的存储解决方案不必像传统存储系统那样提供高容错性。

## 4.2解决方案

本节根据期望的描述结果给出了两个目前现有的解决方案：Pocket（4.2.1节），一种用于临时数据的弹性分布式存储服务，可自动动态地对存储群集资源分配进行权限化，从而在降低成本的，同时提供高I/O性能以及Selecta（4.2.2节），一个为数据分析工作负载推荐近乎最优的云计算和存储资源配置的工具。Pocket侧重于构造一个新的存储系统交给云服务商去应用，而Selecta则是一个基于现有的存储系统的分析工具，分析用户的任务并自动给出一些存储选择指南。

### 4.2.1 Pocket

Pocket[8]是一个分布式数据存储系统，设计用于在无服务器分析中高效地共享数据。Pocket为任意大小的数据集、自动资源扩展和跨多个存储层（如：DRAM、Flash和Disk）的只能数据放置提供了低延迟和高吞吐量。Pocket的独特属性来自于跨三个层面的严格职责分离：一个控制平面，用于确定作业的数据放置策略；一个元数据层面，用于管理分布式数据放置；数据层面负责存储数据。Pocket根据当前的负载，在惊喜的资源和时间粒度上独立的伸缩这三个方面。Pocket使用了启发式的方式，根据作业的特性来分配合适的存储介质、容量、带宽和CPU资源，从而在降低成本的同时提供高I/O性能。

Pocket采用了以下设计原则：

**职责分离：**Pocket在三个不同的平面上划分责任：控制平面，元数据平面和数据平面。 控制平面管理群集大小调整和数据放置。元数据平面跟踪存储在数据平面中的节点上的数据。 这三个平面可以根据负载的变化独立调整。

**亚秒级响应时间：**所有I/O操作都是简单的，目标是亚毫秒级别。 Pocket的存储服务器针对快速I/O进行了优化，仅负责存储数据（而非元数据），使其易于扩展或缩小。 控制器以第二粒度缩放资源，并通过智能地控制传入的作业数据来平衡负载。 这使Pocket具有弹性。

**多层存储：**Pocket利用不同的存储介质（DRAM，闪存，磁盘）将作业数据存储在满足应用程序I/O需求的层中，同时最大限度地降低成本。

**设计架构：**如图7所示，该系统由逻辑集中控制器，一个或多个元数据服务器和多个数据平面存储服务器组成存储服务器上的彩色条显示集群中所有作业的已使用和已分配资源。

控制器为作业分配存储资源，并随着作业数量及其要求随时间变化而上下动态扩展Pocket元数据和存储节点。 控制器还为作业（即，用于作业数据的节点和存储介质）做出数据放置决定。

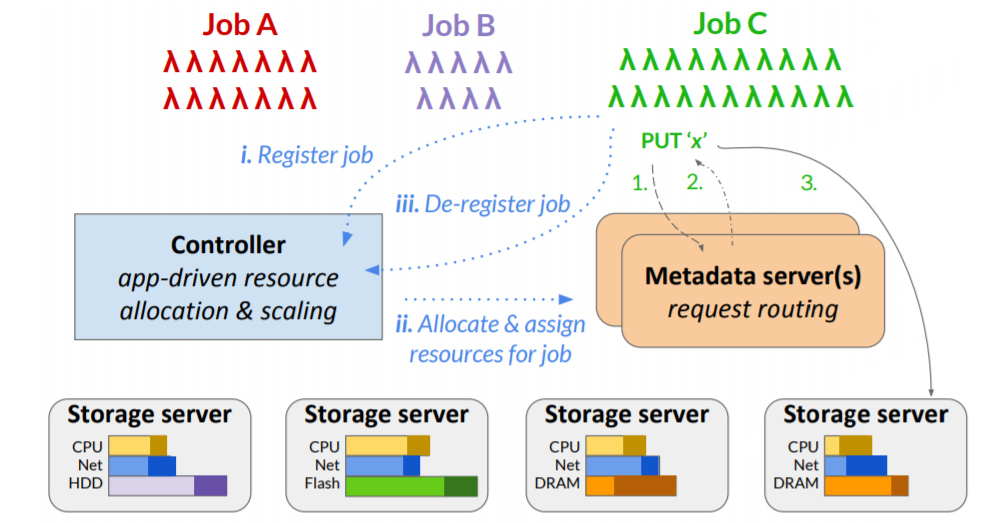
元数据服务器通过将客户端请求引导到适当的存储服务器来实施由控制器生成的粗粒度数据放置策略。 Pocket的元数据平面以块的粒度管理数据，块的大小是可配置的。 在部署中使用64 KB的块大小。 大于块大小的对象被分成块并分布在存储服务器上，使Pocket能够支持任意对象大小。 客户端访问配备有不同存储介质（DRAM，闪存和（或）HDD）的性能优化的存储服务器上的数据块。

