Slacker：通过惰性 Docker 容器来进行快速分发

学号：2017282110277
姓名：周明浩

Slacker：通过惰性 Docker 容器来进行快速分发

摘要

容器化的应用程序越来越流行，但当前遇到的问题是，容器化应用程序的部署非常缓慢。我们开发了一个新的容器评价基准 HelloBench 来评估 57 个不同的容器化应用程序的启动时间。我们使用 HelloBench 详细分析了其工作负载，研究了它们启动时的 I/O 过程以及镜像的可压缩性。通过我们的分析发现，包的 pull 过程占启动时间的 76%，但其实只读取了 6.4%的数据。我们使用这些发现来指导 Slacker 的设计，Slacker 是一个为容器快速启动而优化的 Docker 存储驱动程序。使用者通过后端克隆快速提供容器的存储，并通过延迟获取容器的数据来减少启动时间。Slacker 使容器化应用的开发速度提高了 20 倍、部署速度提高了 5 倍。

1 引言

隔离是云计算或其他的一些多租户平台中非常理想的特性。没有隔离，用户将会面临不可预知的问题，例如：性能不足、崩溃以及各种各样的隐私问题。

通常来说，虚拟机监视器（VMM）用来为应用程序提供隔离。每个应用程序都部署在自己的虚拟机中，拥有自己的环境和资源。但不幸的是，管理程序需要进行各种特权操作，并使用各种技术来推断资源的使用。这导致的结果是管理程序启动速度慢，运行的时间开销很大。

由于 Docker 技术的普及，容器最近成为替代基于虚拟机管理程序的轻量级虚拟化技术。在一个容器中，所有的进程资源都被操作系统虚拟化，包括网络端口以及文件系统挂载点。容器本质上就是使所有资源虚拟化的过程，而不仅仅是 CPU 和内存。因此，容器启动不会比正常启动时慢。

再次遇到的不幸是，由于文件系统的瓶颈，启动容器在实际过程中要慢得多。尽管网络、计算和内存资源的初始化相对快速和简单，但是容器化的应用程序需要一个完全初始化的文件系统，包含应用程序二进制文件，完整的 Linux 发行版以及相关的软件包。在 Docker 或 Google Borg 集群中部署容器通常涉及大量的复制和安装开销。 Google Borg 最近的一项研究显示：“任务启动延迟是不稳定的，通常约为 25 秒。软件包安装占其中 80%左右的时间，已知的瓶颈之一是争用本地磁盘。

如果启动时间可以得到改善，则将出现一些机会：应用程序可以立即进行扩展以处理闪存拥堵事件，集群调度程序可以频繁地以低成本重新平衡节点，开发人员可以交互式地构建和测试分布式应用程序。

我们采取双管齐下的方法解决容器化应用的启动问题。首先，我们开发了一个新的开源 Docker 基准 HelloBench。HelloBench 基于 57 个不同的容器工作负载，测量从部署开始到容器准备好并开始执行工作的时间。我们使用 HelloBench 和静态分析来表现 Docker 镜像和 I/O 模式。我们的分析表明：（1）复制包数据占容器化应用启动时间的 76%，（2）容器化应用开始有用的工作实际

只需要有6.4%的复制数据，(3)在压缩率上，跨镜像删除重复数据比镜像的 gzip 压缩更好。

其次，我们使用我们的研究结果来构建Slacker，一种新的Docker存储驱动程序，通过在堆栈的多个层次上利用专门的存储系统支持来实现容器的快速分发。具体来说，Slacker 使用后端存储服务器的快照和克隆功能来显着降低常见 Docker 操作的成本。Slacker并没有预先传播整个容器镜像，而是懒惰地拖拽镜像数据，大大减少了网络I/O。Slacker还利用我们对Linux内核所做的修改来改善缓存共享。

使用这些技术的结果是普通 Docker 操作的性能大幅提高；镜像的上传速度提高153 倍，下载速度提高 72 倍。Slacker 使容器化应用的开发周期提高了 20 倍、部署周期提高了 5 倍。

我们还构建了一个新的基于容器的构建工具MultiMake，展示了Slacker快速启动的好处。MultiMake使用不同的容器化GCC版本，从相同的源代码生成 16 个不同的二进制文件。通过 Slacker，MultiMake，达到了10倍的加速。

本文的其余部分安排如下。首先，我们描述现有的Docker框架。接下来，我们介绍HelloBench，我们用它来分析Docker工作负载特性。我们使用这些发现来指导我们的Slacker的设计。最后，我们评估Slacker，提出MultiMake。

2 Docker背景

现在我们来介绍一下 Docker 框架，存储以及存储驱动程序。

2.1 容器的版本控制

虽然Linux一直使用虚拟化来隔离内存，但cgroups（Linux的容器实现）通过提供六个新的命名空间来虚拟化更广泛的资源：文件系统挂载点，IPC 队列，网络，主机名，进程ID和用户ID。 Linux cgroups于2007年首次发布，但最近才被广泛使用，与Docker（2013年发布）等新的容器管理工具的可用性一致。使用Docker，像“docker run -it ubuntu bash”这样的单一命令将从互联网上获取Ubuntu软件包，使用全新的Ubuntu安装初始化一个文件系统，执行必要的cgroup设置，并在环境中返回一个交互式的bash会话。

这个示例命令有几个部分。首先，“ubuntu”是镜像的名称。镜像是文件系统数据的只读副本，通常包含应用程序二进制文件，Linux 发行版以及应用程序所需的其他程序包。在 Docker 镜像中捆绑应用程序非常方便，因为分发者可以选择一组特定的包（及其版本），这些包将在应用程序运行的任何地方使用。其次，“run”是运行镜像的操作；运行操作基于用于新容器的镜像创建初始化的根文件系统。其他操作包括“push”（用于发布新镜像）和“pull”（用于从中央位置获取发布的镜像）;如果用户试图运行非本地镜像，则镜像被自动拉出。第三，“bash”是在容器内启动的程序；用户可以在给定的镜像中指定任何可执行文件。

Docker 管理镜像数据的方式与传统版本控制系统管理代码的方式大致相同。这个模型适用于两个原因。首先，可能有相同镜像的不同分支（例如，“ubuntu：latest”或“ubuntu：12.04”）。其次，镜像自然而然地建立在彼此之上。例如，Ruby-on-Rails 镜像建立在 Rails 镜像上，Rails 镜像又建立在 Debian 镜像上。这些镜像中的每一个都表示对之前提交的新提交;可能会有其他提交未被标记为可运行镜像。当容器执行时，它从一个提交的镜像开始，但文件可能被修改;在版本控制的说法中，这些修改被称为未分离的更改。Docker“commit”操作将一个容器及其修改变成一个新的只读镜像。在 Docker 中，一个图层引用一个提交的数据或者一个容器的未分离变化。

运行 Docker 的机器上运行一个本地的 Docker 守护进程。通过向本地守护进程发送命令，可以在特定的机器上创建新的容器和镜像。镜像共享是通过集中注册表完成的，这些注册表通常在 Docker 工作人员所在的机器上运行。镜像可以通过从守护进程到注册表的推送发布，并且可以通过对群集中的多个守护进程执行拉取来部署镜像。只有接收端不可用的层才能被传送。图层通过网络和注册表机器表示为 gzip compressed tar 文件。

