工学硕士学位论文

大规模虚拟网络
镜像分发优化策略研究

RESEARCH ON OPTIMIZING STRATEGY OF IMAGE DISTRIBUTION IN LARGE SCALE VIRTUAL NETWORK

胡尧

哈尔滨工业大学

2018 年 6 月

国内图书分类号：TP315　　学校代码：10213
国际图书分类号：681.5　　密级：公开

工学硕士学位论文

大规模虚拟网络镜像分发优化策略研究

硕士研究生：胡尧
导　　师：张伟哲 教授
申请学位：工学硕士
学　　科：计算机科学与技术
所在单位：计算机科学与技术学院
答辩日期：2018 年 6 月
授予学位单位：哈尔滨工业大学

Classified Index: TP315
U.D.C: 681.5

Dissertation for the Master Degree in Engineering

RESEARCH ON OPTIMIZING STRATEGY OF IMAGE DISTRIBUTION IN LARGE SCALE VIRTUAL NETWORK

Candidate: Hu Yao
Supervisor: Prof. Zhang Weizhe
Academic Degree Applied for: Master of Engineering
Speciality: Computer Science and Technology
Affiliation: School of Computer Science and Technology
Date of Defence: June, 2018
Degree-Conferring-Institution: Harbin Institute of Technology

摘　要

随着虚拟化技术的不断发展，虚拟网络系统在应用场景上变得越来越复杂，规模越来越大，如何快速高效地部署大规模虚拟网络系统逐渐成为一个重要的课题。虚拟网络系统的部署性能通常受多个因素的影响，包括系统节点的物理性能、虚拟机 Hypervisor 的软件性能、镜像存储文件系统性能和网络传输性能等。本文从虚拟机镜像存储的角度，对虚拟网络系统的部署进行优化。

传统虚拟网络部署方案是将磁盘完整拷贝到计算节点之后，再进行引导，然而虚拟机引导实际所需的数据占整个磁盘数据的比例很小，虚拟机的引导过程并不需要将整个磁盘数据都传输到计算节点。本文将对虚拟机的启动过程进行分析，记录操作系统引导产生的 I/O 请求，对操作系统实际读取的数据进行统计，精确提取虚拟机引导过程实际需要的数据，并将这些数据按照请求先后排序后从原有镜像中分离出来，作为启动镜像，在部署时候优先将这部分数据传输到计算节点。在此基础上，我们将在已有的文件系统之上做一个过滤层，实现对虚拟机磁盘镜像格式的重新规划，以便于控制虚拟机的读请求逻辑，实现虚拟机启动和镜像传输同步进行。

除此之外，在很多虚拟网络部署场景中，有大量简单任务的虚拟机实例。虽然他们执行的任务非常简单，但是系统却为它们分配了一个虚拟化实例应有的资源，造成大量的资源闲置。本文将容器技术融入到虚拟网络部署系统中，用以替代大量简单的虚拟化实例，并打通容器和传统虚拟化网络之间的网络连接，对虚拟化接口进行抽象，隐藏容器与虚拟机的区别，以实现一个统一的虚拟网络部署框架系统。

关键词：虚拟网络；镜像；云计算；容器

Abstract

With the continuous development of virtualization technology, virtual network systems have become more and more complex in many scenarios, and they have become larger and larger. How to quickly and efficiently deploy large-scale virtual network systems has gradually become an important issue. The performance of deploying a virtual network system is often affected by a number of factors, including the physical system nodes, the virtual machine hypervisor, the file system images storage in, and the performance of the network transmission. This paper optimizes the deployment of virtual network systems from the perspective of virtual machine image storage.

The traditional virtual network deployment scheme is to completely copy the image file to the compute node before OS booting. However, the data actually required by virtual machine when booting accounts for a small proportion of the entire disk data. The virtual machine boot process does not require the entire disk data to be transferred to the compute node. This paper will analyze the booting process of the virtual machine, record the I/O requests generated by the operating system, collect statistics on the data actually read by the operating system, and accurately extract the data actually required by the virtual machine when booting. After sorted, the requests are separated from the original image and used as the boot image. During the deployment, this part of data is preferentially transmitted to the compute node. On this basis, we will do a filtering layer on the existing file system to re-plan the format of the disk image of the virtual machine so as to control the logic of reading request by the virtual machine and synchronize the startup of the virtual machine and the image transmission.

In addition, in many virtual network deployment scenarios, there are a large number of virtual machine instances which is very simple. However, the system allocates resources for them to a virtualized instance, causing a lot of resources to be idle. This paper integrates container technology into a virtual network deployment system to replace a large number of simple virtualized instances and connect the network between the container and the traditional virtual network to abstract the virtualized interfaces and hide the difference between containers and virtual machines and implement a unified virtual network deployment system.

keywords: virtual network, image, cloud computing, container

目 录

第 1 章　绪　论

1.1 课题来源及研究的背景和意义

1.1.1 课题研究的背景

随着芯片工艺的提升，硬件框架的升级，计算机的集成度越来越高。如今移动设备的计算能力，已经超越了几年前的桌面主流平台的计算能力。芯片的计算能力越来越强，给上层的应用提供的更大的适用空间。作为云计算服务的基础，虚拟化技术从 60 年代在 IBM 大型机系统里面诞生开始，一直在不断的更新换代，推动着云计算发展，为如今大规模集群、虚拟网络部署、按需服务等需求提供了可能性。

计算主机的发展经历了单核到多核、单机到多机、单节点到集群的发展，核心规模上已达到百万级别，对于这些虚拟资源的管理，虚拟化技术在其中起了重要作用。虚拟化技术将这些计算资源集中管理，按需分配，避免了单节点资源划分造成的浪费。在资源使用之后，会被虚拟机管理器释放，并送回资源池。除了计算资源，虚拟化技术同样可以用于内部存储、外部存储、网络和服务的虚拟化。

在虚拟化技术[1]的支撑下，云计算服务逐步成型。 规模可以做得很大，Google 等服务提供商已经拥有了上百万个节点的商用云，云计算可以灵活利用这些计算资源[2]，按需服务，降低用户的成本，利用多副本容灾容错技术，云计算可以提供较高的可靠性，针对不同的应用场景，云计算可以提供服务类型，云计算往往分为三个层次：IaaS、PaaS、SaaS[3, 7]。对用户而言，相对常规计算，云计算服务更能直接的满足用户需求，不需要用户关心服务之外的相关配置。例如，对于一个想要搭建博客的个人用户，云计算可以为用户提供预设的博客环境（如 Wordpress、Jekyll 等），用户只需要将内容发布到云端服务器。如果不使用云计算，用户需要购置主机，安装操作系统，安装依赖软件，并配置博客使用环境，同时还需要向 ISP（Internet Service Provider，互联网服务提供商）申请互联网地址，并维护整个系统环境。

和常规的云计算服务类似，大规模虚拟网络的快速灵活构建也需要虚拟化技术的支持。例如在网络靶场中，需要复现一个网络场景，其输入是一个网络拓扑，网络中包含多种类型的主机、路由器、交换机和他们之间的连接关系，输出是一个模

拟的网络复现场景[4]。一般来说，场景往往包括成千上万的虚拟节点，节点之间的网络带宽通常是千兆或者万兆，这样的场景对计算资源、网络资源的性能较为敏感，其部署的耗时往往是评价一个靶场性能的重要指标。同时，对靶场的灵活性有较高的要求。这样的场景，现有的云计算解决方案并不能很好的适应，往往需要进行优化，其中对镜像存储和部署策略的优化方案对大规模虚拟网络系统部署[5]的性能有着重要的影响。

存储技术从诞生到现在，容量从最早几 MB 的软盘发展到如今几十 TB 的硬盘，读写性能由最早磁带的几 KB 每秒发展到 SSD 的数 GB 每秒。但虚拟化技术的发展过程中，存储性能一直是一个挑战。为了提高连续读写的性能和数据的稳定性，引入了 RAID（Redundant Arrays of Independent Disks，磁盘阵列）[6,32]等技术；为了扩大存储规模，引入了分布式存储技术；为了降低 IO 负载，引入了缓存技术。在虚拟网络环境中，虚拟实例的存储镜像通常是以文件的形式保存在宿主计算节点之上，对于虚拟实例来说，镜像文件即是磁盘。不同类型的虚拟实例通常是基于不同镜像文件构建得到，在大规模虚拟网络环境中，这样的镜像文件大小与数量会和网络规模与场景的复杂程度呈正相关。为了提升大规模环境下磁盘镜像的性能，出现了很多对镜像文件结构的优化方案，包括去冗余技术、混合存储技术等。但这些优化方案的环境大多在实例已经完成启动或者镜像已经传输到计算节点，研究的重点在于提升实例的运行效率。

1.1.2 课题研究的目的和意义

在大规模虚拟网络系统部署的过程中，镜像存储[8-11]和网络分发[17-19]的效率对系统部署的性能有着较高的影响。部署需要从存储节点将镜像资源通过网络分发到各个计算节点，然后由计算节点构建虚拟实例，并引导启动，提供服务。这些镜像资源大小各不相同，从几百 MB（精简型 Linux 系统）到几十 GB[25]（Window Server 服务器）都可能存在，而部署环境中的网络带宽往往是 100Mbps 或者 1Gbps。通常来说，一个镜像引导完成，通常是指计算节点从存储节点通过网络获取镜像、配置实例、引导操作系统并最后提供服务。在这个过程中，获取镜像往往是耗时最长的。例如一个 4GB 大小的镜像，通过百兆网络（100Mbps），理论上需要 6 分钟的时间。而实例配置、操作系统的引导往往能够在 1 分钟之内完成。在大规模的虚拟网络部署环境中，有着成千上万的虚拟实例需要部署，这种问题更为明显。

针对这个问题，考虑到物理网络带宽的硬件限制，一般从镜像结构和部署策略两个方面进行优化。

对于镜像存储结构，可以进行压缩，降低镜像的体积。例如支持稀疏存储结构的磁盘镜像格式仅在镜像文件中保存实际使用的数据，磁盘中未使用的部分将不会被保存在文件中。也可以对磁盘格式进行调整，实现层级存储。相同操作系统的虚拟实例镜像往往有很多相同的数据，例如操作系统内核、基础类库等，如果对这些数据进行共享，去除冗余，可以降低网络资源的占用。

从策略上说，可以根据需求，对部署的顺序进行调整，对实例设置优先级，降低不同实例部署之间的资源浪费，提高网络资源的利用率，以实现在有限带宽资源的限制下的最优部署方案。

除此之外，基于容器的虚拟化技术[26-28]在一定的应用场景中，非常适合大规模的部署。容器技术充分利用了宿主机的资源，大幅降低冗余，能够灵活，快速，大规模地构建。由于容器本质上是宿主机的进程，其构建、销毁的开销相对虚拟机来说小很多。在性能方面，传统虚拟机在软件层面上面会做一些硬件的虚拟化，不可避免地带来性能损失，而容器的性能却是跟运行在物理机上并无差别。但是容器也是有自己的限制，比如不能运行宿主机操作系统之外的虚拟实例。

本文在对传统虚拟化技术中镜像存储过程进行深度分析，以数据块为粒度进行优化。在大规模虚拟化部署场景了，很多时候我们只是需要将虚拟机引导起来，运行较为简单的程序，并不需要完整的操作系统功能。从这个角度看，虚拟机对镜像的需求并非完整的镜像。以此为基础，我们将对镜像的存储结构重新规划，并实现一个在有限带宽下快速引导的分发方案。除此之外，我们还将实现一个虚拟机容器混合部署系统，以在满足部署需求的条件下，充分利用容器的优势，提高部署效率。

1.2 国内外研究现状

1.2.1 基于镜像存储的优化

Constantine [12]等人针对 DSL 网络环境，使用 COW（Copy-On-Write，写时复制）[13-16]技术对快照更新进行优化。他们对镜像进行了分层，初始镜像作为基础，每当有数据写入，都会在新的缓存层进行写入，而不会对基础镜像进行改动。当需要对镜像进行分发时候，只需要将快照分发出去，并覆盖到基础镜像之上，形成新的一层数据。除此之外，他们还对数据进行了压缩，对比和分析了几

种压缩算法的性能和效果。他们实现了基础的基于写实复制方案优化的镜像分发方式。

Matthias[20]等人对分层技术进行了完善[21-23]。Constantine 等人的分层是面向镜像数据的分层，新的数据到来都会形成一个新的层，分层的依据是快照的捕获阶段。而 Matthias 等人将镜像按照服务来划分，一个镜像可以划分为三个层面：base layer、vendor layer 和 user layer。其中 base layer 包含了操作系统需要的数据，vendor layer 包含了驱动、服务框架等数据内容，user layer 就是用户的应用程序、服务和数据内容，例如 apache、nginx、ftp 等服务。他们的思路是按需获取需要的数据层，base layer 是系统层，所有的虚拟实例的启动都需要这层数据，所以在分发阶段必须被发布到各个计算节点，而 user layer 则有很多种不同的类型，则是按照目标需求按需选取。

陈彬[24]等对分布式环境下虚拟机部署问题进行了分析，设计和实现了基于 COW 技术的按需分发机制，在满足用户需求的条件下降低镜像传输的数据量，实现镜像的快速分发，降低虚拟机部署的耗时。他们对写实复制技术进行了深入分析，提出了基于虚拟机镜像细粒度划分的部署方法，对镜像进行分割和回收，降低镜像冗余，减少网络传输数据量和计算节点存储占用空间。但是他们对于镜像的划分还不够细，写实复制技术依旧需要传输完整的包含操作系统内核的虚拟机镜像，这其中还有很多数据是不需要在引导阶段传输。

张钊宁[29-30]等对大规模虚拟资源的部署策略进行了研究，设计和实现了基于中继树形结构的“VMThunder”部署系统，他们对传统虚拟化存储技术进行了改进，增加了中间缓存层，当计算节点已缓存镜像数据之后，也可以作为镜像中继节点，为其他计算节点提供镜像服务。实现了 1 分钟启动 1000 台虚拟机的部署能力。他们使用的方式实际上就是远程引导，这样会产生大量零碎的网络 I/O 请求，这些请求会对镜像节点的磁盘带来大量的寻道负载。

刘圣卓[31]等分析了虚拟集群中镜像的存储结构、分发原理和性能，针对大规模复杂 I/O 密集的场景，设计了虚拟镜像按需分发框架，提出了面向多虚拟机的混合存储替换算法，并实现了基于副本的镜像迁移优化方法。他们使用 I/O 跟踪技术对虚拟集群的 I/O 原理进行了实验分析，对虚拟机启动和运行过程的 I/O 流程进行了探索。他们将 I/O 读写密集的区域定义为“热区”，并设计了热区跟踪替换算法，把具有出色随机读写性能的 SSD 作为缓存，将热区的数据放入 SSD 之中，提升普

通硬盘的 I/O 性能。通过对虚拟机启动过程的分析，他们将引导过程所必须的数据从磁盘中分离，划分为启动区和工作区，基于 QCOW2 文件系统重新设计存储结构，对镜像进行分组，减少系统引导所需要传输的数据量，使用 SSD 作为缓存，提高整体存储性能。设计 FlimCD 迁移算法，对镜像的脏块进行记录，对数据库进行优先级排序，提升数据中心任意主机之间虚拟实例迁移的效率。虽然他们真正实现将引导数据和启动数据分离，但是在远程系统中依旧需要在数据传输完成之后进行引导，不能实现操作系统的引导和传输同步进行。

1.2.2 基于数据传输的优化

Romain[33]等人针对 CERN 的需求，设计了一个高效可信的镜像分发方案，他们构建了一个叫做 Large Hadron Collider(LGC)的网络，一方面用于适应 CERN 的节点网络，另一方面便于使用一些优化协议。结合二进制树和 BitTorrent 协议相关技术，他们实现了在 10 分钟内，将 10G 的镜像分发给超过 400 个物理节点的性能。

陈彬等在基于虚拟机镜像细粒度划分的部署方法基础之上，提出了按需获取原则设置预取机制，设计对应的策略和算法，根据访问频率，动态设置数据的优先级和预取数据量，降低虚拟机部署时用户的等待时间，同时对网络传输进行优化，通过使用 P2P 协议，解决存储节点单点网络带宽不足的缺点，提高网络整体的利用率，降低镜像分发耗时。P2P 协议的引入可以缓解单点负载过高的问题，但是其传输的数据是零碎的，只有在所有数据都传输完成之后，得到的镜像才有意义。