图7 架构设计

一个关于Pocket的重要内容就是Pocket是如何适当调整资源分配的规模。当作业注册时，Pocket的控制器利用通过API传递的可选提示来保守估计作业的延迟，吞吐量和容量要求，并找到经济高效的资源分配。除了预先为作业分配资源外，Pocket还会持续监控集群的整体利用率，并根据负载决定何时以及如何扩展存储和元数据节点。

当作业注册时，控制器首先确定三个维度的资源分配：吞吐量，容量和存储介质的选择。 然后，控制器使用在线bin-packing算法将资源分配转换为节点上的资源分配Pocket使用适用于通过注册作业API传递的可选提示的启发式方法。

如果没有关于作业的提示，Pocket使用默认的资源分配，保守地过度配置资源以高成本实现高性能。 默认情况下，Pocket保守地假设作业对延迟敏感。因此，Pocket会在溢出到其他存储层之前填充作业的DRAM资源，以便保证存储延迟。如果作业暗示它对延迟不敏感，则控制器不会为作业分配DRAM，而是使用最具成本效益的存储技术来控制控制器估算作业需求的吞吐量和容量。知道作业的最大并发lambda数N，允许Pocket计算对作业吞吐量要求的不太保守的估计。 如果提供此提示，则Pocket分配的吞吐量等于每个λ的峰值网络带宽限制的N倍（例如，AWS上每个lambda约600 Mb / s）。Pocket的API还接受有关作业的总吞吐量和容量要求的提示，此信息可以来自分析。 当Pocket仅收到吞吐量提示时，控制器会分配与作业吞吐量分配成比例的容量。该比例由所使用的VM上的存储吞吐量与容量比设置。反之亦然，如果仅提供容量提示，Pocket将根据VM容量与吞吐量的比率分配吞吐量。

通过以上关键技术的实现，Pocket实现了与ElastiCache Redis类似的无服务器分析应用程序性能，同时将成本降低了近60％。

### 4.2.2 Selecta

Selecta[9]是一个为数据分析工作负载推荐近乎最优的云计算和存储资源配置的工具。Selecta基于稀疏数据，使用潜在因素协同过滤来预测在不同配置下应用的表现。

它的具体应用是在为指定的应用程序选择存储配置。用户指定的性能成本下的应用程序分析。传统配置方案数据分析作业的输入输出文件传统上存储在分布式文件系统中，如：HDFS或对象存储系统，如：Amazon S3。中间数据对每个节点通常是用专用本地块存储卷，并且r-HDD作为扩展的存储空间。在典型的In typical Spark-as-a- service云部署中，每个实例默认提供两个远程存储卷：一个用于实例的根、另一个用于日志。

现有的方案有自动选择云中最佳的VM配置。然而，这些工具往往忽略了云存储配置方案的异质性，充其量只能区分“快”和慢。

Selecta采用了推荐系统中常用的机器学习技术——隐性因子协同过滤。Selecta使用稀疏的性能数据来训练不同云配置下的应用程序性能数据。以及目标应用在两种配置下的性能对比。Selecta利用稀疏训练数据来学习，比穷举搜索更快、更经济有效。

如图8所示，Selecta的输入包括：（1）一组训练应用程序在集中配置上的执行时间；（2）目标应用程序在两种参考配置上的执行时间；（3）目标应用程序的性能指标成本范围。配置由节点（VM 实例）数量、每个节点的CPU内核和内存以及用于输入输出数据和中间数据的存储截止类型和容量决定。