2.2 存储驱动程序接口

Docker 容器以两种方式访问存储。第一种，用户可以在容器的主机上挂载目录。例如，运行容器化的编译器，可以将需要编译的源代码目录挂载到容器中，以便编译器可以读取代码文件并在主机目录中生成二进制文件。第二种，容器访问表示应用程序二进制文件和库的 Docker 图层。Docker 通过容器，进而通过挂载点来用作其根文件系统，用作此应用程序数据的视图。容器存储和安装由 Docker 存储驱动程序管理；不同的驱动程序可能选择不同的方式表示图层数据。表 1 中显示了驱动程序必须实现的方法（一些不重要的函数和参数未展示）。所有的函数都采用一个字符串“id”参数来标识正在被操作的图层。

Get 函数请求驱动程序安装图层并返回到安装点的路径。返回的挂载点不仅包含“id”层，而且还包含其所有祖先的视图（例如，在挂载点的目录结构中应该可以看到“id”层的父层中的文件）。把卸载作为单独一层。从父图层创建副本以创建新图层。如果父项为 NULL，则新图层应为空。Docker 调用 Create 函数有两个用途：（1）为新容器提供文件系统，（2）分配图层以存储来自 pull 的数据。

在 Docker push 和 pull 操作中分别使用 Diff 和 ApplyDiff，如图 1 所示。当 Docker 推送图层时，Diff 将图层从本地表示转换为包含图层文件的压缩过的 tar 文件。ApplyDiff 的做法相反：给定一个 tar 文件和一个本地层，它解压 tar 文件在现有的层。

图 2 显示了当第一次运行四层镜像时所进行的驱动程序调用。在镜像 pull 过程中创建四个图层;还有两个是为容器本身创建的。图层 A-D 代表镜像。 A 的创建需要一个 NULL 父项，所以 A 最初是空的。然后，后续的 ApplyDiff 调用告诉驱动程序将文件从拉出的 tar 中添加到 A.层 BD 分别填充两个

Method	Description
Get(id)=dir	mount "id" layer file system, return mount point
Put(id)	unmount "id" layer file system
Create(parent, id)	logically copy "parent" layer to "id" layer
Diff(parent, id)=tar	return compressed tar of changes in "id" layer
ApplyDiff(id, tar)	apply changes in tar to "id" layer

Table 1: **Docker Driver API.**

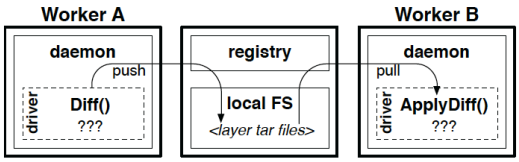

Figure 1: **Diff and ApplyDiff.** *Worker A is using `Diff` to package local layers as compressed tars for a push. B is using `ApplyDiff` to convert the tars back to the local format. Local representation varies depending on the driver, as indicated by the question marks.*

步骤：从父项复制，并添加 tar 文件。在步骤 8 之后，pull 完成，Docker 准备创建一个容器。它首先创建一个只读图层 E-init，向其中添加一些小的初始化文件，然后创建 E，容器将使用的文件系统作为其根。

2.3 AUFS 驱动程序的实现

AUFS 存储驱动程序是 Docker 发行版的通用默认值。该驱动程序基于 AUFS 文件系统（Another Union File System）。联合文件系统不直接将数据存储在磁盘上，而是使用其他文件系统（例如 ext4）作为基础存储。

联合挂载点提供底层文件系统中多个目录的视图。AUFS 在底层文件系统中安装有目录路径列表。在路径解析期间，AUFS 遍历目录列表；将选择包含正在解析的路径的第一个目录，并使用该目录中的 inode。AUFS 支持特殊的白屏文件，使得低层的某些文件被删除；这种技术类似于其他分层系统中的删除标记（例如，LSM 数据库）。AUFS 还支持文件粒度的 COW（copy-on-write）；一旦写入，在写入被允许继续之前，较低层中的文件被复制到顶层。

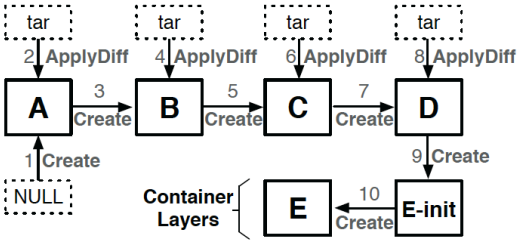

Figure 2: **Cold Run Example.** *The driver calls that are made when a four-layer image is pulled and run are shown. Each arrow represents a call (`Create` or `ApplyDiff`), and the nodes to which an arrow connects indicate arguments to the call. Thick-bordered boxes represent layers. Integers indicate the order in which functions are called.*

AUFS 驱动程序利用 AUFS 文件系统的分层和写入时复制功能，同时还直接访问 AUFS 下的文件系统。驱动程序为其存储的每个图层在底层文件系统中创建一个新目录。 ApplyDiff 简单地将归档文件解压到图层的目录中。在 Get 调用后，驱动程序使用 AUFS 创建图层及其祖先的联合视图。当调用 Create 时，驱动程序使用 AUFS 的 COW 高效地复制图层数据。不幸的是，我们将看到，文件粒度的 COW 在性能上有一些问题。

3 HelloBench

我们提出了 HelloBench，一个为快速容器启动而设计的新基准。HelloBench 直接执行 Docker 命令，所以可以独立测量 pushes，pulls 和 runs。基准包括两部分：（1）容器镜像的集合;（2）用于在所述容器中执行简单任务的测试工具。截至 2015 年 6 月 1 日，这些镜像是 Docker Hub 库的最新版本。HelloBench 包含当时 72 个可用的 57 个镜像。我们选择了可以使用最少配置运行的镜像，而不依赖于其他容器。例如，不包括 WordPress，因为一个 WordPress 容器依赖于另一个单独的 MySQL 容器。

Linux Distro:	alpine, busybox, centos, cirros, crux, debian, fedora, mageia, opensuse, oraclelinux, ubuntu, ubuntu-debootstrap, ubuntu-upstart
Database:	cassandra, crate, elasticsearch, mariadb, mongo, mysql, percona, postgres, redis, rethinkdb
Language:	clojure, gcc, golang, haskell, hylang, java, jruby, julia, mono, perl, php, pypy, python, r-base, rakudo-star, ruby, thrift
Web Server:	glassfish, httpd, jetty, nginx, php-zendserver, tomcat
Web Framework:	django, iojs, node, rails
Other:	drupal, ghost, hello-world, jenkins, rabbitmq, registry, sonarqube

Table 2: **HelloBench Workloads.** *HelloBench runs 57 different container images pulled from the Docker Hub.*

表 2 列出了 HelloBench 使用的镜像。如图所示，我们将镜像分成六大类。有些分类可能有点主观；例如，Django 镜像包含一个 Web 服务器，但大多数可能会认为它是一个 Web 框架。

HelloBench 通过运行容器中最简单的任务或等待容器准备就绪来度量启动时间。对于语言容器来说，这个任务通常包括编译或解释一个简单的“hello world”程序。Linux 发行版镜像执行一个非常简单的 shell 命令，通常是“echo hello”。对于长时间运行的服务器（特别是数据库和 Web 服务器），HelloBench 会测量容器向标准输出写入“up and ready”消息的时间。对于特别安静的服务器，会轮询一个开放的端口，直到有响应。