张钊宁等使用计算节点已缓存镜像[34]数据，作为镜像中继节点。虚拟机的存储镜像通过挂载+缓存的方式从远程映射到本地，虚拟机启动所需数据直接从远程拽去，降低非引导必须数据在虚拟机启动过程对网络带宽的占用。在此基础上，它们设计了一种内存映射的快速部署方式，直接将模板虚拟实例的内存进行复制，生产较大规模的虚拟实例集群，并利用休眠技术进行恢复，跳过虚拟实例启动过程，直接进入工作状态。最后他们设计了虚拟化存储的预读优化方法，针对磁盘预读、跨网络预读和多层预读现象进行了优化，提升存储系统的性能。通过内存复制的方式可以移除虚拟机引导的带来的开销，但是会带来较大的数据传输量。

赵勋[35]等将 SDN[36]技术引入部署系统，使用规则增量更新的方法，以更快地构建和更新虚拟网络拓扑，降低网络延迟。C. LEE[37]等根据多个虚拟机镜像的的相似性，提取公共部分，降低镜像传输量。

1.2.3 基于部署方法的优化

赵勋等针对超大规模物理主机组成的计算集群，提出了以 I/O 特征为主要参考指标的虚拟集群动态构建方法，该方法将虚拟机 CPU、内存、网络等参考指标与 I/O 特性相结合，计算出热点预测值，并借助平滑预测算法，提升调度性能，降低传统调度算法带来的额外开销。

现有的云计算平台[38]也是一种基于部署方法的优化方案，平台以传统虚拟化技术为基础，进行统一的管理和配置，将复杂的配置工作，抽象为统一的接口，并交给平台完成。但是对于大规模虚拟网络的部署，这些云计算方案并不能够直接满足需要。一个最直接的问题就是部署速率。传统的这些云计算平台的方案对镜像优化重点往往是在后期的缓存和高可用性，对全新的镜像和网络拓扑部署并无特殊优化。以 Openstack[39]为例：

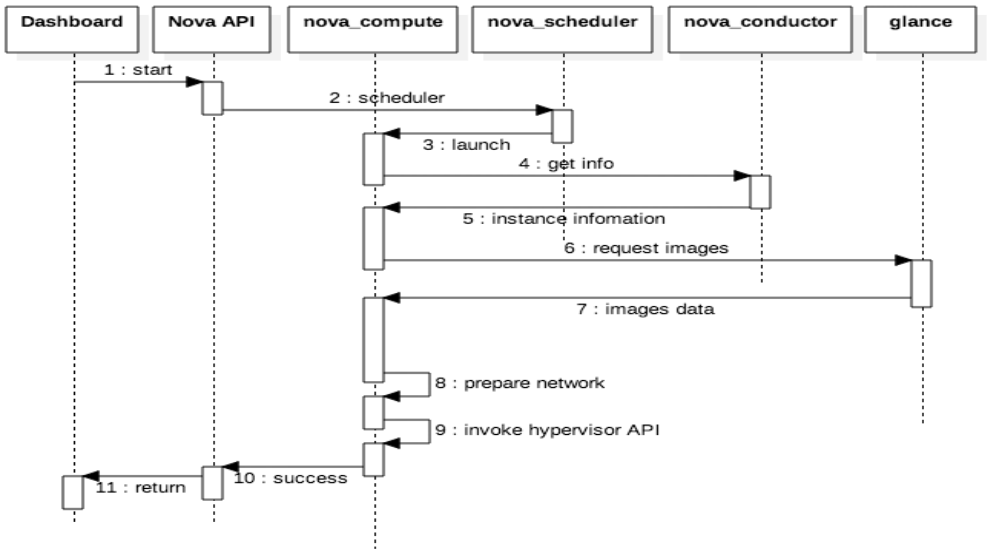

图 1-1 Openstack 实例部署流程

图 1-1 是 Openstack 平台的部署流程，启动一个镜像，首先由控制台发起，调用 Nova API，发送消息给 nova-scheduler 选择调度合适的计算节点，然后发送给消息给计算节点的 nova-compute 服务，让其准备启动实例，计算节点在收到部署请求后，向 nova-conductor 服务请求部署实例所需要的所有信息，然后向 glance 镜

像服务下载镜像数据，最后准备好网络和其他环境，完成引导。这个过程中对镜像获取的操作，只是简单的下载，并没有做过多的优化，当面临体积庞大的镜像传输请求时，会出现性能瓶颈。

1.3 本文工作

1.3.1 I/O 与文件系统研究

虚拟机的镜像以文件形式保存在宿主机的磁盘之上，虚拟机对镜像的读写都是通过宿主机的 I/O 请求完成的，要完成对虚拟机引导过程优化，需要对操作系统（Linux）的 I/O 流程进行分析，以获取操作系统引导所请求的数据。我们将对虚拟机发起的所有 I/O 请求进行监视并记录，由于操作系统分为很多层次，这个过程的实现可以从多个角度来做，包括虚拟机 Hypervisor 程序本身、用户空间基础库、系统调用、VFS（Virtual Filesystem Switch）[40]和文件系统驱动程序等。不同层次所对应的接口不同，对 I/O 的处理也是不一样的。本文将从系统调用和文件系统驱动层次进行研究，对镜像相关的 I/O 过程进行过滤，并在此基础上对流程进行优化，实现一个优化的文件系统过滤层。

1.3.2 虚拟机引导过程优化

传统虚拟化技术是将磁盘完整拷贝到计算节点之后，再进行引导，但虚拟机启动所需的数据占整个磁盘数据的比例很小，实现虚拟机快速引导的需求并不需要将整个磁盘数据都传输到计算节点，一种优化方案是直接使用如 NFS[41]、CIFS[42]等网络文件系统，将本地 I/O 请求映射到网络，保证只有需要的数据才通过网络传输，降低传输的数据量。但是虚拟机引导过程会产生大量的随机 I/O 请求，这些请求会对网络和远程存储节点带来较高的 I/O 负载，在较大规模的虚拟实例部署场景下性能并不出色。本文的优化方案是对操作系统引导的 I/O 流程进行分析，将引导产生的 I/O 请求数据进行记录和分离，在部署过程优先传输这部分磁盘数据到目标节点。在此技术之上，对操作系统引导流程的 I/O 请求进行分析和记录，按照先后顺序进行优先级排序，在远程操作系统引导时，以数据流的形式传输到目标系统，保证操作系统引导和数据传输同步进行，尽可能性降低等待时间，提高部署效率。

1.3.3 虚拟机与容器混合优化部署系统

单一的虚拟化方案在大规模虚拟网络部署的场景中会面临一些资源分配问题，本文将实现一个统一的虚拟化部署框架，融合不同的虚拟化技术，以适应不同的需求环境，课题以容器为例，与传统虚拟机技术进行整合。

容器技术是一种轻量级虚拟化方案，通过共享操作系统内核，隔离进程空间和网络命名空间，在宿主操作系统创造了一个独立高效的运行环境。容器的本质是系统进程，其性能和直接运行在宿主机的程序几乎没有差别，并且构建与销毁的过程开销较小，灵活高效。通过共享系统内核、共享基础库、使用层级文件系统等方式去除冗余数据，降低存储和网络传输开销。在大规模虚拟网络部署场景中，很对模拟任务虚拟实例（如模拟大规模简单的用户主机、生成背景流量）具有任务简单但规模庞大的特点，使用传统虚拟机方式会造成大量的资源浪费和性能损耗，如果把这些实例以容器的方式实现，可以克服以上弊端，提高资源的利用率。现有的云计算方案大多将传统虚拟化技术与容器技术分离开来，在上层部署逻辑看来，不能进行无差别地当做普通虚拟实例使用。本文将设计一个虚拟机与容器混合的部署系统，为上层应用提供统一的接口，同时设计优化的存储、网络传输优化方案，以适应大规模虚拟网络部署场景的需求。

第 2 章　虚拟机镜像优化部署策略研究

2.1 研究内容

在虚拟网络部署场景中，往往会对底层的 hypervisor 接口进行装，以完成复杂的部署任务，同时提高稳定性。其中对镜像的操作也是由上层管理系统提供，包括镜像的分发和存储。本文将对现有的镜像管理机制进行分析，并在其基础上进行优化，通过 I/O 分析、写实复制以及索引缓存的方法，引入基于网络流的虚拟机快速引导方式，以提高镜像部署效率，适应虚拟网络快速部署场景的需求。

2.1.1 虚拟机引导过程

在很多虚拟化场景中，我们对虚拟实例部署的一个重要指标是虚拟实例从分发到进入工作状态的耗时。一般来说，部署的过程分为两个阶段：镜像数据从存储节点传输到计算节点，计算节点使用镜像作为虚拟实例的存储磁盘并启动虚拟实例。

第一个阶段所做的工作主要是网络传输，其性能受镜像大小和网络带宽影响。第二阶段相当于本地引导一个虚拟机，其效率跟 hypervisor 的性能和节点的物理性能有关。传统部署方案中，这两个过程往往是分离的。当第一个阶段没有完成时，第二个阶段只能等待；第二个阶段完成之前，部署任务不能被释放。这样会造成大量的计算资源闲置，影响部署任务的效率。

2.1.2 优化原理

一方面操作系统的启动实际上并不需要镜像的完整数据，往往只需要读取镜像中极少的一部分，也就是说完成实例的引导并不需要将整个镜像的数据完全传输过去，另一方面，操作系统的引导和数据的传输是可以相对同步进行的，并不需要数据传输完成再开始引导。我们将从以下几个方向进行优化：

（1）分析操作系统引导 I/O 流程，以获取数据读取的分布和优先级，并提取操作系统引导所必须数据。

（2）使用写时重定向、索引和缓存技术重构镜像节点，将计算节点作为存储节点缓存，提高系统网络的利用率，并降低大规模虚拟网络部署时的 I/O 代价。

（3）设计基于网络流镜像传输的引导方法，实现低 I/O 开销的远程快速引导方案。

2.2 虚拟机优化部署概述

本文的研究内容以存储优化分发子系统的形式，融合进虚拟网络快速部署系统，本章节将结合子系统进行说明。

2.2.1 系统结构

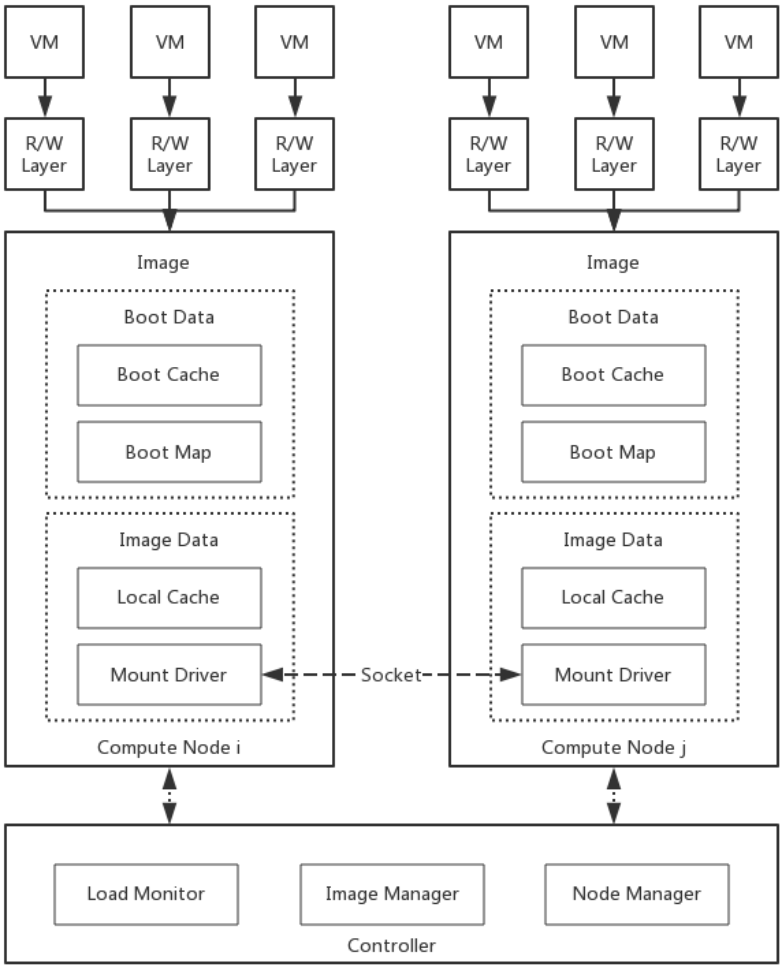

图 2-1 系统结构

图 2-1 为优化子系统结构，包含计算节点(Compute Node)和控制节点(Controller)，控制节点包含大量对各个节点上下文管理的 Node Manager 模块、镜像信息管理的 Image Manager 模块和用于监视各个计算节点负载的 Load Monitor。计算节点包含了虚拟机 Hypervisor、虚拟机镜像读写层(R/W Layer)、镜像设备(Image)，其中镜像实体包含镜像数据缓存及对于文件系统驱动(Image Data)、操作系统引导所必须数据及映射表(Boot Data)，总体步骤如下：

（1）部署开始，计算节点从其他合适的计算节点挂载镜像，并启动镜像传输，优先传输 Boot Cache，其中包含了镜像中虚拟机操作系统引导需要的数据，再传输镜像中其他部分数据。

（2）创建虚拟机实例和对应的读写层(R/W Layer)，读写层链接到镜像设备，启动虚拟实例。镜像设备对虚拟机实例只读，多个虚拟机实例共享同一个镜像实体，虚拟机写入的数据保存在读写层。Boot Cache 中的数据是按照虚拟机实例启动读取的顺序传输，虚拟机启动过程和数据传输过程同步进行。

（3）当镜像数据传输完成之后，计算节点向控制节点注册镜像缓存服务，为其他计算节点提供缓存服务。

2.2.2 关键问题

在对虚拟机部署系统进行优化的过程中，以下问题需要解决：

（1）引导必须数据预测及排序：虚拟机原始镜像包含了操作内核、标准库及应用程序和脚本，虚拟机启动所必须的数据只占很小的部分，系统需要将这部分数据提取出来。除此之外，为了完成操作系统和数据传输同步进行，这些引导所必须的数据需要按照操作系统引导顺序排序。

（2）文件系统过滤：虚拟机对镜像数据的访问都是以文件的形式，不同虚拟机 Hypervisor 对镜像的存储格式支持各不相同，常见的包括 QCOW2、VDI 和 VMDK 等格式，系统需要开发一个文件系统过滤层，兼容以上各种类型文件格式。系统对数据处理逻辑对虚拟机透明，在虚拟机看来，和操作正常文件/目录并无区别。

（3）数据映射：镜像实体中包含 Boot Data 与 Image Data，其中 Boot Cache 将由 Boot Map 映射到 Local Cache 中，同时需要对 Local Cache 的数据块进行跟

踪，以确定是否已从远处获取。Local Cache 的数据只读，虚拟机通过 R/W Layer 读取 Local Cache 中的数据，R/W Layer 中需要记录写入的数据。

（4）并发处理：同一个计算节点之上，会保存多个镜像，每个镜像会同时被多个虚拟机使用，当计算节点的 Local Cache 缓存完成后，会作为镜像缓存节点提供给其他计算节点访问，其访问过程也是并发的，所以需要考虑这些过程的并发问题。

2.2.3 部署流程

图 2-2 描述了虚拟机镜像优化部署的流程：

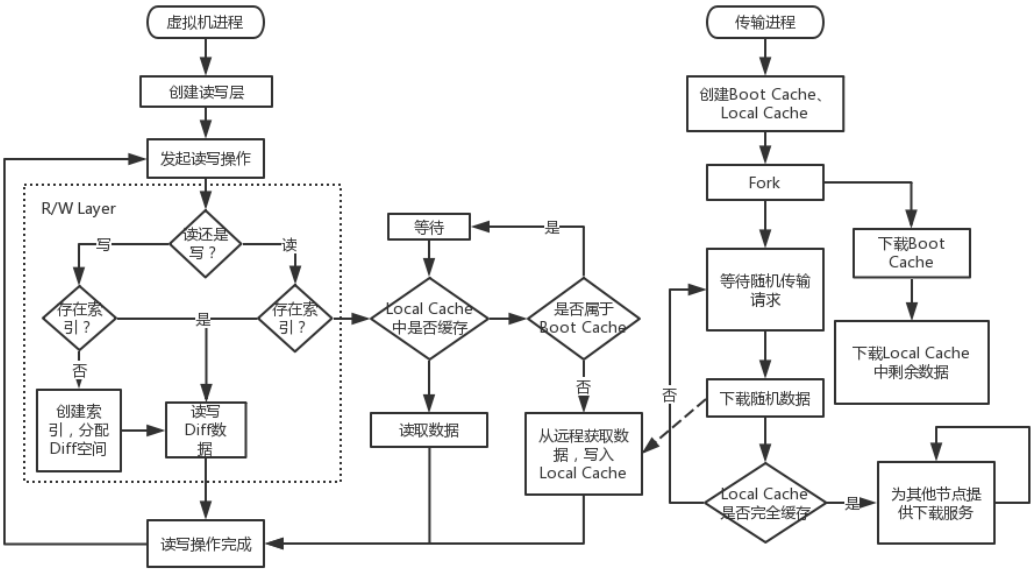

图 2-2 部署流程

虚拟机进程和镜像传输进程同步进行。传输进程包括以下步骤：

（1）创建 Boot Cache、Local Cache 等数据结构，初始化驱动设备和网络套接字。

（2）创建新进程，用于从远程镜像服务器下载 Boot Cache，并将数据放入 Local Cache 中，之后下载 Local Cache 中剩余数据，下载完成后通知父进程，进程退出。