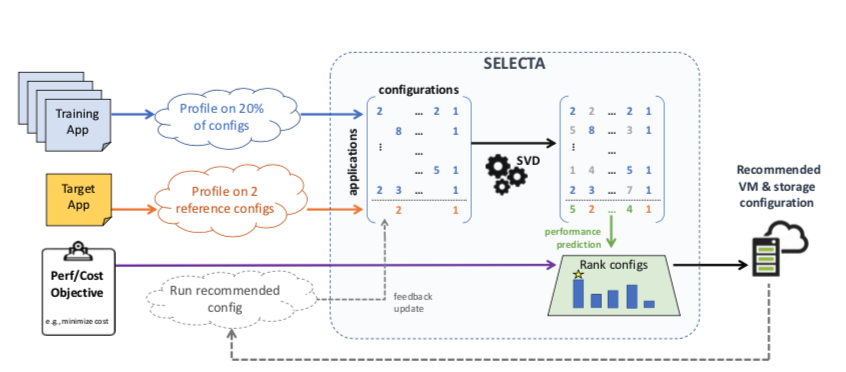
Selecta使用潜在因素协同过滤来预测目标应用在其余没有给出的候选配置参数上的性能表现。通过这些性能预测结果、各种VM实例和存储存储选项的单位时间成本，Selecta可以根据用户的目标成本推荐合适的配置。例如：Selecta可以推荐执行时间最小化、成本最小化或在给定预算中执行时间最小化的配置方案。在持续的使用过程中，Selecta经常或定期重新运行新数据的分析作业，以确保当前的应用处在最优配置下。

图8 Selecta实现方案

1. 预测过程：

**预测方法：**选择协作过滤，因为它与所使用的数据分析框架（例如Spark vs. Storm）的细节无关，同时可以利用跨应用程序和配置收集的稀疏训练数据。相对于CherryPick和Ernest仅基于目标应用程序的培训数据构建性能模型，Selecta的目标是利用多个应用程序提供的训练数据，两次运行目标应用来获得概要分析，在根据分析结果聚合为合理的推荐。

**潜在因素协同过滤：**Selecta的协同过滤模型将应用程序和配置转换为潜在因素集合。这个集合以潜在特征的形式描述目标应用和配置。而这些特征又是从用于训练的应用的运行情况分析中自动推断出来的。然后使用称为奇异值分解（SVD）的矩阵分解技术构建潜在因素模型。尽管该模型没有说明潜在特征在物理上代表什么，有一个假设说法是随机的I/O吞吐量。

运行SVD的一个要点是输入矩阵是稀疏的，因此对目标应用的性能度量只能建立在具体的配置之上。尤其在于矩阵中，代表目标应用的一行中，只有两个位置是用于表示确定的潜在参数的。需要抢其他缺失的项补充完全以预测配置中需要的参数。目前的有人提出的想法是通过迭代分解和矩阵的更新，来最小化已知项的错误，这样还可以用相对准确的预测更新未知项。

1. 具体操作

**传入目标应用：**第一次将应用传递给Selecta时，利用预设的参考配置对其进行解析，这两个参考配置最好再计算量和存储资源属性上相差很大。Selecta要求对所有应用程序的参考配置中的参数保持一一对应，因此在传入SVD之前性能指标可以被定量的估计。

**给定性能指标和成本预算：**在用所有可能的配置对目标应用测试完成之后，Selecta根据用户设定的目标给出推荐配置方案，比如：为了最小化运行市场，把运行时间设置为最小（运行时长\*耗费时长/hour）。在选择存储技术（例如：SSD vs. NVMe闪存）时，Selecta还必须考虑应用程序的存储容量需求。Selecta利用Spark监控日志中可以利用的运行分析统计信息来确定中间数据和输入/输出数据容量。

**自适应：**重复出现的作业以及其输入的数据集可能会发生变化。Selecta依赖于来自初始应用程序分析和后续执行操作的CPU利用信息来检测可能影响最佳配置选择的特征的变化，当应用程序首次引入系统时，Selecta分配一个唯一的ID来存储应用程序相对应的信息，如：iowait的CPU利用率。每当重新执行应用程序时，Selecta将当前的iowait时间与存储的配置进行比较。根据iowait时间的不同，Selecta要么根据可用的度量值得出一个精准的预测，要么将目标应用视为新的传入应用，重新分析。