每个 HelloBench 镜像由许多图层组成，其中一些图层在容器之间共享。图 3 显示了层之间的关系。在 57 幅镜像中，有 550 个节点和 19 个根。在一些情况下，标记的镜像用作其他标记镜像的基础（例如，“ruby”是“rails”的基础）。只有一个镜像由单层组成：“alpine”，一个特别轻量级的 Linux 发行版。应用程序镜像通常基于非最新的 Linux 分发镜像（例如，较早的 Debian 版本）。

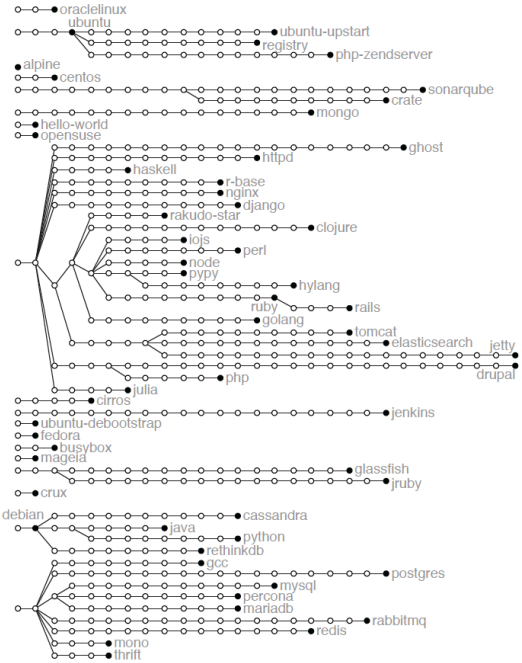

Figure 3: **HelloBench Hierarchy.** *Each circle represents a layer. Filled circles represent layers tagged as runnable images. Deeper layers are to the left.*

为了评估 HelloBench 对于常用镜像的代表性，我们计算了 2015 年 1 月 15 日每个 Docker Hub 库镜像的 pulls 数量。在此期间，图书馆镜像从 72 增加到 94。图 4 显示了按照 HelloBench 类别划分的 94 个镜像的 pulls。HelloBench 占所有 pulls 的 86%，足以代表较为流行的镜像。其中大部分是 Linux 发行版（例如 BusyBox 和 Ubuntu）。数据库（例如，Redis 和 MySQL）和网络服务器（例如，nginx）也很流行。

4 工作负载分析

在本节中，我们分析了 HelloBench 工作负载的行为和性能，提出了四个问题：容器镜像的大小以及执行时需要多少数据（4.1）？push，pull 和 run 镜像需要多长时间（4.2）？镜像数据如何在各个层之间分布，性能影响如何（4.3）？不同运行的访问模式有多相似（4.4）？所有性能测量均取自运行在

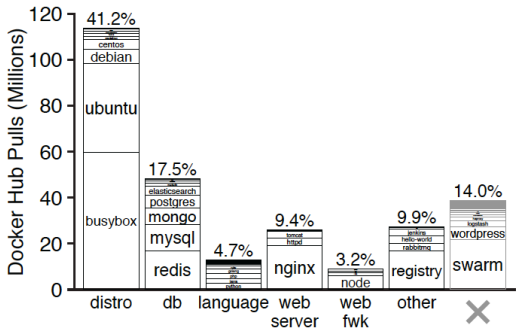

Figure 4: **Docker Hub Pulls.** *Each bar represents the number of pulls to the Docker Hub library, broken down by category and image. The far-right gray bar represents pulls to images in the library that are not run by HelloBench.*

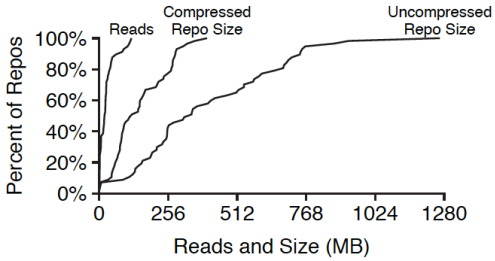

Figure 5: **Data Sizes (CDF).** *Distributions are shown for the number of reads in the HelloBench workloads and for the uncompressed and compressed sizes of the HelloBench images.*

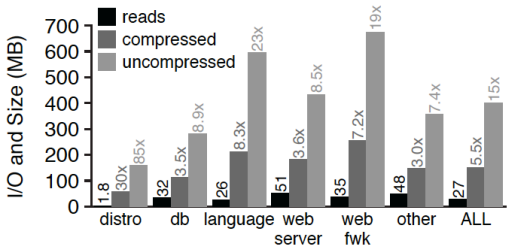

Figure 6: **Data Sizes (By Category).** *Averages are shown for each category. The size bars are labeled with amplification factors, indicating the amount of transferred data relative to the amount of useful data (i.e., the data read).*

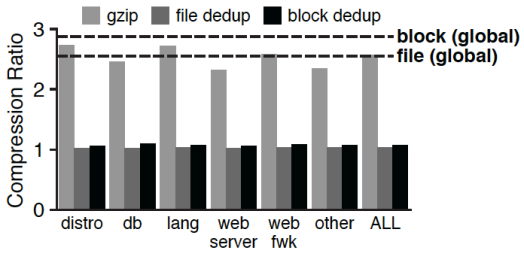

Figure 7: **Compression and Deduplication Rates.** *The y-axis represents the ratio of the size of the raw data to the size of the compressed or deduplicated data. The bars represent per-image rates. The lines represent rates of global deduplication across the set of all images.*

PowerEdge R720 主机上的虚拟机，主机 CPU 为 2 GHz Xeon CPU（E5-2620）。VMis 提供了 8 GB 的 RAM，4 个 CPU 核心以及一个由 Tintri T620 支持的虚拟磁盘。实验期间，服务器和 VMstore 没有其他的负载。

4.1 镜像数据

我们通过从 Docker Hub 获取的 HelloBench 镜像开始我们的分析。对于每个镜像，我们都进行三次测量：压缩大小，未压缩大小以及 HelloBench 执行时从镜像中读取的字节数。我们通过在 blktrace 跟踪的块设备上运行工作负载来测量数据读取。图 5 显示了这三个数字的 CDF。我们观察到只有 20MB 的数据在中位数上被读取，但是中值镜像压缩之后为 117MB，未压缩时为 329MB。

我们按照图 6 中的类别对读取以及其大小数据进行了细分。最大的浪费是发行版的工作负载（分别为压缩和未压缩的 30 倍和 85 倍），但是这个类别的绝对浪费也是最小的。语言和网络框架类别的绝对浪费最高。在所有镜像中，平均只读取 27 MB；平均来看，未压缩的镜像是 15 倍大，而镜像中，只有 6.4%的数据是需要容器启动的。

虽然 Docker 镜像在压缩成 gzip 压缩文件时小得多，但这种格式不适合运行在需要修改数据的容器上。因此，工作人员通常将数据存储为未压缩数据，这意味着压缩可以减少网络 I/O，但不会降低磁盘 I/O。重复数据的删除是压缩的简单替代方法。我们扫描 HelloBench 镜像以获得文件块之间的冗余，以计算重复数据消除的有效性。图 7 比较了文件和块在 4 KB 粒度下的 gzip 压缩率和重复数据删除。Bars 代表单一镜像的速度。而