（3）监听虚拟机进程的请求，以从远程下载虚拟机操作系统引导过程中对 Boot Cache 之外的数据。

（4）循环检查镜像数据是否下载完成，当数据下载完成之后，进程开始为其他计算节点提供镜像下载服务。

虚拟机进程包括如下步骤：

（1）创建读写层(R/W Layer，数据结构将在后续章节说明)，读写层链接到部署文件系统中的镜像(Image)。

（2）发起 I/O 请求，请求会首先发送到读写层处理逻辑，读写层使用索引和缓存等技术进行增量存储，虚拟机写入的数据会保存在读写层中，读取数据时，如果读写层中没有所请求的数据，需要将请求转发到其下一层磁盘（系统仅使用两层），也就是系统实现的文件系统。

（3）文件系统接收到请求之后，会检查请求的数据是否在本地缓存（Local Cache 的位图是否置位），如果本地不存在所请求的数据，就需要从远程下载。由于 Boot Cache 通过传输进程不断地下载到本地，如果请求的数据属于 Boot Cache，说明传输的速度跟不上启动的速度，这时候需要进行等待，以降低网络 I/O 次数，直到所需数据下载完成。如果请求的数据不属于 Boot Cache，说明操作系统已经不处于引导阶段，这时候需要虚拟机进程直接从远程下载所需数据。

（4）完成本次读写操作，跳转到（2）进行下次读写。

2.3 引导数据提取

2.3.1 目标数据

提取引导必须数据本质上是减少镜像数据传输量，传统方案中也使用了类似的方法，例如通过裁剪内核和基础库的方式，减小镜像体积，但是由于部分数据不可分割和粒度划分不够细的原因，还是传输了不少多余的数据。本文的方案将以 I/O 请求粒度，记录操作系统启动过程中的所有读请求，所有请求的并集就是引导所需要的数据。设镜像文件的总大小为 S $(S \in N^*)$，操作系统引导过程对镜像文件的读请求数总为 C $(C \in N^*)$，第 i $(i \in N^*, 1 \leq i \leq C)$ 次请求的偏移为 O_i $(O_i \in N^*, 0 \leq O_i < S)$，长度为 L_i $(L_i \in N^*, 0 < L_i \leq S)$，则引导所需数据在镜像文件中所占区间 D 为：

$$D = \cup_{i \in C} Q_i$$

$$其中Q_{\mathrm{i}} = \{x \in N^* \mid O_i \leq x < O_i + L_i\}$$

为了提高读写效率，在系统实现时，会将请求数据偏移按照 4k(4096)字节对齐。

2.3.2 系统调用

为了获取这些数据，需要对操作系统引导过程对镜像的 I/O 操作进行监控（系统的宿主机使用 Linux 系统，非特别注明，系统环境默认为 Linux），使用 Hook 相关接口的方式来实现，Linux 中应用程序对镜像数据读写的接口调用层次如下：

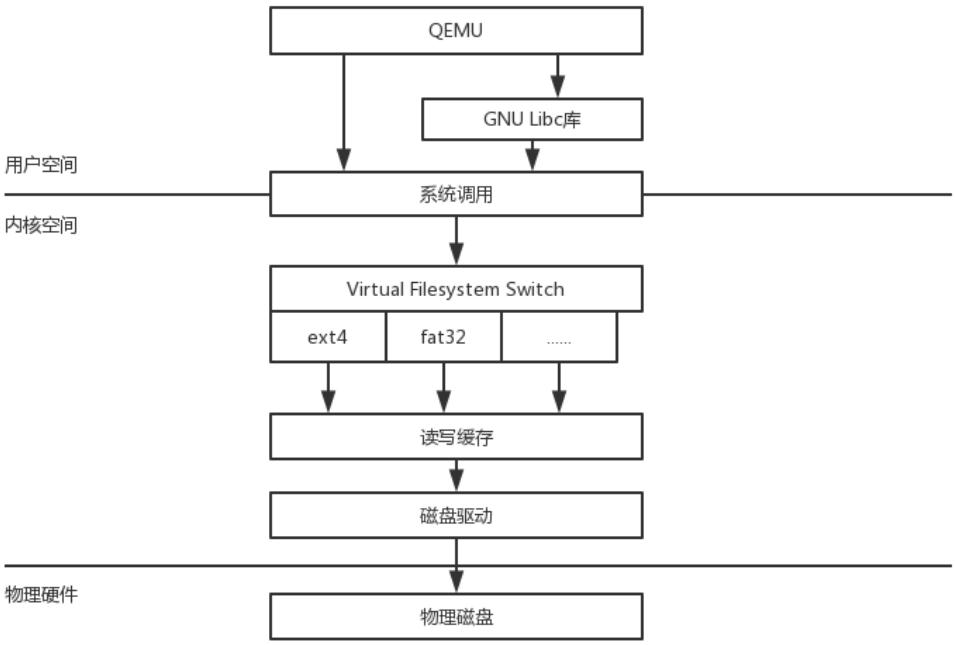

图 2-3 文件系统

这里使用 QEMU 作为 Hypervisor 来跟踪客户操作系统的 I/O，如图 2-3 所示，当 QEMU 需要访问镜像文件数据时，会直接或者间接(通过 GLibc 库)调用系统调用进入内核，然后调用 VFS(Virtual Filesystem Switch)中对应的接口(例如 read 系统调用对应的接口是 vfs_read)将请求分发到镜像文件所在的文件系统，由文件系统去操作读写缓存和磁盘驱动，最后由磁盘驱动程序实现对磁盘数据的读写。对文件操作相关的 Hook 可以挂在以下三个层面：

（1）Hypervisor(QEMU)

（2）系统调用

（3）系统内核

因为虚拟机对磁盘镜像的操作都是通过 Hypervisor 做的，对 Hypervisor 进行 Hook 是最直接监控 I/O 的方式，通过修改 Hypervisor 的源码，对每一个对镜像操作的函数都做一层记录。但这样做会造成一些问题，一方面因为 Hypervisor 对磁盘操作的函数较多，而且不一定在同一个模块，很容易遗漏，造成数据不准确，另一方面引入的代码可能会带来一些 bug，解决这些问题的代价较大，偏离了本文的研究重点。对系统内核进行 Hook 需要处理内核中复杂的数据结构相关关系，而且在内核中很多操作是不如应用层方便，例如对文件的读写、调试等操作。这里我们对系调用层进行 Hook。

Linux 中系统调用是用户态程序与操作系统通信的接口，用户态程序在执行特权级操作时，需要使用系统调用提高权限。用户态进程对文件系统的操作都是要通过系统调用进行的，通过 Hook 系统调用，能够过滤所有跟文件系统相关的操作，包括 open、read 和 write 等：

表 2-1 文件系统读写相关部分系统调用（x86_x64）

调用号	系统调用名	描述	实现文件
0	read	读取数据	fs/read_write.c
1	write	写入数据	fs/read_write.c
2	open	打开设备	fs/open.c
3	close	关闭设备	fs/open.c
19	readv	读取数据到多个缓冲区	fs/read_write.c
20	writev	写入多个缓冲区的数据	fs/read_write.c

系统调用地址存储在系统内核空间的系统调用表(sys_call_table)中，系统调用号(表格中 rax)作为索引，一种 Hook 系统调用的方式是修改系统调用表，将表中目标调用的地址改为 Hook 函数的地址，系统调用表的地址在内核编译后会导出，通常保存在/boot/System.map-<kernel verison>中：

```
# root @ user in ~ [2:42:44]
$ cd /boot

# root @ user in /boot [2:42:46]
$ cat System.map-4.4.0-62-generic | grep sys_call
ffffffff81287d50 t proc_sys_call_handler
ffffffff8183864d T int_ret_from_sys_call_irqs_off
ffffffff8183872b T int_ret_from_sys_call
ffffffff81a001c0 R sys_call_table
ffffffff81a01500 R ia32_sys_call_table
```

图 2-4 系统调用表地址

其中 0xffffffff81a001c0 就是系统调用表的地址。这种 Hook 方式是全局有效的，也就是能够过滤所有应用层程序的系统调用，但对于我们的需求来说是不需要的，所以考虑在应用层过滤，使用动态库注入的方式进行 Hook。

用户态程序在使用系统调用时，一般会使用系统基础库(libc)封装后的接口，这些接口的地址在应用程序运行时动态加载到应用程序内存空间中，Hook 这些接口的方式就是修改应用程序导入表地址。应用程序在导入外部符号时是按加载库的顺序导入的，如果同一个符号出现在多个将被加载的动态库中，第一个加载的动态库中的符号将被应用程序导入。系统通过在一个自定义动态库中实现 read、write 等文件操作相关的函数，并让其优先被 Hypervisor 加载，以保证 Hypervisor 对文件读写的系统调用都是调用系统实现的动态库中相应函数。为了保证动态库被优先加载，只需要在虚拟机运行之前设置 LD_PRELOAD 环境变量为动态库文件路径。

2.3.3 数据获取

为了获取指定镜像的 Boot Cache，在 Hook 函数中，我们分析读取文件数据的相关操作的参数和返回值，并记录读取位置偏移，读取长度和时间戳，得到记录集合 R，每条记录由(O, L, T)元组构成，O 为偏移，L 为长度，T 为时间戳。Boot Cache 的结构由一条一条记录构成，每一条记录如下：

图 2-5 Boot Cache 记录结构

Base 表示该记录的数据在原镜像中的偏移，Length 表示该记录的数据长度，Data 部分是数据内容，其中 Base 和 Length 都是以 4kb 为单位。然后按照以下算法生成 Boot Cache：

算法 2-1 Boot Cache 生成算法

算法 2-1 Boot Cache 生成算法

```
输入:  镜像文件 I，记录数据 R
输出:  Boot Cache 数据 bc
 1:    I_map = { 0 } * (1 + len(I) >> 12) //I_map 为镜像文件的位图
 2:    bc = {0}
 3:    for O, L, T in R:
 4:      L += O % 0x1000
 5:      O = O >> 12
 6:      L = L % 0x1000 == 0 ? L >> 12 : (L >> 12)+1 //4K 对齐
 7:      Base = O; Length = 0
 8:      while L > 0:
 9:        if I_map[O + Length] == 1:
10:          if Length != 0:
11:            bc += {Base, Length, I[Base << 12, (Base << 12) + (Length
       << 3)]}
12:          Base = ++O; Length = 0
13:        else:
14:          Length++
15:        I_map[O + Length] = 1
16:        L--
17:      if Length > 0:
18:        bc += {Base, Length, I[Base << 12, (Base << 12) + (Length <<
       12)]}
19:    return bc
```

算法的输入为镜像文件 *I* 和操作系统启动过程的 I/O 记录 *R*，算法首先生成镜像文件的位图 I_map，每一位对应于镜像中每 4K 单位的数据块，并初始化为 0，返回值 bc(Boot Cache)初始化为空。对于每一条记录，将记录中的偏移、长度按照 4kb 对齐，然后从记录中 Base 开始，依次往后按 4kb 数据块扫描，如果 I_map 为 0(当前数据块没有被记录)，记录当前数据块，并继续扫描，否则，写入已扫描的数据块，并创建新的 bc 记录和对应的 Base 与 Length。检测是否有未写入的数据，并进行写入，然后进入下一个循环，处理下一条记录。最后返回 Boot Cache 的数据。设操作系统引导记录的规模为 N，磁盘镜像的规模为 M，算

法对每条记录进行处理，每条记录最多包含 M 个数据块，所以算法的时间复杂度为 $O(M*N)$，空间复杂度为 $O(M)$。

因为操作系统引导过程可能多次读取同一个数据块的数据，所以算法使用 I_map 位图结构来进行跟踪，防止重复记录相同数据块。通过以上过程，我们能够获取到引导所必须的数据，在虚拟机部署的过程中，只需要将这部分是数据传送到计算节点，虚拟机就可以完成启动。

2.4 镜像格式优化

2.4.1 Local Cache

虚拟机在启动之前，其数据为远程镜像文件的映射，并作为启动镜像参数传递给上层虚拟机。在部署开始之前，Local Cache 的内容为空，随着虚拟机的启动，Boot Cache 的数据首先被传输过来，写入到 Local Cache 中，然后是镜像剩余的数据。

系统使用位图来跟踪 Local Cache 的数据缓存状态，以确定本地是否已缓存远程镜像数据。镜像文件将被划分为以 4KB 为单位的数据块，对应位图的每一 bit 位。对于一个 1GB 大小的镜像文件，其对应的位图的大小为 32KB。Local Cache 每个数据块对应位图比特位为 0 表示该块数据本地缺失，需要从远程拽去，而 1 表示已经完成缓存。之所以选择 4KB 作为单位数据块，是为了和主流处理器架构分页机制对齐，提高性能。当虚拟机从 Local Cache 读取数据时，会通过检测位图是否置位选择是否从远程镜像服务器下载数据。每次新的数据传入之后，对应位置的位图会被置 1。

Local Cache 将作为本地的镜像数据缓存，不能直接被虚拟机使用，因为虚拟机在运行过程中，会对镜像进行写入操作，如果 Local Cache 被写入数据，原始镜像的数据就会被污染，造成数据不一致。系统仅允许虚拟机对 Local Cache 的进行读取操作，并在上层对每个虚拟实例构建独立的 R/W Layer 数据层，作为虚拟机直接操作的镜像文件。

2.4.2 Boot Cache

通过引导数据的提取，我们获取了 Boot Cache。对操作系统必要数据预测的优化方式本质上是降低传输数据量，虽然能够将传输数据量降低到 10%以下，但依旧没有改变系统引导需要先将镜像数据传输到计算节点，再进行引导的流程，在引导必须数据传输到计算节点之前，操作系统并不能引导。

一种更直接的优化方案是进行远程引导，将镜像服务器上的镜像文件直接挂在到本地，然后像操作本地镜像文件一样直接进行引导。比较成熟的方案例如 NFS 和 CIFS。虽然这种方式能够将传输数据量降到理论最低值，但也会带来额外的 I/O 负担。操作系统引导所需要的数据是零碎地分布在镜像文件中，每次启动，系统会发起成千上万大小不等的 I/O 请求，对于每个 I/O 请求，都需要完成一次网络数据通信，对应的，也需要远程镜像服务器完成一次磁盘寻道。在大规模虚拟网络部署环境中，这种大量零碎的 I/O 会给镜像节点带来很大负担，严重时会造成网络线速还有富余，I/O 已经到了处理能力上限的情况。

这个问题出现的原因在于数据存储的非连续性，系统考虑将这些数据按数据排列（也就是 Boot Cache），图 2-6 显示了我们构建 Boot Cache 的过程：

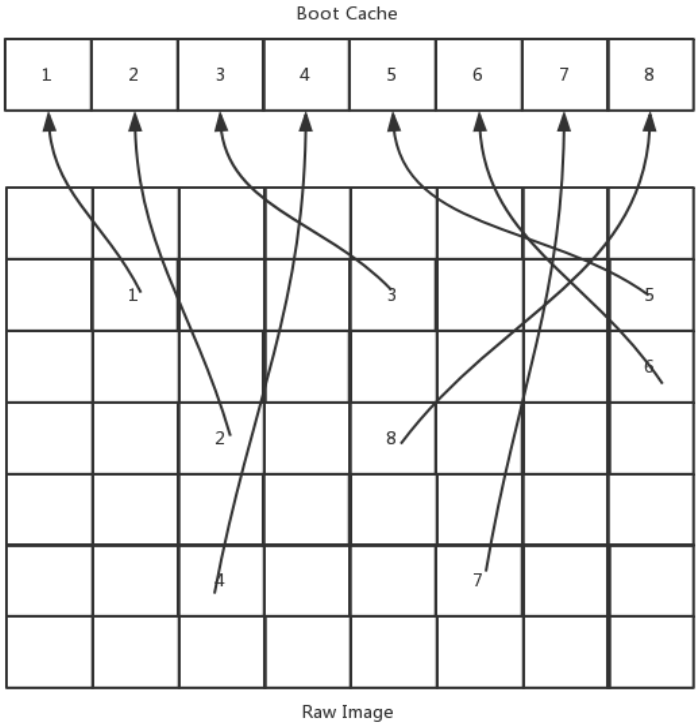

图 2-6 Boot Cache 结构

Boot Cache 映射了原有的虚拟磁盘镜像，将操作系统引导过程中对磁盘读取的数据块按照先后顺序重新排列，同时将映射关系保存到附属的 Boot Map 数据结构中。