1. 评价

Selecta的协作过滤方法与应用程序和配置的选择无关。

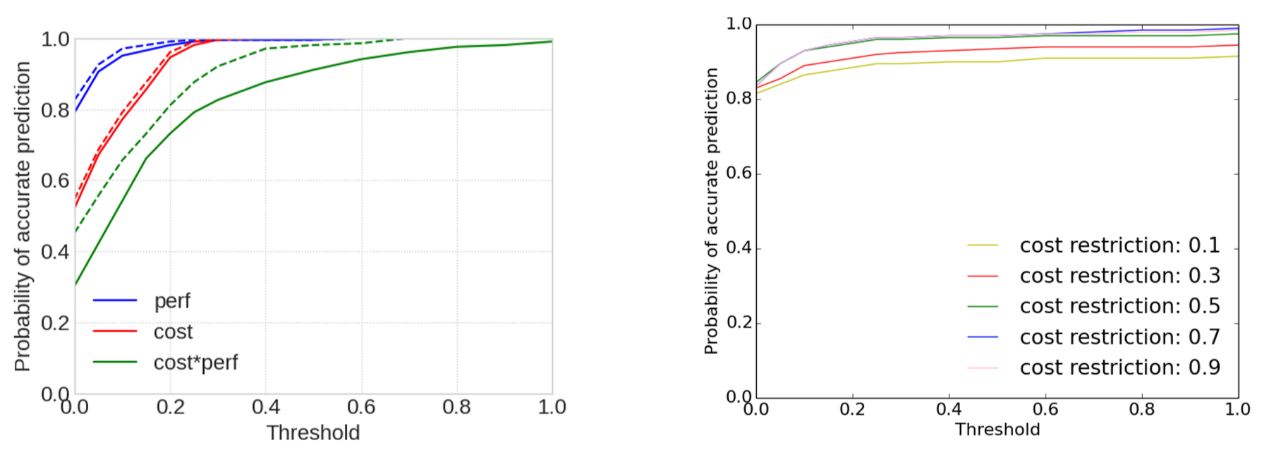
经过试验验证，对于多个应用程序，Selecta预测性能均方根误差为36%。图9中表示的是性能、成本和性能\*成本的准确率。图中显示了接近最优推荐的概率，它是一个阈值T的函数，阈值T定义给出最优推荐的百分比。在搜索性能最佳的配置是，Selecta有94%的可能性会在最优配置的10%的波动范围内推荐配置。图9的虚线显示了一次预测反馈后的精度提高。在反馈后，成本\*性能目标中，接近最优推荐的概率显著提升，从52%增加到65%。其中T=10%。

图9 性能、成本和性能\*成本的准确率 图10 准确推荐概率

图10显示了比东安“在给定确定成本的情况下，选择性能最好的配置”目标的准确推荐概率。假设T=10%，Selecta对图10中成本限制的准确率在83%到94%之间。

另外，在Selecta中，用户希望通过在所有候选配置中配置一组有限的应用程序来启动系统，这会造成一种冷启动的情况。对于每个目标应用程序测试回合，作者取203个已有的训练应用程序，然后随机删除一部分行（训练应用程序），确保删除与不同数据集比例因子运行的目标应用程序相对应的行，以确保Selecta的准确性不依赖于与目标应用程序直接相关的训练应用程序。在冷启动时，用户应提供比候选配置数量多2.5倍的训练应用程序，以达到理想的精度。研究显示，对于冷启动，用户可以根据40%到60%的候选配置来确定初始培训应用程序。随着Selecta继续运行并积累更多的训练应用程序，用户需要使训练应用程序提供的配置百分比下降到20-30%。

## 4.4本章小结

本章的内容就Serverless的短暂存储进行了讨论，对于不同存储介质，文件传输速度、存储空间、价格等都需要考虑在内，对于服务商和用户都有追求效益的需求，这也是Serverless框架的重要设计要旨。

文中列举了两个方案。分别是有服务商提供的，Pocket中自动进行存储方案配置以及帮助用户自主选择配置方案的SOCK工具。

Pocket通过三大设计原则：职责分离、亚秒级相应以及多层存储的原则，在内部构建了一个可以根据负载变化独立调整、智能控制传入作业数据来平衡负载以及降低维护成本的内部短暂存储机制。

Pocket最大的问题在于用户的粘合度，当用户想要转换服务商，而目标服务商并没有使用Pocket系统时，对用户就很不便捷。这时，若用户需要优化自己的API服务，就得自行进行存储参数的配置。而Selecta就很好地解决了这一痛点。

Selecta只需用户传入目标应用、给定性能指标和成本预算以及其他的系统需求，Selecta基本上就可以给出一个符合当前应用的最佳配置方案。而且当应用出现变动，Selecta也可以自行进行调整。

总之，若服务商提供了Pocket方案，则可以减轻用户的配置负担；当服务商没有内置Pocket，同样可以用Selecta作为一个替代方案。

# 第5章 思考与总结

无服务器计算是一个诱人却非万能的新兴技术，应当理性看待无服务器的落地。Serverless 并非银弹，它只是云计算在逐渐成熟之后的一个细分技术。根据本文讨论的测试结果可以看出，目前的无服务器商依旧存在许多亟待解决的问题，目前无服务器的落地更多的是套用以前存在的一些技术，它更像一个以一种无服务器的方式创建基于http的前端微服务，进行FaaS函数的扩展、管理和其它功能。本文的测试结果表明无服务器在一些关键性能和短暂存储方面都有很大的优化空间。