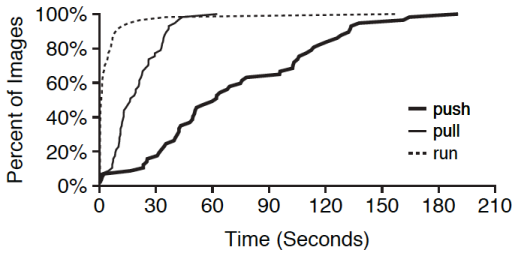

Figure 8: **Operation Performance (CDF).** *A distribution of push, pull, and run times for HelloBench are shown for Docker with the AUFS storage driver.*

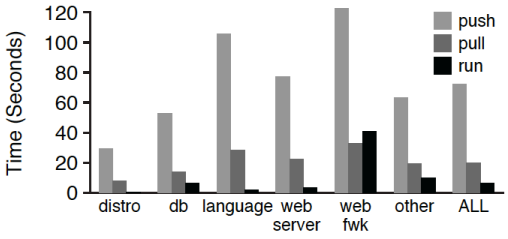

Figure 9: **Operation Performance (By Category).** *Averages are shown for each category.*

gzip 达到 2.3 到 2.7 之间的速度，重复数据删除在每个镜像基础上效果不佳。然而，对所有镜像进行重复数据删除的效率是 2.6（文件粒度上）和 2.8（块粒度上）。

含义：执行期间读取的数据量远远小于压缩或未压缩的总镜像大小。镜像数据通过压缩后的网络发送，然后读取并写入未压缩的本地存储，因此网络和磁盘的开销都很高。减少开销的一种方法是用更少的安装包来构建更精简的镜像。或者，镜像数据可能会随着容器的需要而被拉长。我们还能看到，即使与 gzip 压缩相比，基于全局块的重复数据删除是表示镜像数据的有效方法。

4.2 操作性能

一旦构建，容器化应用程序通常部署如下：开发人员将应用程序镜像一次 push 到中央注册中心，许多工作人员将镜像 pull，每个工作人员运行应用程序。我们使用 HelloBench 测量这些操作的延迟，报告图 8 中的 CDF。push，pull 和 run 的中位时间分别为 61、16 和 0.97 秒。

图 9 按工作负载类别分解了运行时间。一般情况下，run 速度快，push、pull 速度慢。对于发行版和语言类别，run 速度最快（分别为 0.36 秒和 1.9 秒）。push，pull，run 的平均时间分别为 72 秒，20 秒和 6.1 秒。因此，在远程注册表中启动一个新镜像时，启动时间的 76%将花费在拉动上。

由于 push 和 pull 是最慢的，所以我们想知道这些操作是否仅仅是高延迟，或者即使多个操作同时运行也是成本高昂，限制了吞吐量。为了研究可伸缩性，我们同时 push、pull 不同尺寸的不同数量的随机镜像。每个镜像包含一个随机生成的文件。我们使用随机镜像而不是 HelloBench 镜像来创建不同尺寸的镜像。图 10 显示了总的时间尺度大致与镜像数量和镜像大小成线性关系。因此，push、pull 不仅高延迟，而且消耗网络和磁盘资源，限制了可扩展性。

启示：集装箱启动时间以 pull 为主;新部署时间的 76%将花在 pull 上。对于迭代开发应用程序的程序员来说，使用 push 方式发布镜像将会非常缓慢，尽管这可能比多次部署并且已经发布的镜像要快一些。大多数 push 工作都是通过存储驱动程序的 Diff 函数来完成的，而大部分的推送工作都是通过 ApplyDiff 函数完成的。优化这些驱动程序功能可以提高分发性能。

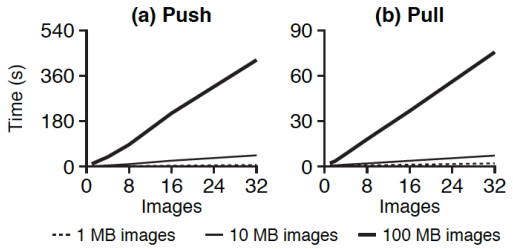

Figure 10: **Operation Scalability.** *A varying number of artificial images (x-axis), each containing a random file of a given size, are pushed or pulled simultaneously. The time until all operations are complete is reported (y-axis).*

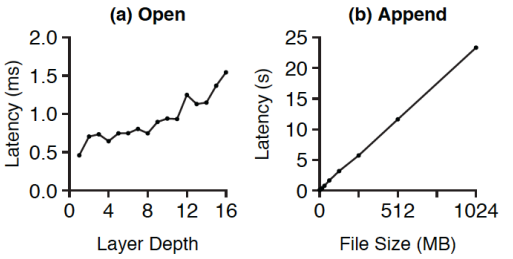

Figure 11: **AUFS Performance.** *Left: the latency of the* `open` *system call is shown as a function of the layer depth of the file. Right: the latency of a one-byte append is shown as a function of the size of the file that receives the write.*

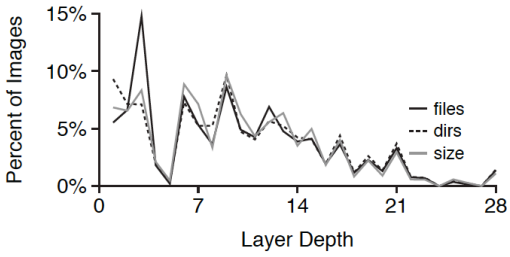

Figure 12: **Data Depth.** *The lines show mass distribution of data across image layers in terms of number of files, number of directories, and bytes of data in files.*

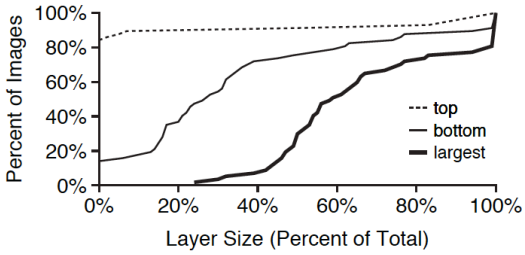

Figure 13: **Layer Size (CDF).** *The size of a given layer is measured relative to the total image size (x-axis), and the distribution of relative sizes is shown. The plot considers the topmost layer, bottommost layer, and whichever layer happens to be largest. All measurements are in terms of file bytes.*

4.3 层

镜像数据通常分成多个层。AUFS 驱动程序在运行时组成镜像的图层，为容器提供文件系统的完整视图。在本节中，我们将研究分层的性能影响以及跨层的数据分布。我们首先看看分层文件系统容易出现的两个性能问题（图 11）：深层查找和非顶层的小写入操作。

首先，我们创建 16 层，每层包含 1K 空文件。然后，用一个冷藏缓存，我们从每一层随机打开 10 个文件，测量打开延迟。图 11a 显示了结果：层深度和延迟之间存在很强的相关性。其次，我们创建两个图层，其底部包含大小不一的大文件。我们测量一个字节附加到底层存储的文件的延迟。如图 11b 所示，小写入的延迟对应于文件大小（不是写入大小），因为 AUFS 以文件粒度进行 COW。在文件被修改之前，它被复制到最上层，所以写一个字节可能需要 20 秒以上。幸运的是，对较低层次的小写操作会导致每个容器的一次性成本;后续写入将会更快，因为大文件将被复制到顶层。

考虑了层深度与性能的对应关系之后，我们现在要问，为 HelloBench 镜像存储的数据通常有多深？图 12 显示了每个深度级别的总数据的百分比（按文件数量，目录数量和字节大小）。这三个指标大致相当。有些数据和 28 级一样深，但是质量更集中在左边。超过一半的字节至少有九个深度。