计算节点获取 Boot Cache 数据的过程和在网络文件系统引导虚拟机获取数据的顺序保持一致。也就是说，虚拟机的引导过程和数据传输顺序保持一致，在这种情况下，镜像传输和虚拟机启动可以同步进行，Boot Cache 之外后续的数据将通过常规方式以较低的优先级传输。

2.4.3 R/W Layer

Local Cache 将作为原始镜像数据供所有虚拟机共享使用，其内容对虚拟机来说具有只读属性，虚拟机写入的数据会被保存在读写层(R/W Layer)中。读写层是虚拟机视角的镜像文件，每一个虚拟机都有一个对应的读写层，避免虚拟机对原始镜像数据的直接写入，其结构如下：

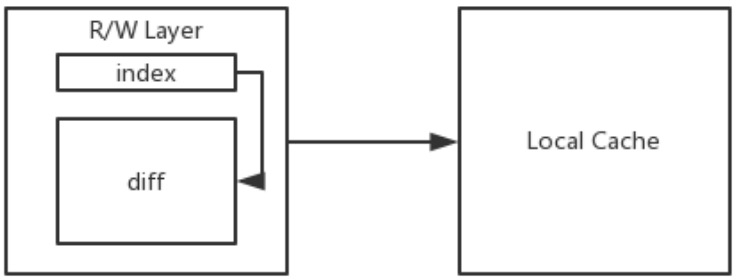

图 2-7 R/W Layer

R/W Layer 使用了写时重定向技术，其数据结构由索引(index)和缓存(diff)组成，diff 是用于保存数据的缓存，index 是一个索引列表，指向 diff 中的数据。虚拟机对 R/W Layer 进行读写时，文件系统首先会检查 index 中是否存在所操作的数据，如果存在，则对 index 指向的数据进行操作。index 的初始状态为空，所有的读请求将从 Local Cache 获取数据，当虚拟机对镜像有写操作时，R/W Layer 会在 index 记录写入操作，创建索引，并将数据保存在 diff 中。R/W Layer 只有在写入新的数据时候会增加 index 的记录，所以在部署初期每个虚拟实例对磁盘造成的额外负担会很小，Local Cache 则被所有基于该镜像运行的虚拟实例共享。

当 Local Cache 缓存完成后，当前的计算节点也会作为该镜像的存储节点，为其他计算节点提供服务，分担原镜像服务器的负担。

2.5 文件系统过滤

系统对镜像的优化均基于对文件的操作，需要在文件系统层面进行实现。一种方式是实现全新的文件系统，这种方式拥有最大的灵活性，但是需要处理文件系统复杂的底层细节。由于本系统并不关心文件在物理磁盘的存储形式，所以选择通过过滤文件系统请求来实现需求。

2.5.1 文件系统过滤层

图 2-8 为文件系统过滤层跟虚拟机和 Linux VFS 的相互关系：

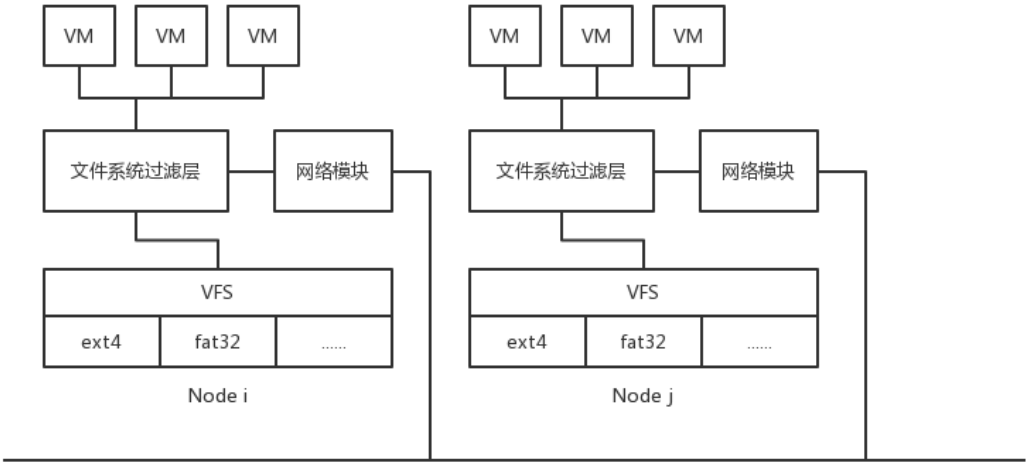

图 2-8 文件系统过滤层

虚拟机(VM)对文件系统的操作会首先经过文件系统过滤层（也就是本文实现的系统）处理，文件系统过滤层会根据需要，将数据保存在实际的文件系统中或者通过网络模块从远程获取数据。

文件系统过滤层可以放在内核中（设备驱动），或者在应用层（Filesystem in Userspace，FUSE）。FUSE 让用户以非特权的身份，不依赖内核代码，在 UNIX 类系统中构建自身文件系统的软件接口。系统基于 FUSE 文件系统实现文件系统过滤层。

2.5.2 接口

图 2-9 为 FUSE 的文件操作方法的回调接口：

```
struct fuse_operations {
    int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);
    int (*mkdir) (const char *, mode_t);
    int (*unlink) (const char *);
    int (*rmdir) (const char *);
    int (*chmod) (const char *, mode_t);
    int (*chown) (const char *, uid_t, gid_t);
    int (*truncate) (const char *, off_t);
    int (*utime) (const char *, struct utimbuf *);
    int (*open) (const char *, struct fuse_file_info *);
    int (*read) (const char *, char *, size_t, off_t,
            struct fuse_file_info *);
    int (*write) (const char *, const char *, size_t, off_t,
            struct fuse_file_info *);
    int (*statfs) (const char *, struct statvfs *);
    int (*flush) (const char *, struct fuse_file_info *);
    int (*release) (const char *, struct fuse_file_info *);
    int (*fsync) (const char *, int, struct fuse_file_info *);
    int (*listxattr) (const char *, char *, size_t);
    int (*removexattr) (const char *, const char *);
    int (*opendir) (const char *, struct fuse_file_info *);
    int (*readdir) (const char *, void *, fuse_fill_dir_t, off_t,
            struct fuse_file_info *);
    ........
};
```

图 2-9 文件操作相关部分接口

当上层应用程序对系统自定义的 FUSE 文件系统进行操作时候，会调用以上接口，系统将重写部分需要的接口，以实现需求。

在文件系统创建之时，会初始化一些包括远程镜像服务节点的套接字等数据，这些数据会在之后文件读写的过程中使用。这些数据共享给所有虚拟机进程，所以系统将其保存在一个全局的数据结构中，这个数据结构的入口地址可以在 FUSE 初始化时候传递进去。

2.5.3 初始化

在文件系统初始化时，首先需要从控制节点获取镜像列表及元数据，在本地创建镜像的缓存文件(Local Cache)和对于的位图，缓存文件的创建由下层文件系统完成(ext4、fat32 等格式)，当前的缓存文件为空，仅用作占位。然后，系统从控制节点获取用于下载数据的远程镜像服务节点，镜像服务节点的选择过程根据其负载决定，包括 CPU、磁盘、内存和网络的负载状态。连接上远程镜像服务节点，保存到全局数据结构中。创建一个互斥锁，用于处理并发读写请求。创建传输进程，依次传输 Boot Cache 和 Local Cache。

2.5.4 主要回调函数实现

在上层应用读写文件系统时，会调用对应的回调函数，其中主要的两个函数（open 和 read）实现如下：

（1）int (*open) (const char *path, struct fuse_file_info *fi);

open 回调将在虚拟机打开镜像文件时候调用，其中第一个参数为打开文件的路径，这里需要替换到镜像文件真实保存的位置（这个位置对虚拟机来说不可见），第二个参数为 fuse 文件系统的上下文信息，用于跟踪打开情况。

系统在收到这个请求后，根据第一个参数的路径，调用原始系统调用打开对应的文件，并将返回的文件描述符保存的 fuse_file_info 结构中。这个结构也将在 read 和 write 调用中作为参数传递进去，

（2）int (*read) (const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi);

read 回调是虚拟机读取镜像数据的接口，同样第一个参数是文件路径，用于判断是哪个镜像，buf 是用于存放读取数据的缓冲区，size 是读取数据的长度，offset 是文件偏移，最后是 fuse_file_info 结构，保存了 open 时候打开的文件描述符。

调用 read 时候首先要判断本地缓存中是否存在请求的数据，如果数据已经缓存完毕（检测全局数据是否缓存完的全局标识位，会在 Local Cache 缓存完毕置位），这一步就会跳过，直接调用 read 系统调用，将数据读取并返回。如果不

是，这里会将 size 和 offset 按照 4kb 对齐，并根据 Local Cache 的数据位图测试 Local Cache 中是否存在请求的数据（由 size 和 offset 生成期望的位图数据段，跟 Local Cache 位图对应位置的内存进行比较），如果有，则调用系统调用读取对应数据，并返回；否则将根据 Boot Map 判断请求的数据是否在 Boot Cache 中，如果在，稍作等待后再次测试是否在 Local Cache 中，并进行下一步处理。如果请求的数据也不属于 Boot Cache，则需要调用文件系统初始化时候保存的远程服务器信息，进行通信，以获取数据，写入到 Local Cache 中，然后返回数据。

2.5.5 并发同步

虚拟机对于 Local Cache 的访问时并发的，在并发写入的时候可能会触发访问冲突，所以需要处理同步问题。系统会在如下情况下触发对 Local Cache 的写入：

（1）传输线程先后获取 Boot Cache 和 Local Cache 的数据。

（2）虚拟机在传输完成之前请求 Boot Cache 以外的数据。

在 Boot Cache 传输完之前，（1）写入的数据都属于 Boot Cache 部分，而（2）写入的数据则是不属于 Boot Cache 的部分，所以这个阶段是不会发生写入冲突的情况。需要处理的是（1）在传输剩余 Local Cache 数据的阶段，这里使用一把互斥锁实现同步，写入 Local Cache 的步骤如下：

（1）等待获取锁

（2）将数据写入到 Local Cache 中

（3）将位图对应位置位

（4）释放锁

这里会有存在一个冗余问题，因为在等待获取锁的时候，即将写入的数据和刚释放锁的进程写入的数据存在重叠的情况，会有一个重复写入的过程。如果要去重，可在第（2）步前对位图进行判断，以跳过以写入数据。但在实际测试中，重叠数据的写入并不影响数据的正确性，而两种方式性能差别不大，为了简化流程，本文选择直接覆盖。

2.6 本章小结

本章介绍分析了虚拟机实例中操作系统的流程，介绍了传统虚拟网络部署过程中出现的问题，针对系统启动的两个阶段，提出了基于网络流的优化部署方案，让虚拟机镜像传输和操作系统引导同步进行，同时统计了启动过程中 I/O 请求的数据分布，在此基础上准确获取引导过程所必须的数据量，以减少镜像的数据传输量。通过分析 Linux 系统调用和文件系统，实现了基于 FUSE 的文件系统过滤层。设计了适用于快速引导的镜像存储结构，介绍了基于网络流的镜像传输流程，借助缓存和写实重定向技术，实现基础镜像的复用。

第 3 章　虚拟机容器混合优化部署系统

在传统虚拟机部署方案的基础之上，本文借助容器技术，提高资源的利用率，并实现虚拟机和容器的网络互连，构造一个混合的部署系统。

3.1 系统结构

3.1.1 系统简介

系统实现了一个统一部署框架，以虚拟机和容器为虚拟化实现，总体结构如下：

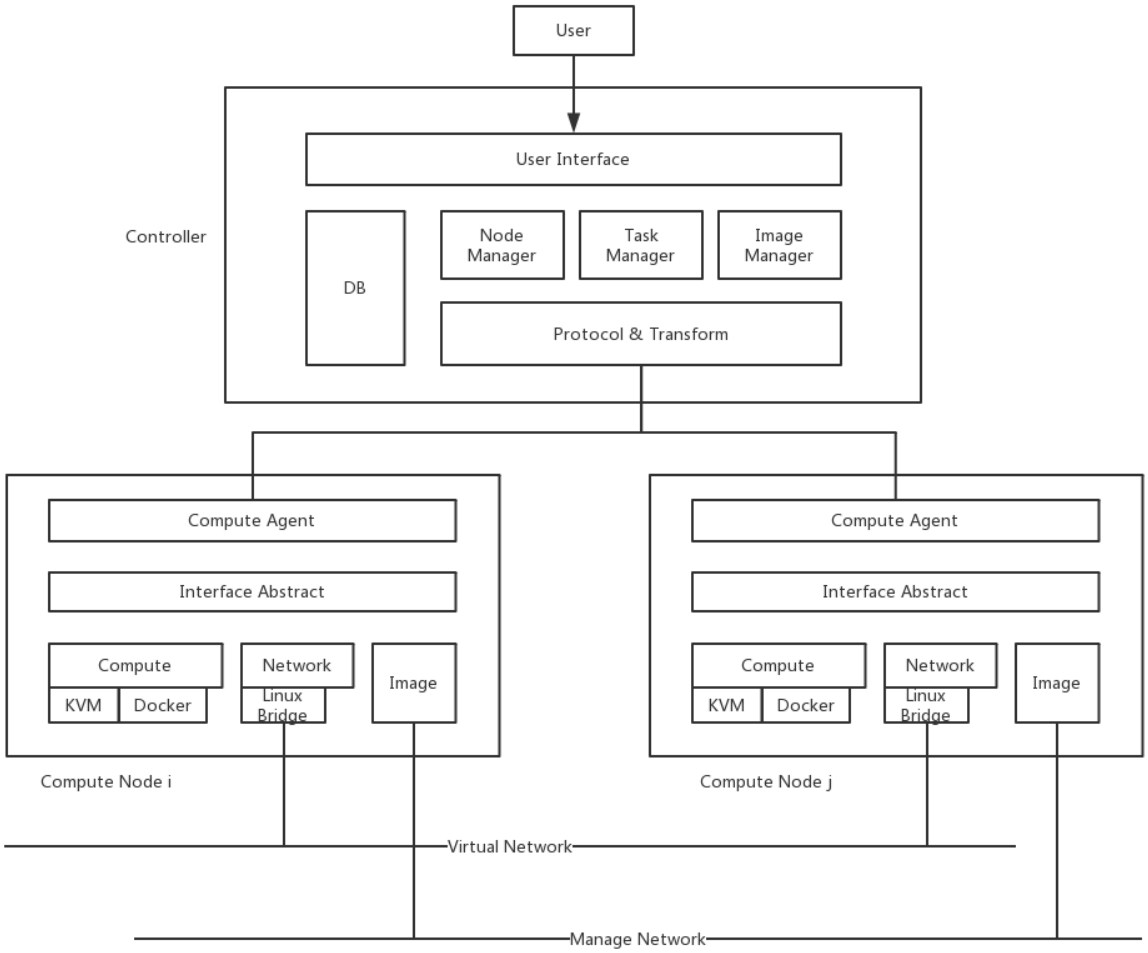

图 3-1 混合部署系统框架

图 3-1 为系统的总体框架图，由控制节点（Controller）和计算节点（Compute）组成。

控制节点包含了用户接口（User Interface）、数据库（DB）、节点管理（Node Manager），任务管理（Task Manager）、镜像管理（Image Manager）和网络传输（Protocol & Transform）模块。用户接口用于为用户提供友好的操作界面，并将用户指令转换为程序逻辑。节点管理模块用户维护计算节点的上下文，包含了节点的基本信息。任务模块维护所有的部署任务，提供分发部署任务和终止指令的接口。镜像管理模块用于维护镜像的元数据，同时监视了各个计算节点的运行负载状态，在虚拟机读取镜像数据时，调度模块会返回一个优化的传输方案。传输模块实现了一套控制协议，用于与各个计算节点通信。