本文重点讨论了冷启动和短暂存储的优化。在冷启动优化方面，本文介绍了三种优化策略，一种是纯粹的代码优化，通过尽量精简代码依赖项减少冷启动时间，一种是在使用过程中采用类似心跳机制的策略多次激活函数实例，保持函数实例的温暖，SOCK则从容器的角度完全重建了无服务器依赖的容器，大幅度减少了冷启动时间。显然，第一种和第二种方法尽管能给目前冷启动问题带来一些缓冲，但如果想要从根本上解决冷启动问题，只有重新设计一种新的容器。SOCK在很大程度上解决了这个问题，SOCK对于容器底层的技术实现进行了大量的重新设计，大幅度降低了构建时间，对于后续的研究发展有很大空间。**可惜的是**目前SOCK只针对了对于Python的优化，局限性太大，并且它对于python模块的优化会随着时间的推进优化效果越来越差。由于不同语言的特性，不同语言的冷启动时间自然不同，因此SOCK同样可以启发后续为不同的语言构建专有的容器，比如Java。Java由于存在自己的虚拟机JVM，其冷启动问题一定显著存在，值得注意的是，由于Java 9支持Oracle JDK的良好功能，如AOT（提前编译），这意味serverless提供商或许能够利用它以及其他创造性的解决方案来减少Java的冷启动时间。

在短暂存储优化方面，本文介绍了两种优化，一种是从系统角度，重新设计了短暂存储系统，系统会分析每个任务，从而判定该任务使用哪一种类型的存储，尽可能的主动为用户节省成本；一种是用户传入目标应用、给定性能指标和成本预算以及其他的系统需求，Selecta基本上就可以给出一个符合当前应用的最佳配置方案。Selecta是一个非常好的配置选择工具，但是不可避免的，用户依旧需要的额外的工具学习曲线，一个理想的无服务器平台应该避免这部分用户的操作，尽可能减少用户的学习成本。Pocket作为一个远程自动分类存储系统，它将指责分离，计算节点直接访问存储服务器以进行数据操作，并与控制平面的元数据服务器通信，但是一方面将职责分为三个不同的平面会带来额外的性能开销，同时，与元数据服务器的通信可能会成为性能关键路径。Pocket由于其设计的元数据容错机制，导致当元数据服务器出错时，需要花费较长的时间由于恢复，或许使用主从复制策略可以有效减少这一方面的耗时。并且，Pocket要求元数据服务器缓存元数据，这意味着当元数据缓存变小时，Pocket的性也会下降。但是总得来说，Pocket先进的职责分离思想使得其设计在CPU利用率，可扩展性，资源分配合理性以及写入读取性能等方面都比现有的远程存储系统或者是单协调器的远程存储系统要高明许多。在未来的无服务器发展中，类似Pocket的自动短暂存储选择系统应当逐渐替代目前市场上纯用户选择的存储系统，进一步提高无服务器平台的自动化。

# 参考文献

1. Liang Wang, UW-Madison; Mengyuan Li and Yinqian Zhang, The Ohio State University; Thomas Ristenpart, Cornell Tech; Michael Swift, UW-Madison, Peeking Behind the Curtains of Serverless Platforms. 2018 USENIX Annual Technical Conference. July 2018.
2. Phodal, Serverless应用开发指南(<https://serverless.ink>).
3. Colby Tresness, Understanding Serverless Cold Start. February 2018.
4. Mikhail Shilkov, Serverless: Cold Start War. August 2018.
5. Mikhail Shilkov, Cold Starts Beyond First Request in Azure Functions. May 2018.
6. Edward Oakes, Leon Yang, Dennis Zhou, and Kevin Houck, University of Wisconsin-Madison; Tyler Harter, Microsoft, GSL; Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, University of Wisconsin-Madison. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. 2018 USENIX Annual Technical Conference. July 2018.
7. Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal, Issue 239, March 2014.
8. Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Christos Kozyrakis, Stanford University, IBM Research. Pocket: Elastic Ephemeral Storage for Serverless Analytics. October 2018.
9. Ana Klimovic, Stanford University; Heiner Litz, UC Santa Cruz; Christos Kozyrakis, Stanford University. Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics. July 2018.