我们现在考虑数据如何分布在各个层之间的差异，为每个镜像测量哪个部分存储在最顶层，最底层以及哪个层最大。图 13 显示了分布：对于 79%的镜像，最顶层包含 0%的镜像数据。相比之下，27%的数据位于最底层。大部分数据通常驻留在单一层中。

含义：对于分层文件系统，存储在较深层的数据访问速度较慢。不幸的是，Docker

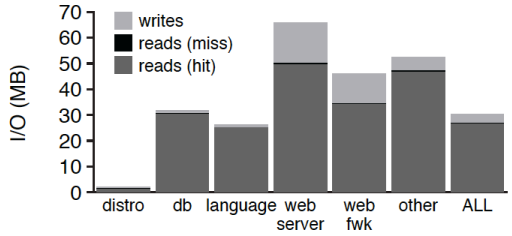

Figure 14: **Repeated I/O.** *The bars represent total I/O done for the average container workload in each category. Bar sections indicate read/write ratios. Reads that could have potentially been serviced by a cache populated by previous container execution are dark gray.*

的图片往往很深，至少有一半的文件数据深度在九或以下。平坦化层是避免这些性能问题的一种技术;但是，可能需要进行额外的复制，并会使得分层文件系统提供的其他 COW 优势无效。

4.4 缓存

我们现在考虑同一个工作人员不止一次运行同一个镜像的情况。特别的，我们想知道是否可以使用第一次执行的 I/O 预填充缓存，以避免后续运行中的 I/O。为此，我们每次连续运行 HelloBench 工作负载两次，每次收集块跟踪。我们计算第二次运行期间的读取部分，这可能会从第一次运行期间的读取所填充的缓存中受益。

图 14 显示了第二次运行的读取和写入。读取被分成命中和错过。对于给定的块，只计算第一次读取。在所有工作负载中，读/写比例是 88/12。对于发行版，数据库和语言工作负载，工作负载几乎完全由读取组成。在这些读取中，有 99%可能从以前运行的缓存数据中得到收益。

含义：在同一镜像的不同运行过程中，通常会读取相同的数据，这表明在同一台计算机上多次执行同一镜像时，缓存共享将非常有用。在拥有许多容器化应用程序的大型集群中，除非容器放置受到高度限制，否则重复执行将不太可能。而且，其他目标（例如，负载平衡和故障隔离）可能使共置不常见。但是，对于容器化的应用程序（例如 python 或 gcc）和运行在小型集群中的应用程序，重复执行可能是常见的。我们的结果表明这些后面的情况将从缓存共享中受益。

5 Slacker

在本节中，我们描述一个新的 Docker 存储驱动程序 Slacker。我们的设计是基于我们对容器工作量和五个目标的分析：（1）push、pull 速度非常快，（2）对长时间运行的容器不引起减速，（3）尽可能地重复使用现有的存储系统，（4）现代存储服务器提供的强大原语，以及（5）除了存储驱动程序插件外，不对 Docker 注册表或守护进程进行更改。

图 15 显示了运行 Slacker 的 Docker 集群的体系结构。该设计基于集中式 NFS 存储，在所有 Docker 守护进程和注册表之间共享。容器中的大部分数据不需要执行容器，因此 Docker 工作人员只能根据需要从共享存储中缓慢地提取数据。对于 NFS 存储，我们使用一个 Tintri VMstore 服务器。Docker 镜像由 VMstore 的只读快照表示。注册表不再用作图层数据的主机，而仅用作将镜像元数据与相应快照关联的名称服务器。Slacker 使用 VMstore 快照将容器转换为可共享镜像，并根据从注册表中提取的快照 ID 克隆以提供容器存储。在内部，VMstore 使用块级的 COW 来实现快照并有效克隆。

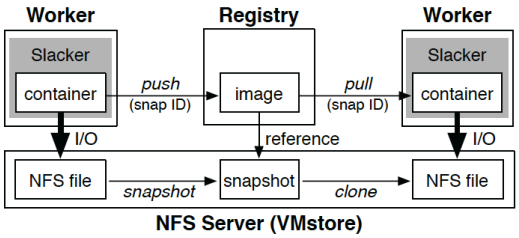

Figure 15: **Slacker Architecture.** *Most of our work was in the gray boxes, the Slacker storage plugin. Workers and registries represent containers and images as files and snapshots respectively on a shared Tintri VMstore server.*

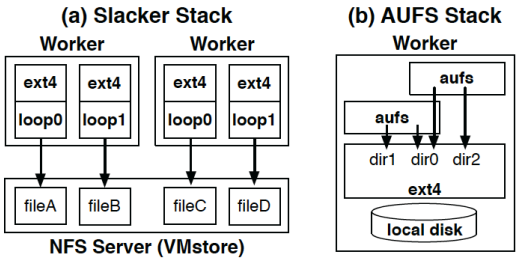

Figure 16: **Driver Stacks.** *Slacker uses one ext4 file system per container. AUFS containers share one ext4 instance.*

Slacker 的设计基于我们对容器工作负载的分析;特别是，以下四个设计小节（5.1 到 5.4）对应于前四个分析小节（4.1 到 4.4）。我们通过讨论对 Docker 框架本身的可能修改作为结语，这将对非传统存储驱动程序提供更好的支持。

5.1 存储层

我们的分析表明，在容器开始有效的工作之前，实际上只需要 pull 6.4％的数据（4.1）。为了避免浪费对未使用数据的 I/O，Slacker 将所有容器数据存储在共享的 NFS 服务器（Tintri VMstore）上;工作人员只提取需要的数据。图 16a 说明了这个设计：每个容器的存储被表示为一个 NFS 文件。 Linux 环回（5.4）用于将每个 NFS 文件视为一个虚拟块设备，可以作为正在运行的容器的根文件系统进行挂载和卸载。Slacker 将每个 NFS 文件格式化为 ext4 文件系统。

图 16b 比较了 Slacker 堆栈和 AUFS 堆栈。虽然都使用 ext4（或其他本地文件系统）作为关键层，但有三个重要的区别。首先，ext4 由 Slacker 中的网络磁盘支持，但是由具有 AUFS 的本地磁盘支持。因此，Slacker 可以懒惰地在网络上获取数据，而 AUFS 必须在容器启动之前将所有数据复制到本地磁盘。

其次，AUFS 在文件级上的 COW 高于 ext4，因此易受分层文件系统（4.3）所面临的性能问题的影响。相反，Slacker 层在文件级被有效地处理。然而，Slacker 仍然通过利用在 VMstore（5.2）中实现的块级 COW 从 COW 中受益。此外，VMstore 在内部对相同的块进行重复数据删除，从而进一步节省在不同 Docker 工作人员上运行的容器之间的空间。

第三，AUFS 使用单个 ext4 实例的不同目录作为容器的存储，而 Slacker 通过不同的 ext4 实例来备份每个容器。这种差异呈现出一个有趣的折衷，因为每个 ext4 实例都有它自己的日志。通过 AUFS，所有容器将共享相同的日志，从而提供更高的效率。然而，已知日志共享会导致优先级反转，从而破坏 QoS 的保证，这是多租户平台（如 Docker）的一个重要特性。内部分裂是 NFS 存储被分成许多小的非全部 ext4 实例的另一个潜在问题。幸运的是，VMstore 文件是稀疏的，所以 Slacker 不会遇到这个问题。