计算节点包含了节点代理（Compute Agent），接口抽象（Interface Abstract）、计算（Compute）、网络（Network）和镜像（Image）模块。其中接口代理实现了网络和系统协议，用于跟控制节点通信。接口抽象模块用于为控制节点提供统一的控制接口，隐藏底层实现。计算模块用于实现并封装 Hypervisor 接口，包括 Libvirt(KVM)、Docker 等。网络模块用于封装网络接口，包括 Linux Bridge、Open vSwitch 等，各节点的网络互联互通，构成系统的网络。镜像模块为第二章优化部署方案的实现，用于虚拟机的镜像优化传输。

在大规模虚拟网络部署场景中，有很多类似于用户行为模拟的虚拟实例，这些实例往往数量众多、功能单一、资源需求少，对于这些实例，一个完整的最小虚拟化实例所提供的资源已经超过了需求，这样就造成了浪费。这里将使用容器替代这些实例，以提高资源的利用率。

3.1.2 总体流程

一个虚拟网络的部署过程分为三个阶段：任务规划、资源分配和实例运行。任务规划阶段是将抽象的网络拓扑映射到实际的物理主机，系统假设这部分工作用户已经完成，在用户使用本系统时，需要将任务规划的输出将作为系统的输入，系统将准备好虚拟化资源，并运行实例。总体流程如下：

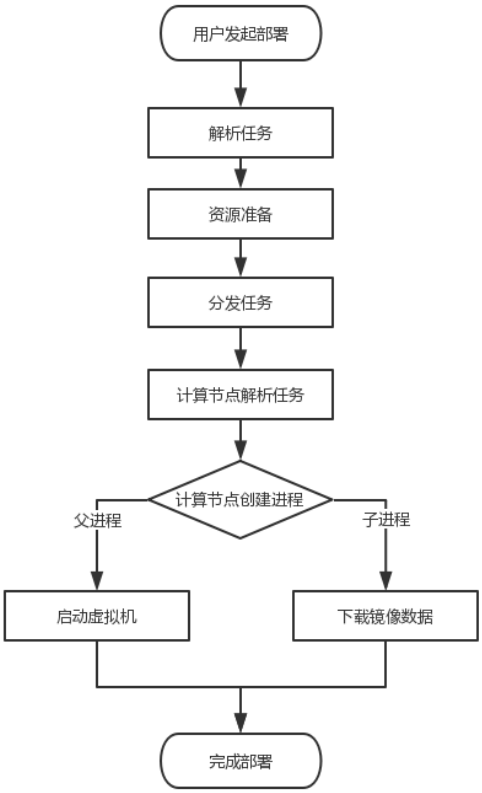

图 3-2 总体流程

首先由用户发起部署，将任务规划通过用户接口，发送到控制节点。控制节点在收到任务规划之后，将任务规划解析，准备好资源，然后将任务按系统协议封装，通过网络模块发送到各个计算节点，对每一个计算节点，在接受到部署任务的数据包之后进行解析，准备好虚拟机的资源，创建虚拟机启动进程和数据传输进程，按照第 2 章实现的优化部署方案下载镜像，同时启动虚拟实例，最后完成部署。

3.2 数据结构及通信协议

3.2.1 数据表

系统使用 MySQL 作为数据库，包含如下几张数据表，结构如下：

host(host_id, status, type)

ip(ip_id, ip_addr, ip_type)

image(img_id, img_type, img_name, os_type, img_size, bt_size)

instance(inst_id, inst_type, inst_status, host_id, task_id)

task(task_id, task_status)

host_load (host_id, item_id, value)

host_ip(host_id, ip_id, bind_mac)

host_img(host_id, img_id)

几张数据表的相互关系如下图：

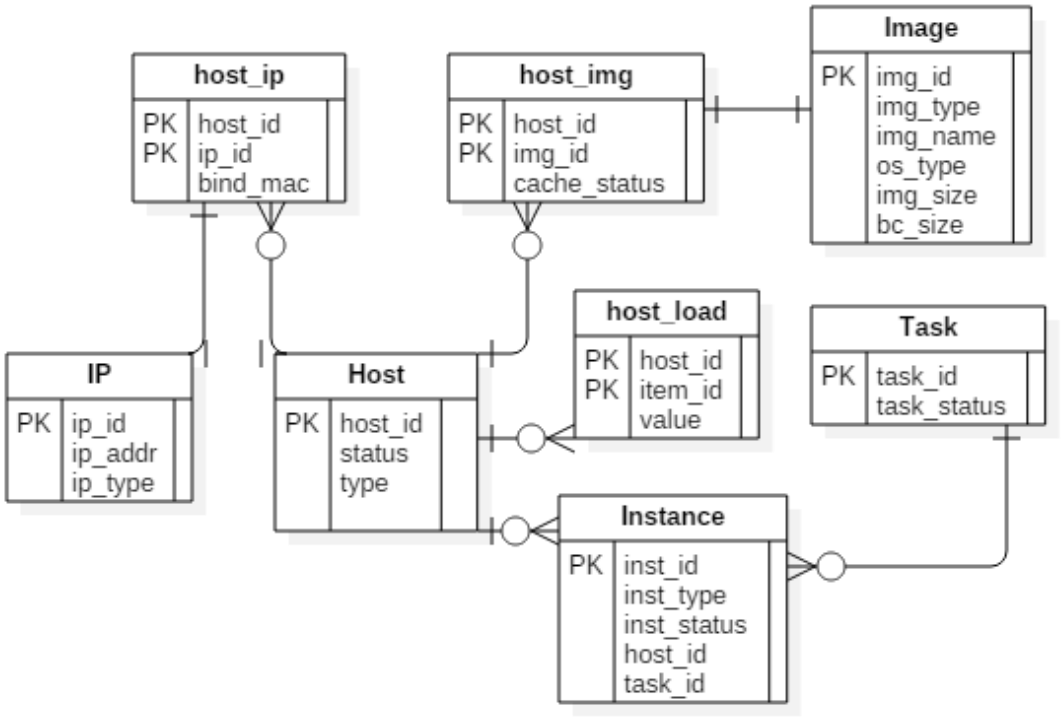

图 3-3 数据表关系图

控制节点通过增删查改图 3-3 中的数据表来维护系统的状态，其中 host 表用于保存为物理节点信息，包括每个节点的 IP、节点状态和类型，节点状态用于表示节点是否在线，控制节点会定时通过心跳包检测计算节点的状态，类型用于区分计算节点和控制节点。ip 表保存了节点的 ip 地址，通过 host_ip 表中的项与节点关联，其中 ip_type 为协议类型（ipv4 或者 ipv6），ip_addr 是 ip 地址。image 表用于保存镜像的信息，包括镜像 ID、镜像类型、镜像名字、操作系统类型、镜像总大小和引导数据大小，其中镜像类型用于区分虚拟机和 Docker 容器。instance 表用于保存虚拟实例，包含了实例 ID，实例类型，实例状态、宿主计算节点 ID 和从

属的任务 ID，其中实例类型用于区分实例是传统虚拟机和 Docker 容器，实例状态包括已停止、资源准备（传输镜像等）、镜像启动中和运行中。task 表用于保存任务项，每次任务由用户发起，包含了部署的虚拟实例个数、种类、宿主实例等信息，这些信息一部分保存在实例表中，这个表只是用作识别任务。host_load 表用于保存节点的负载，包括对应的负载项和值，其中负载项目包括 cpu 使用率、内存使用率、网络使用率等，可以根据用户需求进行扩展。host_img 用于关联镜像和计算节点，当计算节点缓存完镜像之后，会向控制节点注册缓存服务，控制节点就会在这个表中增加对应的一项。

3.2.2 协议数据结构

系统的模块和节点之间需要进行通信，主要包括两个通信过程：

（1） 用户控制接口与控制节点的通信

（2） 控制节点和计算节点的通信

（1）过程主要包括用户控制系统进行部署和查询系统状态，底层是用户控制接口通过 socket 与控制节点守护进程通信；（2）过程是在系统网络中，控制节点的通信模块和计算节点的计算代理模块进行通信。系统在这里使用 TCP 协议，以保证数据传输的正确性，并在这之上封装了一层私有的数据包协议，以便于系统各个节点进行解析和执行。数据包的结构如下图：

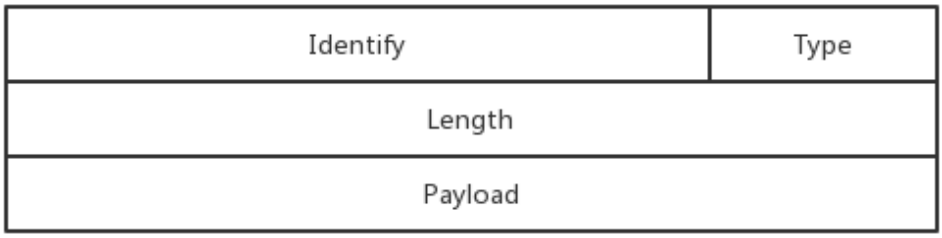

图 3-4 协议数据包结构

数据包由 4 个部分组成：标识符 Identify、操作类型 Type、数据长度 Length 和 Payload 构成。

Identify 固定为’MFD’三个字符的 ASCII 码，所有数据包都是以这三个字节开头，以确定边界。Type 为数据包的类型，其含义如下：

表 3-1 数据包类型

执行节点	Type	含义
控制节点	0x00	通用回复数据包
控制节点	0x01	计算节点注册计算服务
控制节点	0x02	计算节点注册镜像服务
控制节点	0x03	用户发起部署任务指令
控制节点	0x04	用户发起终止任务指令
控制节点	0x05	计算节点发送物理节点状态
控制节点	0x06	计算节点发送虚拟实例状态
控制节点	0x07	用户或计算节点查询节点状态
计算节点	0x08	执行部署任务
计算节点	0x09	执行终止任务

Length 是数据包去除头部的长度，也就是 Payload 的长度。Payload 是数据部分，通信过程需要的数据都保存在这里。3 字节的 Identify、1 字节的 Type、4 字节的 Length 和变长的 Payload 构成了整个数据包。

3.3 控制节点

3.3.1 节点管理模块

节点管理模块主要用于维护系统所有节点，提供了节点注册、节点信息查询等服务。新的节点要加入系统，首先需要在本模块进行注册，这样才能使用系统的服务，注册流程如下：

（1） 计算节点连接到控制节点，发送节点注册请求。

（2） 控制节点测试计算节点的连通性，生成节点 ID，将节点基础信息和网络信息写入到数据库，将节点 ID 回复给计算节点。

（3） 计算节点设置自身 ID，完成注册。

在节点注册之后，控制节点会定期向计算节点发送心跳包，一方面确认计算节点是否存活，另一方面，实时获取计算节点的状态，并更新数据库。这样数据库的内容是实时更新的，在用户或者其他计算节点获取目标计算节点信息时，只需要

从数据库中查询相应的内容。更新数据库的过程是全局统一更新，也就是在轮询了所有计算节点的状态之后，统一进行写入，以减少数据库的访问次数。

3.3.2 任务管理模块

虚拟实例的因为调度部分的任务已经在任务规划阶段完成，本系统并不进行任务规划，而是将规划好的任务快速的分发、部署和引导起来。系统的任务管理模块用于根据用户的指令，管理虚拟机和 Docker 实例，任务管理模块实现了对实例启动和终止的操作。

用户通过用户接口将部署任务发送到任务管理模块，任务的数据结构包含了目标节点 ID，部署的实例的类型、数目和镜像等信息。任务管理模块将这些信息解析后，一方面在数据库中创建对应的条目，另一方面，通过通信模块，向各个计算节点分发部署任务。同时，任务管理模块将根据计算节点返回的虚拟实例状态信息动态更新数据库。

3.3.3 镜像管理模块

镜像管理模块用于维护系统的镜像，提供镜像注册、镜像查询和镜像缓存节点优化选择服务。当计算节点完成缓存之后，会向控制节点声明并注册镜像，此时镜像管理模块将创建新的镜像关联条目，将其绑定到计算节点，并写入数据库。在此之后，其他的计算节点都可以通过计算节点得知该节点提供对应镜像的缓存服务。同时镜像管理模块将监视各个计算节点的状态，一方面，用于在镜像传输时候选择优化的节点，另一方面，调度模块提供了一个接口，通过实现这个接口以集成任意镜像传输算法，以适应不同的环境。在本文的场景中，用于在镜像传输过程中选择一个最优的目标镜像节点。

本文将最优节点定义为负载最低的节点，其负载是由多个参考指标组成，包括 CPU、内存、磁盘和网络，每个指标对部署系统的影响不同，所以需要对每一个指标设置一个权值。系统通过盘点函数计算负载，定义如下：

$$f(x_1, x_2, \dots, x_N) = \Sigma_i^n w_i x_i$$

其中，函数输入为 N 个参考指标，x_i表示第 i 个参考指标，w_i表示第 i 个指标的权重。满足：

$$0 \le x_i \le 1$$

$$0 \le w_i \le 1$$

$$\Sigma_{\mathrm{i}}^{n} w_i = 1$$

计算节点在获取镜像之前，会通过控制节点使用判定函数选取最优节点。由于镜像传输对网络和磁盘的利用率较为敏感，这里将这两个指标的权值设置得相对较高。

3.3.4 协议封装和传输模块

这个模块主要用于将数据按照协议封装，并进行网络通信；同时监听请求，并将请求分发到各个模块进行处理，其流程如下：

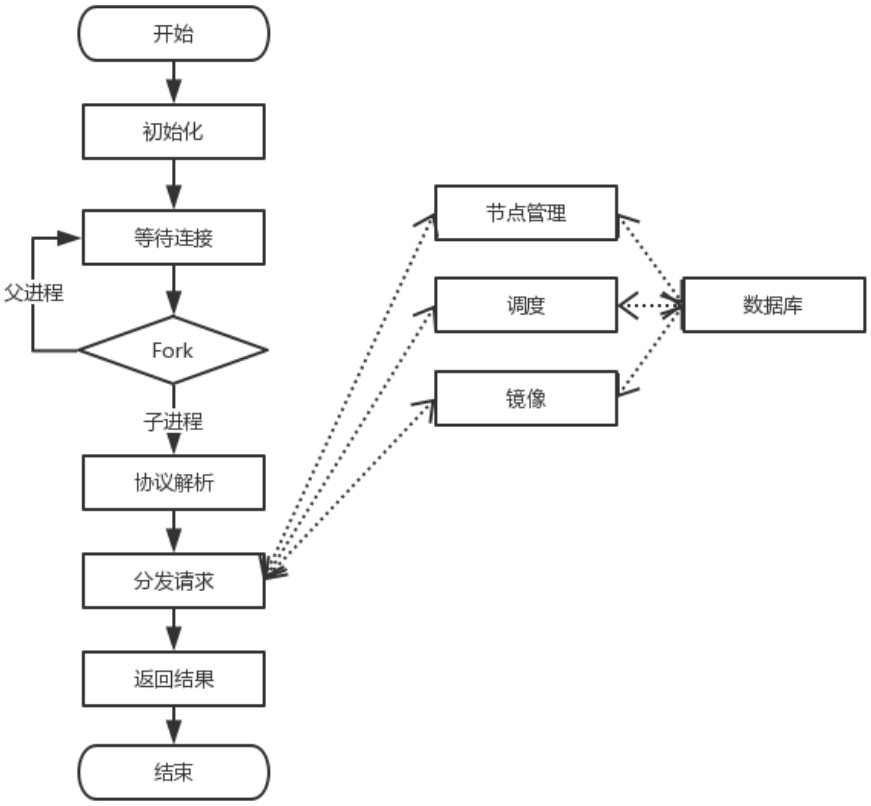

图 3-5 请求分发处理流程

模块首先进行初始化，包括创建套接字、绑定地址、监听端口等，然后开始等待连接。当有新的连接连入之后，模块创建新的进程以处理请求，父进程继续等待下一个连接。子进程根据协议将网络数据接收完成之后，根据请求的类型，调用对应模块的接口，将请求发送到该模块，并在模块处理完成之后，将结果返回到远程连接。

3.4 计算节点

3.4.1 计算代理及抽象接口

计算节点通过计算代理模块跟控制节点通信，包括接收控制节点的部署/终止指令、向计算节点注册物理节点、发送实例/物理节点的状态。在收到数据之后，计算节点代理在将指令解析后调用抽象接口执行。