5.2 VMstore 集成

之前，我们发现与 run 相比，Docker push 和 pull 相当慢（4.2）。运行速度很快，因为使用 AUFS 提供的 COW 功能，可以从镜像

初始化新容器的存储。相比之下，传统驱动程序的 push、pull 速度较慢，因为它们需要在不同机器之间复制较大的图层，所以 AUFS 的 COW 功能无法使用。与其他 Docker 驱动程序不同，Slacker 是建立在共享存储上的，因此在守护进程和注册表之间进行 COW 共享是概念上可行的。

幸运的是，VMstore 使用基于 REST 的辅助 API 来扩展其基本的 NFS 接口，其中包括两个相关的 COW 函数，快照和克隆。快照调用为 NFS 文件创建一个只读快照，克隆从快照创建一个 NFS 文件。快照不会出现在 NFS 命名空间中，但具有唯一的 ID。文件级快照和克隆是功能强大的基元，用于构建更高效的日志记录，重复数据删除和其他常见存储操作。在 Slacker 中，我们分别使用快照和克隆来实现 Diff 和 Apply-Diff。

图 17a 显示了一个运行 Slacker 的守护进程在 push 时如何与 VMstore 和 Docker 注册表进行交互。Slacker 要求 VMstore 创建表示图层的 NFS 文件的快照。 VMstore 获取快照，并返回一个快照 ID（约 50 个字节），本例中为“212”。 Slacker 将 ID 嵌入压缩的 tar 文件中，并将其发送到注册表。Slacker 将 ID 嵌入到 tar 中以实现向后兼容性：未经修改的注册表期望接收 tar 文件。如图 17b 所示，pull 本质上是相反的。Slacker 从注册表中接收一个快照标识符，从中可以克隆用于容器存储的 NFS 文件。Slacker 的实现速度很快，因为（a）层数据从不压缩或未压缩，（b）层数据永远不会离开 VMstore，所以只有元数据通过网络发送。

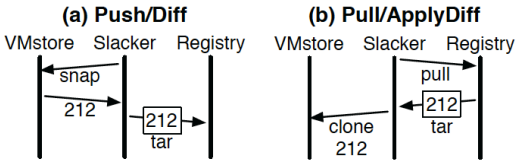

Figure 17: **Push/Pull Timelines.** *Slacker implements* `Diff` *and* `ApplyDiff` *with* `snapshot` *and* `clone` *operations.*

在 Slacker 的实现中，名称“Diff”和“ApplyDiff”是轻微的误称。特别是，Diff（A，B）应该返回一个三角形，从这个三角形已经有 A 的另一个守护进程可以重建 B.通过 Slacker，层被有效地置于名称空间级别。因此，Diff（A，B）不是返回一个 delta 值，而是返回另一个 worker 可以从中获得 B 副本的引用。

Slacker 与其他运行非 Slacker 驱动程序的守护进程部分兼容。当 Slacker 拉取一个 tar 时，在处理它之前，它会查看流式 tar 的前几个字节。如果 tar 包含图层文件，Slacker 就会退化为简单的解压缩而不是克隆。因此，Slacker 可以 pull 其他驱动 push 的镜像，尽管缓慢。但是，其他驱动程序将无法 pull Slacker 镜像，因为他们不知道如何处理嵌入在 tar 文件中的快照 ID。

5.3 优化

快照和克隆镜像通常由许多层组成，超过一半的 HelloBench 数据深度至少为 9（4.3）。对于这样的数据，块级别的 COW 比文件级别的 COW 具有固有的性能优势，因为遍历块映射索引比遍历底层文件系统的目录更简单。

但是，深层次的镜像对 Slacker 来说仍然是一个挑战。正如所讨论过的（5.2），Slacker 层是被加密的，所以安装任何一个层

将提供一个容器可以使用的文件系统的完整视图。不幸的是，Docker 框架没有“受保护层”的概念。当 Docker pull 一张图片时，它会抓取所有图层，并通过 ApplyDiff 传递给驱动程序。对于 Slacker 来说，最上面的层就足够了。对于 28 层镜像（例如，码头），额外的克隆是昂贵的。

我们的目标之一是在现有的 Docker 框架中工作，所以我们不用修改框架来消除不必要的驱动程序调用，而是使用懒惰的克隆来优化它们。我们发现 pull 的主要成本不是快照 tar 文件的网络传输，而是 VMstore 克隆。虽然克隆只需要几分之一秒，但执行其中的 28 个会对等待时间产生负面影响。因此，Slacker 代替了一个记录快照 ID 的本地元数据。ApplyDiff 只是设置这个元数据，而不是立即克隆。如果 Docker 在某个时候调用了 Get 层，那么 Slacker 将在这一点之前执行一个真正的克隆。

我们还使用快照 ID 元数据进行快照缓存。特别是，Slacker 实现了 Create，这个层创建了一个层（2.2）的逻辑副本，紧接着是一个克隆（5.2）。如果许多容器是从同一个镜像创建的，则 Create 会在同一个图层上多次调用。 Slacker 不是为每个 Create 创建一个快照，而只是第一次执行快照，随后重新使用快照 ID。如果图层被挂载，则图层的快照缓存将失效。

快照缓存和惰性克隆的组合可以使 Create 非常高效。尤其是，从 A 层复制到 B 层可能只涉及从 A 的快照缓存条目复制到 B 的快照缓存条目，而不会特别调用 VMstore。在图 2 的背景部分（2.2）中，我们展示了 10

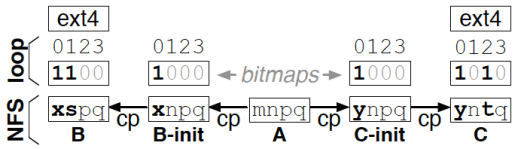

Figure 18: **Loopback Bitmaps.** *Containers B and C are started from the same image, A. Bitmaps track differences.*

个 Create 和 ApplyDiff 调用，这些调用是为了简单的四层镜像的 pull 和 run 而发生的。没有懒惰的缓存和快照缓存，Slacker 需要执行 6 个快照和 10 个克隆。通过我们的优化，Slacker 只需要做一个快照和两个克隆。在第 9 步中，Create 做了一个懒惰的克隆，但是 Docker 在 E-init 层上调用了 Get，所以必须执行一个真正的克隆。对于第 10 步，Create 必须同时执行快照和克隆，以生成并装载图层 E 作为新容器的根目录。

5.4 Linux 内核修改

我们的分析表明，从同一镜像开始的多个容器倾向于读取相同的数据，这表明缓存共享可能是非常有用的（4.4）。AUFS 驱动程序的一个优点是 COW 在底层文件系统之上完成。这意味着不同的容器可能在该底层文件系统中预热并使用相同的高速缓存状态。Slacker 在 VMstore 中执行 COW，位于本地文件系统的级别之下。这意味着两个 NFS 文件可能是相同快照的克隆，但是缓存状态不会被共享，因为 NFS 协议不是围绕 COW 共享的概念构建的。缓存重复数据删除可以帮助节省缓存空间，但这不会阻止初始 I/O。重复数据删除不可能实现两个块相同，两个块都从 VMstore 通过网络传输。在本节中，我们描述了在 NFS 文件级别实现在 Linux 页面缓存中共享的技术。