计算节点实现了一个抽象层，用于隐藏各种不同 Hypervisor 和虚拟网络设备的实现，向控制节点提供统一的接口，主要包含以下功能：

表 3-2 抽象接口

接口	含义
Launch	运行一个实例
Stop	停止一个实例
Delete	删除一个实例
Info	查看实例信息

Launch 接口用于启动一个实例，接收 3 个参数，ID、Config 和 Type。其中 ID 是实例的唯一识别号，由控制节点任务模块分配获得；Config 是一个列表（可选，Docker 不需要配置），包含了虚拟实例的配置信息，例如核心数、内存大小、虚拟网络接口数目；Type 是虚拟实例的类型，对应与不同的 Hypervisor，系统只指定了 Libvirt 和 Docker 两种。启动请求成功返回 0，否者返回-1。需要注意的是，返回 0 并不代表虚拟机启动成功，只是将请求成功提交到了下层 Hypervisor，其是否完成启动需要使用 Info 接口获取。

Stop 接口用于停止一个实例，也就是关机，接收一个参数，虚拟实例的 ID。成功提交返回 0，失败返回-1，虚拟实例是否完全关闭，需要使用 Info 接口获取。

Delete 接口用于销毁一个实例，接收一个参数，虚拟机的 ID。成功提交返回 0，失败返回-1，虚拟实例是否完全关闭，需要使用 Info 接口获取。

Info 接口用于获取实例的实时状态，接收一个参数，虚拟机的 ID。返回虚拟机的状态，状态数据结构包括：虚拟实例状态（运行中、关闭、启动中）、cpu 使用率和内存使用率。

3.4.2 计算模块

计算模块用于封装底层 Hypervisor 的接口，系统使用 Libvirt 和 Docker 两种方案。

Libvirt 使用 XML 文件定义虚拟机实例，一个典型的 XML 文件内容如下图：

```
<domain type='kvm' id='1'>
 <name>186164bd</name>
 <uuid>186164bd-2d0c-eb3c-f397-cc13cfbcd37c</uuid>
 <memory unit='KiB'>524288</memory>
 <vcpu placement='static'>4</vcpu>
.....
 <type arch='x86_64' machine='pc-i440fx-utopic'>hvm</type>
 <boot dev='hd'/>
 </os>
 .....
 <disk type='file' device='disk'>
 <driver name='qemu' type='qcow2'/>
 <source
file='/home/test/vm_test/vms/1461057904434501796/image.qcow2'/>
 .....
 </seclabel>
 <interface type='network'>
   <mac address='52:54:00:37:96:18'/>
   <source network='br0'/>
   <model type='virtio'/>
   .....
 </interface>
</domain>
```

图 3-6 Libvirt XML 文件

其中 memory 项指定了虚拟机的内存大小，vcpu 项指定了虚拟机的核心数目，disk 项指定了镜像文件的路径，interface 指定了网络接口。每一个 XML 唯一定义一个虚拟实例，在任务部署时候，计算模块将解析任务请求得到的配置信息写入 XML，之后使用 Libvirt 的 define 接口对虚拟机进行定义，定义完成之后，虚拟机处于关机状态，此时就可以使用 start 接口运行虚拟机。

Docker 是基于 LXC（LinuX Container）技术，利用 linux namespace 做运行环境隔离，使用 cgroups 技术做资源限制的轻量级虚拟化方案。在 Docker 虚拟容器运行之前，需要导入（下载）镜像文件并导入，然后使用 run 命令运行容器。使用 stop 命令停止容器，rm 命令删除容器。封装的接口对应关系如下：

表 3-3 接口封装

接口	Libvirt 封装	Docker 封装
Launch	virsh define XML + virsh start VM	docker run
Stop	virsh stop VM	docker stop
Delete	virsh destroy VM	docker rm
Info	virsh dumpxml VM	docker info

3.4.3 网络模块

网络模块用于连通虚拟机二层网络，系统在二层使用平坦结构以简化网络结构，并将复杂的虚拟网络拓扑交给三层来做。网络通信模块底层使用 Linux Bridge 实现：

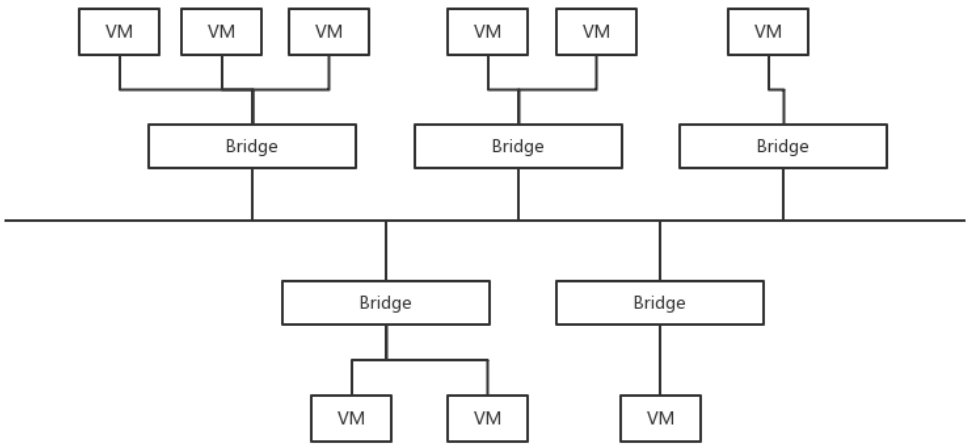

图 3-7 二层网络

Linux Bridge 相当于物理交换机，将虚拟机的虚拟网络接口接入到 Linux Bridge 之后，同一个 Linux Bridge 下的所有虚拟机将处于一个局域网中，Linux Bridge 支持 VLAN，用于控制广播域。系统通过 Linux Bridge 将虚拟机网络和 Docker 的网络连接在一起，再通过在三层网络使用软路由实现拓扑网络的映射。

3.5 本章小结

本章节构建了一个简单高效的虚拟机容器混合部署系统，借助 Docker 容器的轻量级部署方案，替换了传统虚拟化方案中以 Linux 为基础系统、功能简单的虚拟机实例，以提高资源利用率。通过构建虚拟机抽象接口，隐藏 Docker 和传统虚拟机的技术细节，使用 Linux Bridge 将虚拟机网络和 Docker 网络连通。

第 4 章　测试实验与分析

测试分为两个部分，第一个部分对第 2 章镜像优化部署系统进行测试，第二部分对统一部署框架进行测试，将纯容器技术和传统虚拟化技术的部署方案进行对比，测试容器虚拟机混合部署系统的性能。

4.1 测试环境及流程

物理主机节点的配置项如下表所示：

表 4-1 镜像优化系统配置

节点	CPU 数量（个）	内存（*GB*）	网络（*Mbps*）	操作系统
控制节点*1	4*1	16*1	1000/100	Ubuntu 14.04 Server 64bit
计算节点*4	16*2	16*8	1000/100	Ubuntu 14.04 Server 64bit

客户机镜像列表：

表 4-2 操作系统镜像列表

#	操作系统	镜像实际大小（*MB*）	镜像逻辑大小（*MB*）
1	Ubuntu 14.04	1658	5120
2	Ubuntu 16.04	1759	5120
3	Cent OS 6.4	2133	5120
4	Windows 7	7456	20480
5	Ubuntu 14.04 (Docker)	188	5120

在虚拟机测试实验中，使用 Ubuntu（14.04 和 16.04）、Cent OS 6.4 和 Windows 7 三种操作系统镜像进行测试；在虚拟机容器混合部署实验中，增加了基于 Ubuntu 14.04 的 Docker 镜像。

客户虚拟机的配置为 1 个 CPU 核心、512MB 内存。镜像中安装了完整的操作系统，镜像格式为 QCOW2，虚拟化 Hypervisor 为 Libvirt（KVM）。操作系统以

进入系统并主动执行初始化脚本，并能与外部主机相互 ping 通为引导成功标志，总体测试流程如下：

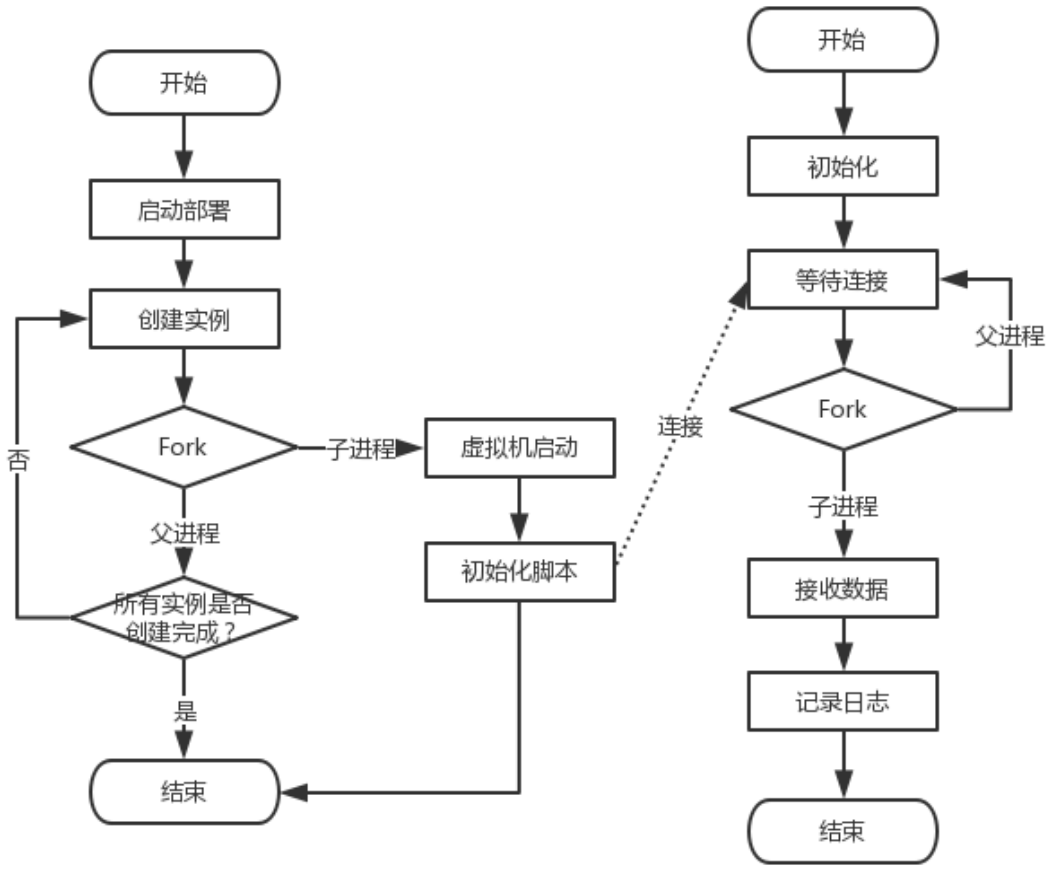

图 4-1 测试流程

左边是部署流程，右边是监视服务流程。监视服务用于记录虚拟机引导完成的时间和相关信息。

控制节点启动部署之后，向计算节点发送虚拟机引导指令。计算节点接收到指令后，对每一个实例，创建一个进程进行启动，父进程则循环创建实例，直到所有实例创建完成。虚拟机镜像中包含一个初始化脚本，脚本在虚拟机启动完成时自动执行。脚本的功能是连接到监视服务监听的端口，并发送虚拟实例自身的 ID 到监视服务器。监视服务进程与部署进程同时启动，启动时候记录时间，作为起始时间。在初始化之后，进入监听循环，等待虚拟实例的连接，对每一个虚拟实例的连接，创建一个新的进程，接收虚拟实例 ID，并记录连接时时间。

部署的总耗时即监视服务器最后一条记录的时间戳减去部署启动时候创建的时间戳。

4.2 镜像优化系统测试

4.2.1 系统配置及测试方案

测试分为两部分，一部分是对操作系统 I/O 进行分析，包括操作系统引导 I/O 请求包大小分布、I/O 请求数据偏移分布和引导数据总和；另一部分是测试系统的性能，并与传统方案进行对比。

针对不同测试项目，使用以下几组测试用例：

表 4-3 测试用例

编号	测试用例	节点配置	备注
1	I/O 大小分布	计算节点*1	
2	I/O 偏移分布	计算节点*1	
3	引导数据占比	计算节点*1	
4	单节点对比测试	控制节点+计算节点*1	测试单虚拟实例启动速度
5	多节点对比测试	控制节点+计算节点*3	测试多虚拟实例启动速度

表 4-3 中 1 到 3 号测试用例均在本地进行，用于统计和分析操作系统引导过程 I/O 请求，用于与后面远程挂载部署方案对比；4、5 号是对比测试，其中 4 号用于测试在有限网络带宽的情况下测试系统方案相对其他方案的性能，5 号相对 4 号增加了计算节点，扩大规模，用于镜像缓存的功能性测试。

4.2.2 I/O 大小分布

操作系统在启动过程中会产生大量的 I/O 请求，这些请求的数据长度各不相同。类似于文件复制，受磁盘寻道性能影响，这些数据长度的不同，会有不同的磁盘响应速度，具体来说，复制相同数据量的大文件所消耗的时间，远小于零碎文件的耗时。为了统计分析操作系统引导过程所有 I/O 请求的大小分布情况，我们记录了几种不同操作系统引导过程的 I/O 请求，并将这些请求数据的大小进行了统计，如图 4-2 所示：

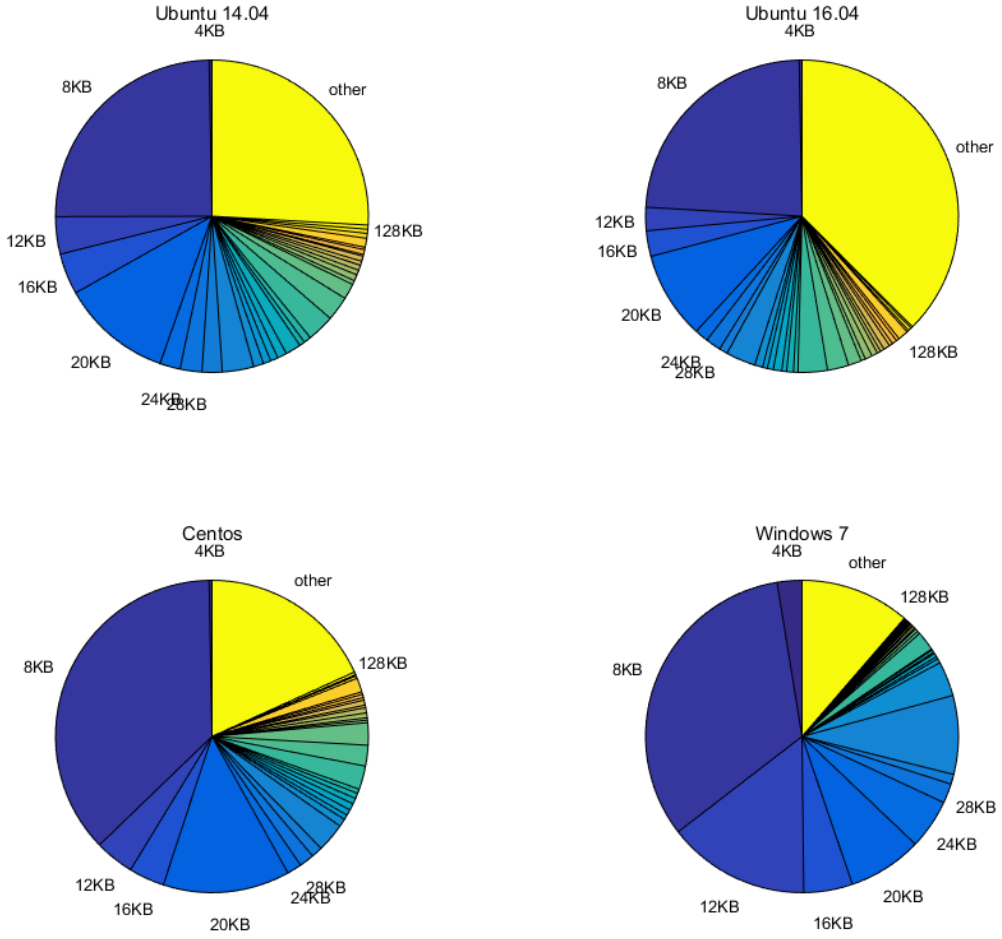

图 4-2 操作系统读取请求数据长度分布

每个操作系统对应一张饼状图，饼状图的区块以 4KB 为单位依次递增，每一块的面积表示该块对应的数据长度的请求数占总请求的比例。从图中可以看出，Windows 7、CentOS 和 Ubuntu 14.04 的小块请求数量较多，20KB 以下的请求数占了较大部分；相对来说，Ubuntu 16.04 启动时候大于 128KB 的 I/O 请求数要多一些。

由于这样的碎片请求较多，虚拟机在引导时候会消耗大量的时间用于磁盘寻道，类似于文件复制，总大小相同，单个大文件的复制速度比大量小文件要快很多。通过 4.2.5 节与远程系统引导的方式对比测试可以看出其性能差距，4.2.3 节将分析这些请求的偏移分布。

4.2.3 I/O 偏移分布

图 4-3 展示了操作系统引导过程中，读取请求的数据在整个镜像的分布情况：

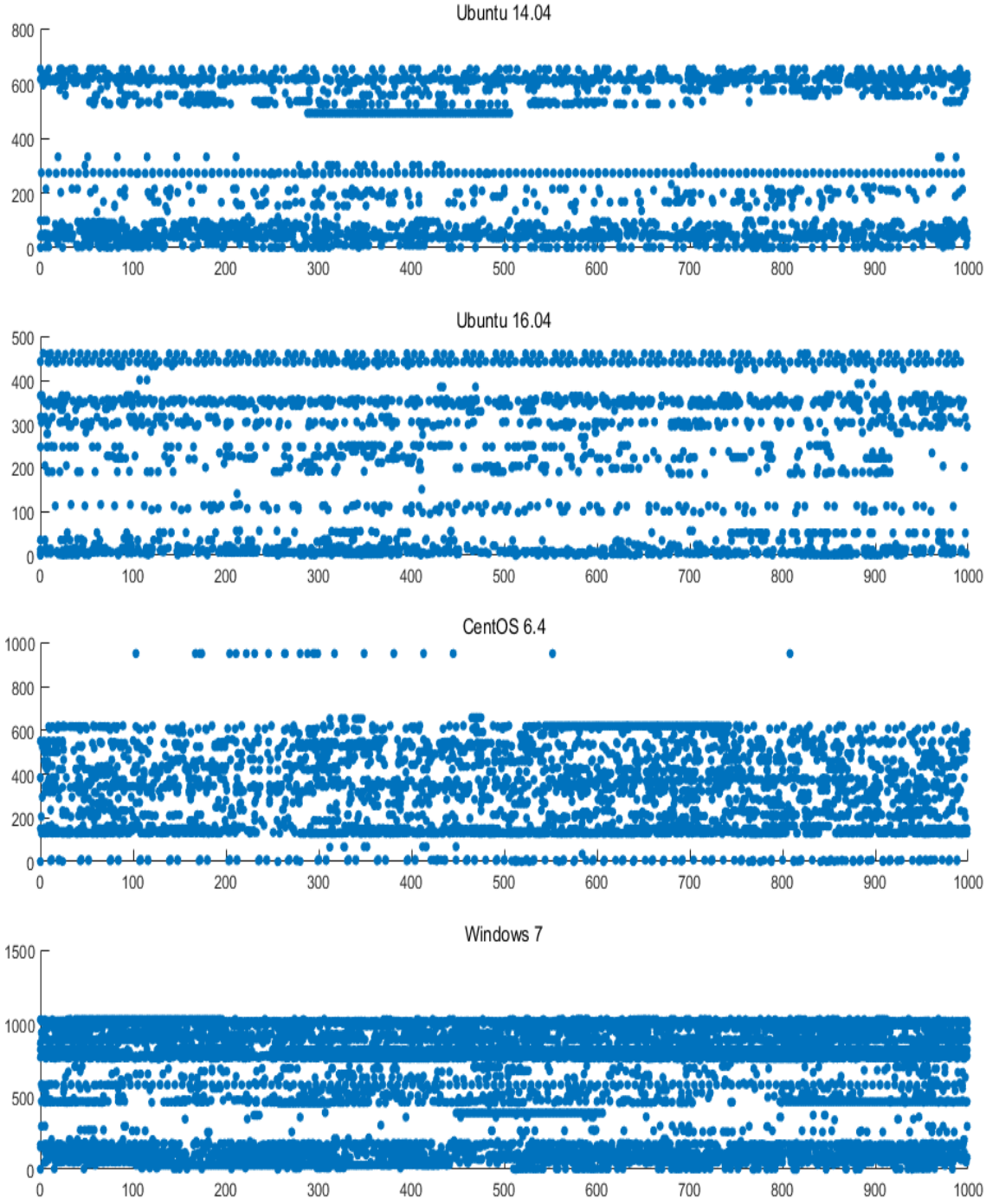

图 4-3 操作系统读请求数据分布

每一个图对应与操作系统的镜像，每张图是操作系统镜像的映射位图，图中每个点代表一次 I/O 请求的起始块。可以看出，几个操作系统引导过程的分布都是不均匀的，同时也可以发现，Windows 7 的度请求数量明显大于其他几个操作系统。这些点的离散度越高，说明启动所需数据在磁盘的分布就越不连续，在系统启动时会有更多的时间消耗在磁盘寻道上。

4.2.4 引导数据大小占比

图 4-4 展示了不同操作系统引导阶段磁盘数据的读取量：

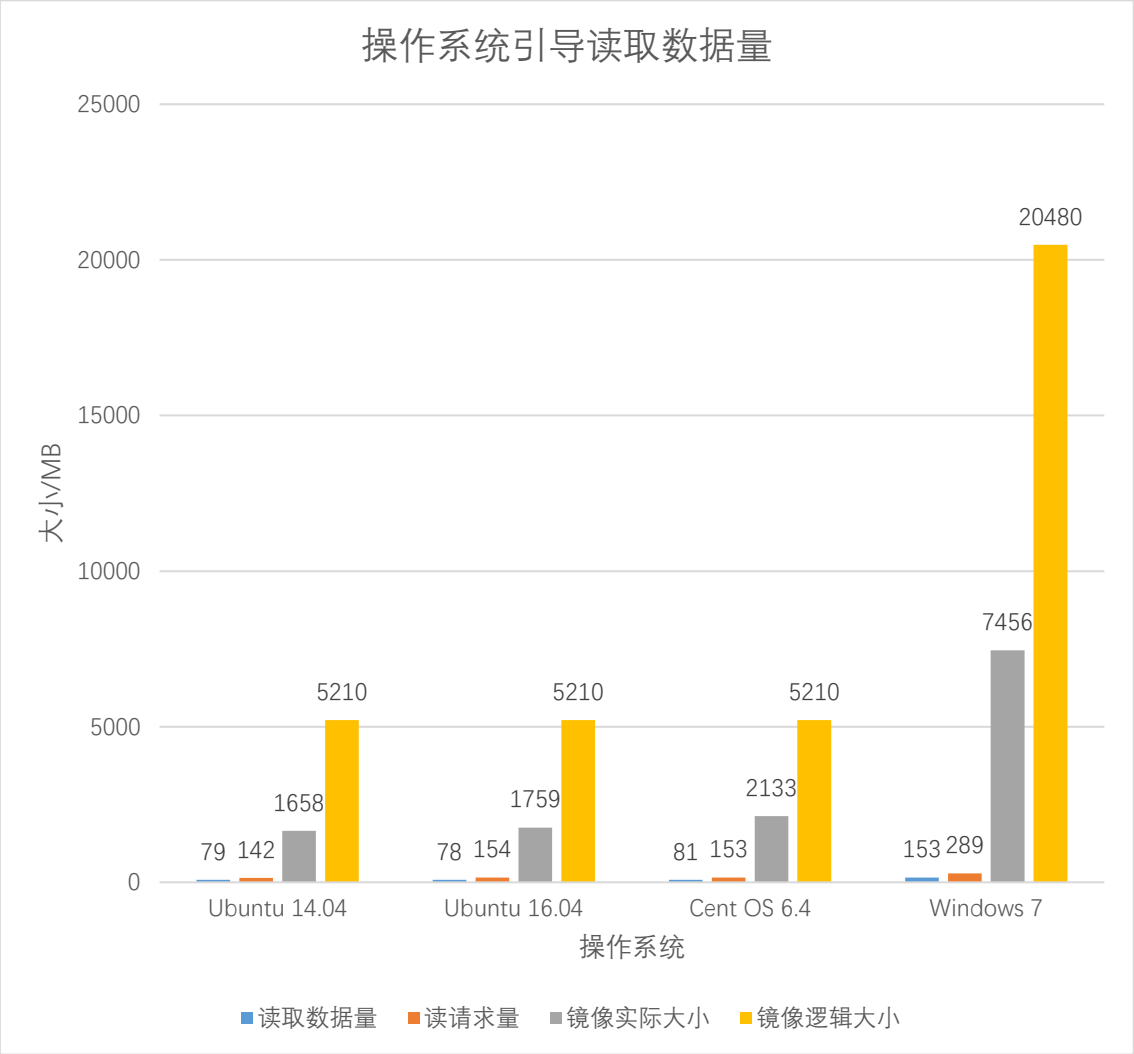

图 4-4 操作系统引导读取数据量

其中镜像逻辑大小指镜像文件对虚拟机声明的大小，镜像实际大小是指镜像文件在宿主机磁盘中实际占用空间，读请求量指的是引导阶段所有读请求的数据量总和，读取数据量指的是引导阶段实际从磁盘读取的数据量。之所以读请求量和读取数据量有所不同，是因为镜像中有些数据块被反复读入，这些重复的读入在远程引导时候是不需要的，所以本文提出的方案将数据传输总量降到了读取数据量显示的数值。

4.2.5 性能测试

本章节测试表 4-3 中第 4、5 项用例，表 4-4 显示了不同的对比测试方案：

表 4-4 对比方案

运行方案		说明
对比方案	本地运行	镜像就在本地，直接使用 Libvirt 启动
	常规部署	将完整的镜像文件从远程下载到本地，再启动
	远程挂载	将远程镜像使用网络文件系统挂载到本地并启动（高 I/O）
优化方案	单镜像节点	使用本文的优化系统进行启动
	双镜像节点	同优化部署，区别在于部分计算节点上已经缓存有部署镜像

对每一个用例按照表 4-4 进行测试，包括本地直接运行、使用网络文件系统进行挂载运行和常规方式运行，实验网络环境为 100Mbps。

针对第 4 项测试，我们在控制节点设置镜像缓存服务，即在控制节点上面运行一个镜像服务进程，其提供服务的镜像通过初始化脚本直接导入到控制节点的数据库中。然后在另外一个计算节点上面部署虚拟机实例，实例数量设置为 1，每个镜像测试 3 次，对不同方案轮流测试，我们将结果统计如下：

表 4-5 用例 4 测试结果表

操作系统	本地运行（s）	优化部署（s）	远程挂载（s）	常规部署（s）
Ubuntu 14.04	4.782740584	25.84562957	30.96441333	173.6324873
Ubuntu 16.04	10.67426848	24.41475798	29.99721726	189.9846581
Cent OS 6.4	12.75928325	25.5419857	28.18787252	228.264858
Windows 7	16.76794164	40.5413679	58.98004955	764.1254896

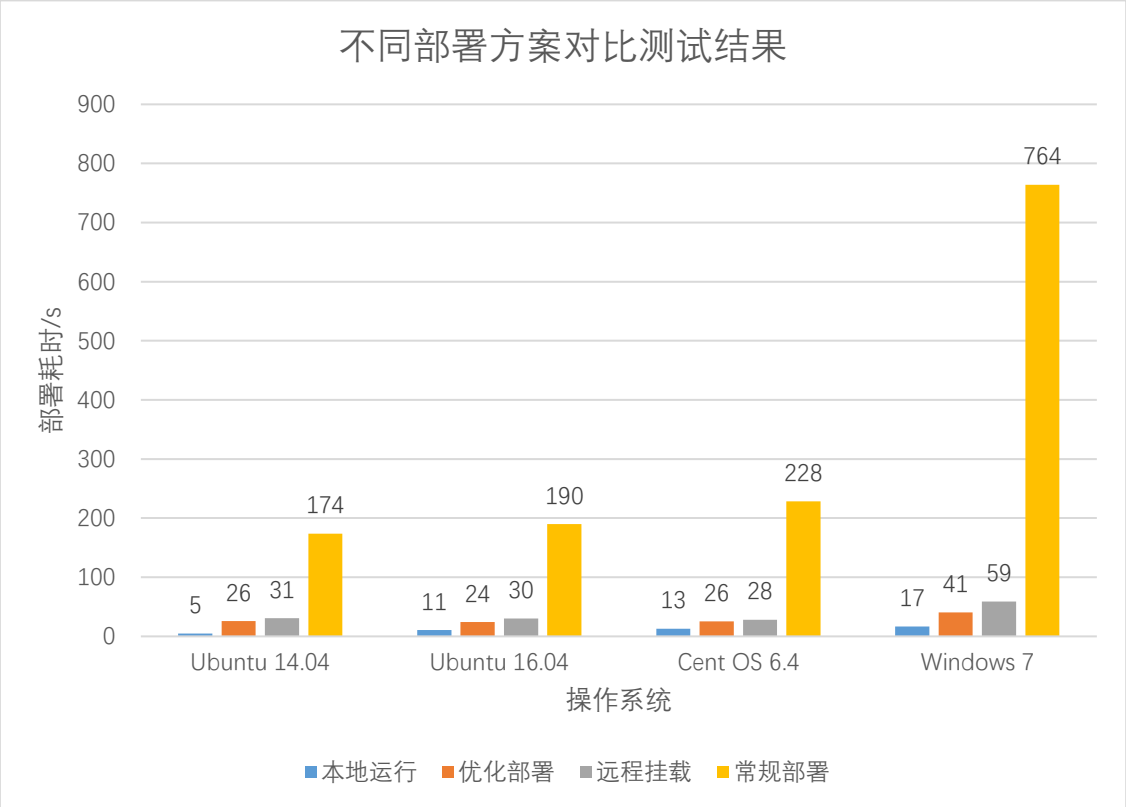

图 4-5 用例 4 测试结果图

通过表 4-5 和图 4-5 可以看出，常规部署方案是很慢的，大约等于把镜像数据传输到计算节点的耗时加上镜像在本地运行的时间，远程引导方案大幅提升了部署效率，而本文使用的优化方案进一步提升了部署效率。作为对比，本地引导的方案的耗时理论上是远程部署耗时的极限，网络性能越好，越接近这个极限。

针对用例 5，控制节点配置跟用例 4 一样，计算节点增加到 3 台，3 台计算节点并行启动，分别运行 1 个虚拟机实例进行测试。由于本地运行模式在这种模式下跟用例 4 无区别，所以不测试本地运行方案，相对的，增加一次将某个计算节点进行缓存后的系统测试，用于测试缓存功能的性能，测试结果如下：

表 4-6 用例 5 测试结果表

操作系统	优化部署(缓存)（s）	优化部署（s）	远程挂载（s）	常规部署（s）
Ubuntu 14.04	26.38573649	48.18362759	64.38593716	502.1728475
Ubuntu 16.04	24.17564398	44.67264939	65.18273522	523.8627738
Cent OS 6.4	26.11836626	46.06839264	70.18267492	631.9892746
Windows 7	41.27483659	84.49385655	102.1728492	2414.384787