为了实现 NFS 文件之间的客户端缓存

共享，我们修改 NFS 客户端上方的层，以增加对 VMstore 快照和克隆的了解。特别的，我们使用位图来跟踪类似的 NFS 文件之间的差异。所有写入 NFS 文件的都是通过回送模块，所以回送模块可以自动更新位图来记录新的更改。快照和克隆由 Slacker 驱动程序启动，因此我们扩展了回送 API，以便 Slacker 可以通知模块文件之间的 COW 关系。

图 18 用一个简单的例子说明了这个技术：从同一个镜像启动两个容器 B 和 C.当启动容器时，Docker 首先创建两个初始层（B-init 和 C-init）。Docker 在这些层中创建一些小的 init 文件。请注意，在初始层中，“m”被修改为“x”和“y”，并且第零位被移动到“1”以标记改变。Docker 从 B-init 和 C-init 创建最顶层的容器层 B 和 C. Slacker 使用新的回送 API 将 B-init 和 C-init 位图分别复制到 B 和 C 中。如图所示，当容器运行并写入数据时，B 和 C 位图会累积更多的突变。 Docker 没有明确区分 init 层和其他层作为 API 的一部分，但 Slacker 可以推断出层类型，因为 Docker 恰好为 init 层的名称使用了“-init”后缀。

现在假设容器 B 读取块 3。环回模块在位置 3 处看到未修改的“0”位，表示块 3 在文件 B 和 A 中是相同的。因此，环回模块将读取发送给 A 而不是 B，因此填充 A 的缓存状态。现在假设 C 读取块 3。C 的块 3 也是未修改的，所以读取再次被重定向到 A。现在，C 可以从 A 的缓存状态中受益，B 被其前面的读取填充。

当然，对于 B 和 C 不同于 A 的块，重要的是读取不重定向。假设 B 读取块 1，然后 C 从块 1 读取。在这种情况下，B 的读取不会填充缓存，因为 B 的数据不同于 A。类似地，假设 B 读取块 2，然后 C 从块 2 读取。在这种情况下，C 的读取不会使用缓存，因为 C 的数据不同于 A.

5.5 Docker 框架讨论

我们的目标之一是不对 Docker 注册表或守护进程进行更改，除了可插拔存储驱动程序之外。尽管存储驱动程序接口非常简单，但对于我们的需求已经足够了。但是，Docker 框架有一些改变，可以更优雅的实现 Slacker。首先，如果注册表可以表示不同的图层格式（5.2），那么驱动程序之间的兼容性将会很有用。目前，如果一个非 Slacker 层拉了一个 Slacker 推送的层，它会不友好的失败。格式跟踪可以提供一个友好的错误消息，或者，理想情况下，启用挂钩进行自动格式转换。其次，增加“受保护图层”的概念是有用的。特别是，如果一个驱动程序可以通知框架一个图层，Docker 就不需要在获取图层时获取祖先图层。这将消除我们对懒惰克隆和快照缓存的需求（5.3）。第三，如果框架明确标识了 init 层，那么 Slacker 就不需要依赖图层名称作为提示（5.4）。

6 评估

我们使用和第四节中相同的硬件进行评估。为了公平的比较，我们还使用了，与运行 AUFS 实验的虚拟机的虚拟磁盘相同的、用于 Slacker 存储的 VMstore。

6.1 HelloBench 工作负载

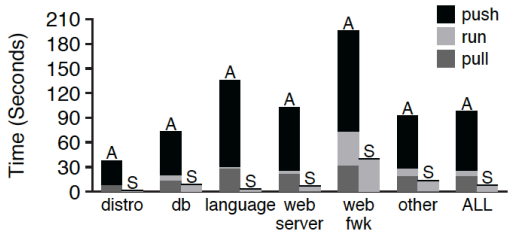

Figure 19: **AUFS vs. Slacker (Hello).** *Average push, run, and pull times are shown for each category. Bars are labeled with an "A" for AUFS or "S" for Slacker.*

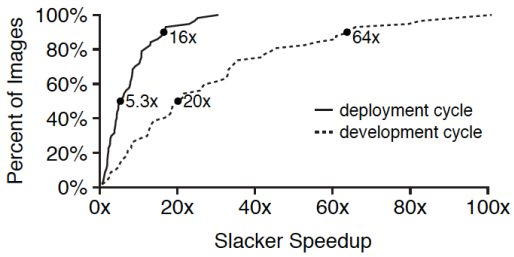

Figure 20: **Slacker Speedup.** *The ratio of AUFS-driver time to Slacker time is measured, and a CDF shown across HelloBench workloads. Median and 90th-percentile speedups are marked for the development cycle (push, pull, and run), and the deployment cycle (just pull and run).*

之前，我们看到，使用 HelloBench 时，pull、push 次数占主导地位，而运行时间非常短（图 9）。我们用 Slacker 重复这个实验，在图 19 的 AUFS 结果的旁边展示了新的结果。平均来说，push 阶段快 153 倍，pull 阶段快 72 倍，但 run 阶段慢了 17%。

不同的 Docker 操作被用于不同的场景。一个用例是开发：每次更改代码之后，开发人员将应用程序 push 到注册表，再将其 pull 到多个工作节点，然后在节点上运行。另一个是部署：一个不经常修改的应用程序由注册表托管，但偶尔的负载突发或重新平衡需要拉动并在新的工作人员上运行。图 20 显示了这两种情况下 Slacker 相对于 AUFS 的加速。对于中等工作负载，Slacker 分别在部署和开发周期中的启动速度提高了 5.3 倍和 20 倍。加速是不稳定的：几乎所有的工作负载都有适度的改善，但是 10%的工作负载在部署和开发方面至少提高了 16 倍和 64 倍。

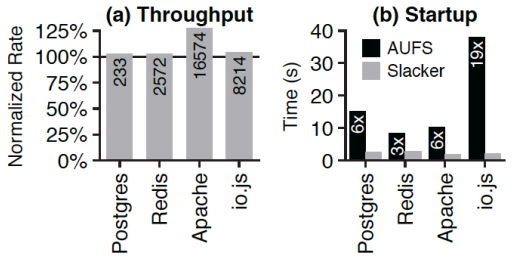

Figure 21: **Long-Running Workloads.** *Left: the ratio of Slacker's to AUFS's throughput is shown; startup time is included in the average. Bars are labeled with Slacker's average operations/second. Right: startup delay is shown.*

6.2 长期性能

在图 19 中，我们看到 Slacker 的 push、pull 速度快得多，run 速度也比较慢。这是预料之中的，因为在任何数据传输之前 run 就开始了，二进制数据只是在需要的时候才被延迟地传输。我们现在运行几个长期的容器实验。我们的目标是，一旦 AUFS 完成 pull 所有镜像数据，Slacker 亦可缓慢地加载热镜像数据，AUFS 和 Slacker 具有相同的性能。

对于我们的评估，我们选择两个数据库和两个 Web 服务器。对于所有的实验，我们执行五分钟，每秒测量。每个实验都从一个 pull 开始。我们使用 pgbench 来评估 PostgreSQL 数据库。我们使用自定义基准测试来评估内存数据库 Redis，这个基准测试可以以相同的频率获取、设置和更新密钥。我们使用 wrk 基准来评估 Apache Web 服务器，反复获取静态页面。最后，我们使用 wrk 基准来评估 io.js，一个类似于 node.js 的基于 JavaScript 的 web 服务器，以反复获取动态页面。

图 21a 显示了结果。AUFS 和 Slacker 的性能大致相同。尽管驱动程序在长期性能方面是类似的，但是图 21b 显示 Slacker 容器处理请求比 AUFS 早开始 3-19 倍。

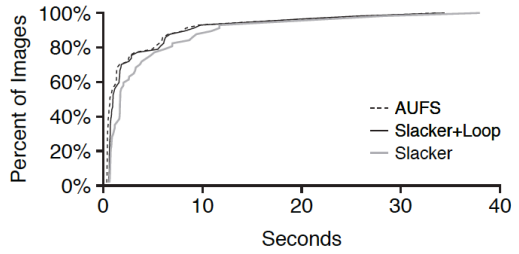

Figure 22: **Second Run Time (CDF).** *A distribution of run times are shown for the AUFS driver and for Slacker, both with and without use of the modified loopback driver.*

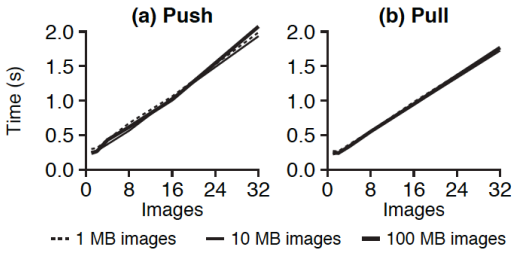

Figure 23: **Operation Scalability.** *A varying number of artificial images (x-axis), each containing a random file of a given size, are pushed or pulled simultaneously. The time until all operations are complete is reported (y-axis).*

6.3 缓存

我们已经发现，相对于 AUFS，Slacker 提供了更快的启动时间和更长期的性能。Slacker 处于劣势的一种情况是在同一台机器上多次运行相同的短工作负载。对于 AUFS，第一次运行会很慢，但后续运行将会很快，因为镜像数据将被存储在本地。而且，COW 是在本地完成的，所以从同一个启动镜像运行的多个容器将受益于共享的 RAM 缓存。

另一方面，Slacker 依赖于 Tintri VMstore 在服务器端执行 COW。这种设计能够实现快速分发，但是一个缺点是，NFS 客户端并不能意识到文件之间的冗余，而不会改变内核。我们将修改后的环回驱动程序与 AUFS 作为共享缓存状态的一种手段。为此，我们运行两次 HelloBench 工作负载，测量第二次运行的延迟。我们将 AUFS 与 Slacker 比较。

图 22 显示了使用三个系统的所有工作负载的运行时间的 CDF。尽管 AUFS 仍然是最快的，但内核修改显着加速了 Slacker。Slacker 的平均运行时间是 1.71 秒;对回送模块的内核修改是 0.97 秒。尽管 Slacker 避免了不必要的网络 I/O，但 AUFS 驱动程序可以直接缓存 ext4 文件数据，而 Slacker 缓存 ext4 下的块，这可能会带来一些开销。

6.4 可扩展性

在 4.2 节，我们看到 AUFS 在镜像大小和被操作的镜像数量方面的 push、pull 效果不佳。 我们用 Slacker 重复我们之前的实验（图 10），再次创建合成镜像并同时 push 或 pull 不同数量的镜像。

图 23 显示了结果：镜像大小不再像 AUFS 那样重要。总时间仍然与正在处理的镜像数量相关，但绝对时间要好得多; 即使是 32 张镜像，push、pull 的时间最多也只有 2 秒左右。 同样值得注意的是，push 时间与 Slacker 的 push 时间相似，而 AUFS 的 push 时间更为昂贵。 这是因为 AUFS 使用压缩进行大量数据传输，压缩通常比解压缩成本高。

7 案例分析：MultiMake

Drew Houston 在启动 Dropbox 时发现，构建一个广泛部署的客户端涉及到很多“不完善的操作系统”，代码不能与平台的特性很好的兼容。例如，一些错误只会在瑞典版本的 Windows XP Service Pack 3 上出现，而其他类似的部署则不会受到影响。避免这些错误的一种方法是在许多不同的环境中广泛地测试软件。有几家公司提供集装箱化的

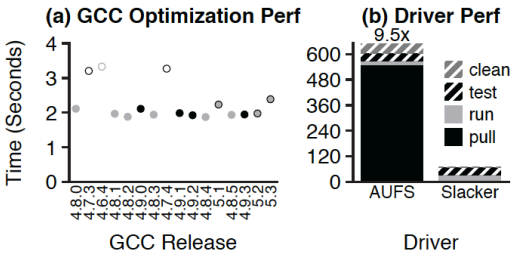

Figure 24: **GCC Version Testing.** *Left: run time of a C program doing vector arithmetic. Each point represents performance under a different GCC release, from 4.8.0 (Mar '13) to 5.3 (Dec '15). Releases in the same series have a common style (e.g., 4.8-series releases are solid gray). Right: performance of MultiMake is shown for both drivers. Time is broken into pulling the image, running the image (compiling), testing the binaries, and deleting the images from the local daemon.*

集成测试服务，其中包括针对几十种 Chrome，Firefox，Internet Explorer 和其他浏览器版本的 Web 应用进行快速测试。当然，这种测试的广度受限于不同测试环境可以提供的速度。

我们演示了使用新工具 MultiMake 进行快速容器配置的有效性。在源目录上运行 MultiMake 使用最后的 16 个 GCC 版本构建 16 个不同版本的目标二进制文件。每个编译器都由中央注册表托管的 Docker 镜像表示。比较二进制文件有很多用途。例如，某些安全检查已知可以通过某些编译器版本进行优化。 MultiMake 使开发人员能够评估跨 GCC 版本的这种检查的健壮性。

MultiMake 的另一个用途是评估不同 GCC 版本的代码片段的性能，这些版本采用不同的优化。作为一个例子，我们用一个简单的 C 程序对多个矢量进行 20M 运算，如下所示：

```
for (int i = 0; i < 256; i++) {
    a[i] = b[i] + c[i] * 3;
}
```

图 24a 显示了结果：最近的 GCC 版本很好地优化了向量操作，但是由 4.6 和 4.7 系列编译器生成的代码却要花费大约 50%的时间来执行。GCC 4.8.0 生成了更快速的代码，尽管它在 4.6 和 4.7 版本发布之前就已经发布了。图 24b 显示收集这些数据的速度比 Slacker 快 9.5 倍，因为大部分时间都是用 AUFS 进行的。清理是 AUFS 比 Slacker 更昂贵的另一个操作。在 AUFS 中删除一个图层涉及删除成千上万个小型的 ext4 文件，而在 Slacker 中删除一个图层只涉及删除一个大的 NFS 文件。

快速运行不同版本的代码的能力可以使 MultiMake 以外的其他工具受益。例如，git bisect 通过在一系列提交中执行二分搜索来查找引入了一个 bug 的提交。除了基于容器的自动构建系统，与 Slacker 集成的对分工具可以快速搜索大量的提交。

8 结论

快速启动已经应用于各种各样的可扩展 Web 服务、集成测试和分布式应用程序上。Slacker 填补了容器与虚拟机这两个解决方案之间的差距。容器本质上是轻量级的，但目前的容器管理系统，如 Docker 和 Borg 在分发镜像方面速度很慢。相比之下，虚拟机虽然是重量级的，但是对虚拟机镜像的多重部署已经有了深入的研究和优化。Slacker 为容器高效的部署提供了有效的方案，其借鉴了 VM 镜像管理的想法，例如惰性传播等，以及引入了一些特定于 Docker 的优化。通过这些技术，Slacker 整体在部署速度上提升了 5 倍，在开发速度上提升了 20 倍。