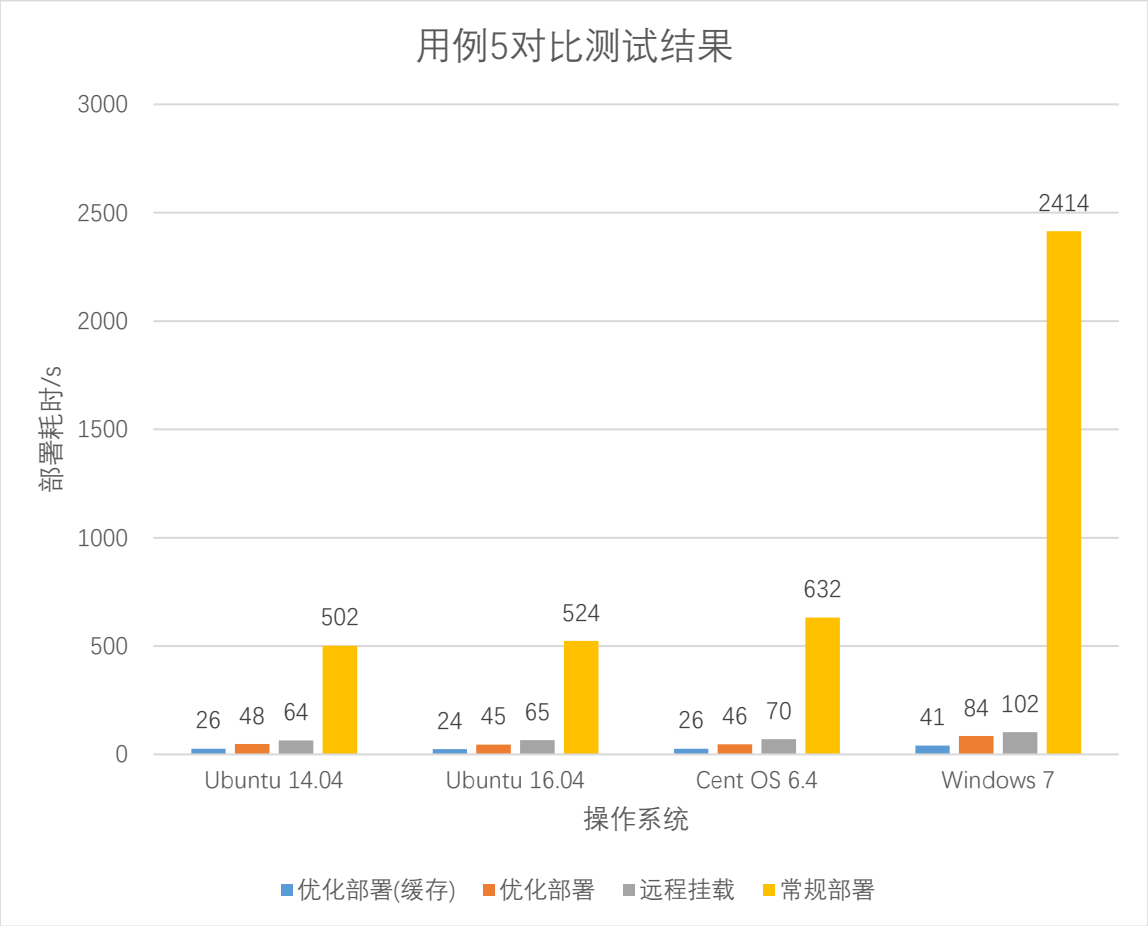

图 4-6 用例 5 测试结果图

通过表 4-6 和图 4-6，对比用例 4 的结果，可以看出随着规模的增加，部署耗时也几乎是线性增加，其瓶颈在于单点的网络带宽，而在其中一个计算节点创建缓存之后，相当于两台计算节点分别从两个镜像缓存服务器下载镜像数据，能够大幅提升部署效率。

4.3 虚拟机容器混合部署系统性能测试

本章节测试统一部署框架的功能，并以虚拟机容器混合优化部署系统为例，测试其性能。

测试使用 1 个物理控制节点和 4 个计算节点，100Mbps 网络，虚拟机镜像使用 Ubuntu 14.04（1658MB），容器镜像使用 Ubuntu 14.04（188MB），虚拟化方案使用 Libvirt，容器方案使用 Docker，网络环境为 1000Mbps。测试用例如下：

表 4-7 混合部署用例

#	实例个数	计算节点个数	部署配置
1	100	4	传统虚拟化方案(Openstack)
2	100	4	优化虚拟化方案
3	100	4	容器方案
4	20 + 80	4	混合优化方案（虚拟机:Docker = 2:8）

我们将启动 100 个虚拟实例用于测试，记录虚拟实例部署总耗时并计算单位时间部署实例的个数。其中用例 1 为传统虚拟化方案，部署时候将拷贝虚拟机镜像到各个节点再引导；用例 2 使用本文的优化系统进行部署；用例 3 使用 Docker 进行部署；用例 4 使用本文的混合部署系统，将虚拟机与容器，按照 2:8 的比例进行部署。经测试有如下结果：

表 4-8 混合测试结果表

#	总耗时（*s*）	单位时间部署实例个数（个）
1	112.182942356	0.89
2	47.048294712	2.13
3	15.663729531	6.38
4	23.294582091	4.29

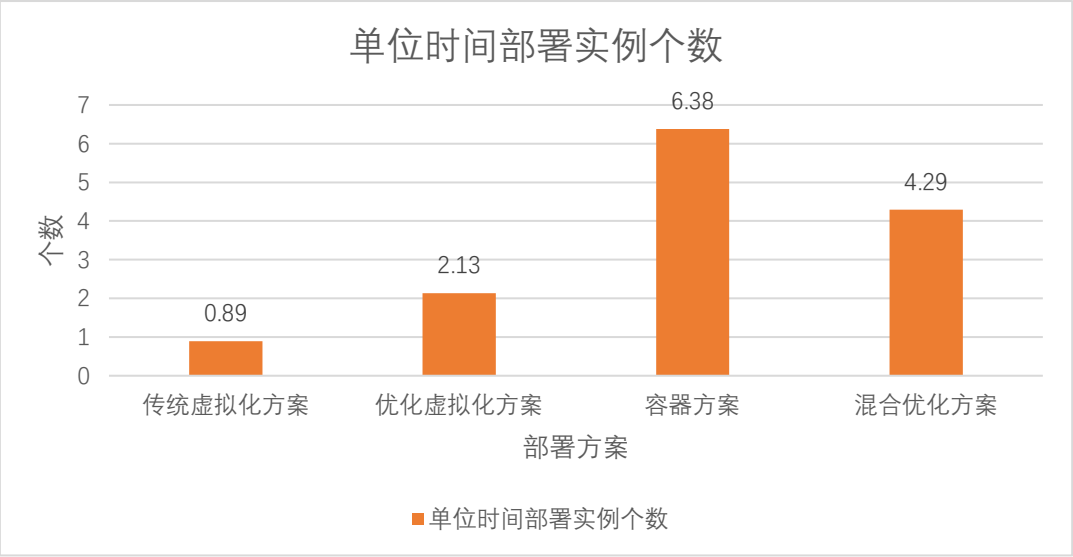

图 4-7 单位时间部署实例个数

从表 4-8 和图 4-7 可以看出，传统虚拟化方案单位时间部署的虚拟机个数是最少的，本文的优化部署方案在此基础上提升了部署的效率。容器方案的部署效率大幅领先于虚拟化方案，而按照 2:8 虚拟机和容器混合比例的部署方案部署效率相对纯虚拟化方案也有明显提升。

4.4 实验分析及结论

通过 4.2.2 到 4.2.4 的实验，我们对操作系统引导过程 I/O 请求的调用进行了多个角度的分析，包括 I/O 请求大小、偏移和整个流程 I/O 请求数据量，得到了操作系统引导真正需要的数据。在虚拟机部署过程中只需要传输这部分数据即可完成启动，并通过 4.2.5 的性能测试与传统方案进行对比，我们发现本文提出的优化方式能够提升部署效率。

除此之外，通过融入不同的虚拟化方案，构造一个统一的部署框架，以克服单一虚拟化方案，我们将容器技术引入传统虚拟化部署方案中，并在 4.3 中对其进行测试，将虚拟机容器混合部署系统和纯虚拟机、容器方案性能进行对比，发现这种方式可以提升传统虚拟机部署效率，同时相对容器有更好的兼容性。

结　论

随着基础硬件的发展和需求的提高，虚拟网络系统的场景变得越来越复杂，规模越来越大，传统的虚拟网络部署方案已经出现了较大的性能瓶颈。本文从虚拟机镜像优化的角度出发，在考虑到 I/O 性能的同时，构造了一种传输带宽占用最少的部署方案，以加快虚拟网络部署应用中虚拟机的启动过程，同时结合现有的容器技术，实现了一个虚拟机容器混合部署系统原型。

本文主要贡献如下：

（1）分析了操作系统引导过程中，I/O 请求的流程，对其单个大小、总大小和偏移做了统计，提取出操作系统引导所必需的数据，以作为虚拟网络部署场景中最先传输的数据。

（2）设计和实现了一个文件系统过滤层，重新规划虚拟网络部署过程中镜像传输的流程，重新构建系统镜像。提出了基于网络流的操作系统引导方式，在不带来额外 I/O 开销的前提下，降低虚拟网络部署过程中镜像传输对网络带宽带来的消耗。

（3）设计了一个统一的虚拟网部署框架，其底层可以由多种虚拟化技术实现，对上层提供统一的接口，用于满足复杂部署环境的需求，提高系统的性能。

（4）使用容器技术提升虚拟网络部署效率，将底层容器技术与传统虚拟化技术进行封装，隐藏实现的细节，向上层部署系统提供统一的抽象接口，并通过网桥打通虚拟机与容器之间的网络，实现了一个虚拟机容器混合部署系统原型。

在接下来的研究中，还需要从以下方向进行改进：

（1）将本文提出的镜像优化部署方案进行拓展，以兼容容器技术，并与传统云计算平台进行整合，提高灵活性。

（2）本文在网络传输过程中对镜像节点的选择是基于选择适合节点的负载，未来可以选择更加准确的负载参考指标，并实现动态的负载均衡。除此之外，镜像在缓存的过程中也是可以把已经缓存的部分，提供给其他需要的计算节点，以提高整体网络的利用率。

（3）本文对容器与传统虚拟化的整合只是做了简单的抽象，例如在虚拟机的网络方面，只是简单的在二层做了连通，并通过 VLAN 和三层路由进行隔离和配置，这种方式存在很多局限。之后的研究可以结合现有的网络技术，引入 SDN，进行改进和优化。

参考文献

[1] Goldberg R P. Survey of virtual machine research[J]. Computer, 1974, 7(6):34-45.

[2] Smith J, Nair R. Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)[M]. Morgan Kaufmann Publishers Inc. 2005.

[3] Bhardwaj S, Jain L, Jain S. Cloud computing: A study of infrastructure as a service (IAAS)[J]. International Journal of Information Technology & Web Engineering, 2010, 2(1):60-63.

[4] Zhu Y, Ammar M. Algorithms for Assigning Substrate Network Resources to Virtual Network Components[C]// INFOCOM 2006. IEEE International Conference on Computer Communications. Proceedings. IEEE, 2007:1-12.

[5] 李爱国, 原建伟. 云计算部署模式及应用类型研究[J]. 电子设计工程, 2013, 21(2):24-26.

[6] Wada K. Redundant Arrays of Independent Disks[M]. Springer US, 2009.

[7] 李大伟. 基于 IaaS 的网络靶场试验系统设计与实现[J]. 指挥信息系统与技术, 2015, 6(5):1-6.

[8] Ammons G, Bala V, Mummert T, et al. Virtual machine images as structured data: the mirage image library[C]// Usenix Conference on Hot Topics in Cloud Computing. 2014:22-22.

[9] Flouris M D, Lachaize R, Bilas A. Violin: A Framework for Extensible Block-Level Storage[C]// MASS Storage Systems and Technologies, 2005. Proceedings. IEEE /, NASA Goddard Conference on. IEEE, 2005:128-142.

[10] Jayaram K R, Peng C, Zhang Z, et al. An empirical analysis of similarity in virtual machine images[C]// MIDDLEWARE 2011 Industry Track Workshop. ACM, 2011:6.

[11] Peng C, Kim M, Zhang Z, et al. VDN: Virtual machine image distribution network for cloud data centers[C]// INFOCOM, 2012 Proceedings IEEE. IEEE, 2012:181-189.

[12] Sapuntzakis C P, Chandra R, Pfaff B, et al. Optimizing the migration of virtual computers[J]. Acm Sigops Operating Systems Review, 2002, 36(SI):377-390.

[13] Fábrega, F. J. T., Javier, F., & Guttman, J. D. (1995). Copy on write.

[14] Hitz, D., Malcolm, M., Lau, J., & Rakitzis, B. (2005). U.S. Patent No. 6,892,211. Washington, DC: U.S. Patent and Trademark Office.

[15] Sawdon, W. A., & Schmuck, F. B. (2004). U.S. Patent No. 6,748,504. Washington, DC: U.S. Patent and Trademark Office.

[16] Duvall, K. E., Hooten, A. D., & Loucks, L. K. (1988). U.S. Patent No. 4,742,450. Washington, DC: U.S. Patent and Trademark Office.

[17] Sheridan, K. H. (1998). U.S. Patent No. 5,760,917. Washington, DC: U.S. Patent and Trademark Office.

[18] Chen Z, Zhao Y, Miao X, et al. Rapid Provisioning of Cloud Infrastructure Leveraging Peer-to-Peer Networks[C]// IEEE International Conference on Distributed Computing Systems Workshops. IEEE Computer Society, 2009:324-329.

[19] Wartel R, Cass T, Moreira B, et al. Image Distribution Mechanisms in Large Scale Cloud Providers[C]// IEEE Second International Conference on Cloud Computing Technology and Science. IEEE, 2011:112-117.

[20] Schmidt M, Fallenbeck N, Smith M, et al. Efficient Distribution of Virtual Machines for Cloud Computing[C]// Euromicro International Conference on Parallel, Distributed and Network-Based Processing. IEEE, 2010:567-574.

[21] Krsul I, Ganguly A, Zhang J, et al. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing[C]// Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference. IEEE, 2004:7.

[22] Nelson M, Lim B H, Hutchins G. Fast Transparent Migration for Virtual Machines.[C]// Atec '05 : Proceedings of the Conference on Usenix Technical Conference. USENIX Association, 2005:391-394.

[23] Foster I, Freeman T, Keahy K, et al. Virtual Clusters for Grid Communities[C]// Proc. Sixth IEEE International Symposium on CLUSTER Computing and the Grid. IEEE Computer Society, 2006:513-520.

[24] 陈彬. 分布环境下虚拟机按需部署关键技术研究[D]. 国防科学技术大学, 2010.

[25] Kozuch M, Satyanarayanan M, Bressoud T, et al. Seamless Mobile Computing on Fixed Infrastructure[J]. Computer, 2004, 37(7):65-72.

[26] Seo K T, Hwang H S, Moon I Y, et al. Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud[C]// NETWORKING and Communication. 2014:105-111.

[27] Xavier M G, Neves M V, Rossi F D, et al. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments[C]// Euromicro International Conference on Parallel, Distributed and Network-Based Processing. IEEE, 2013:233-240.

[28] Dua R, Raja A R, Kakadia D. Virtualization vs Containerization to Support PaaS[C]// IEEE International Conference on Cloud Engineering. IEEE, 2014:610-614.

[29] Zhang Z, Li Z, Wu K, et al. VMThunder: Fast Provisioning of Large-Scale Virtual Machine Clusters[J]. IEEE Transactions on Parallel & Distributed Systems, 2014, 25(12):3328-3338.

[30] 张钊宁. 云计算大规模弹性资源的性能优化技术研究[D]. 国防科学技术大学, 2014.

[31] 刘圣卓. 面向虚拟集群的镜像存储与传输优化[D]. 清华大学, 2015.

[32] Patterson D. A Case for Redundant Arrays of Inexpensive Disks[J]. Proc.acm Sigmod Conf, 1988.

[33] Bhardwaj S, Jain L, Jain S. Cloud computing: A study of infrastructure as a service (IAAS)[J]. International Journal of Information Technology & Web Engineering, 2010, 2(1):60-63.

[34] Zhang Y, Niu K, Wu W, et al. Speeding Up VM Startup by Cooperative VM Image Caching[J]. IEEE Transactions on Cloud Computing, 2018, PP(99):1-1.

[35] 赵勋. 虚拟计算环境资源调度关键技术研究[D]. 清华大学, 2015.

[36] Mckeown N. Software-defined Networking[J]. 中国通信:英文版, 2009, 11(2):1-2.

[37] Lee C, Lee H, Kim E. Speeding Up VM Image Distribution for Cloud Data Centers[J]. Advances in Electrical & Computer Engineering, 2016, 16(4):9-14.

[38] Álvaro López García, Castillo E F D. Efficient image deployment in cloud environments[J]. Journal of Network & Computer Applications, 2016, 63(C):140-149.

[39] Sefraoui O, Aissaoui M, Eleuldj M. OpenStack: Toward an Open-source Solution for Cloud Computing[J]. International Journal of Computer Applications, 2012, 55(3):38-42.

[40] Ernst G, Schellhorn G, Haneberg D, et al. A Formal Model of a Virtual Filesystem Switch[J]. Electronic Proceedings in Theoretical Computer Science, 2012, 102(Proc. SSV 2012).

[41] Shepler S, Callaghan B, Robinson D, et al. Network File System (NFS) version 4 Protocol[J]. Rfc, 2003, 6(5):3-4.

[42] Hertel C. Implementing CIFS: The Common Internet File System[J]. Pearson Schweiz Ag, 2004.

攻读硕士学位期间发表的学术论文

[1] Zhang W, Hu Y, He H, et al. Linear and dynamic programming algorithms for real-time task scheduling with task duplication[J]. Journal of Supercomputing, 2017(1):1-16.

哈尔滨工业大学学位论文原创性声明和使用权限

学位论文原创性声明

本人郑重声明：此处所提交的学位论文《大规模虚拟网络镜像分发优化策略研究》，是本人在导师指导下，在哈尔滨工业大学攻读学位期间独立进行研究工作所取得的成果，且学位论文中除已标注引用文献的部分外不包含他人完成或已发表的研究成果。对本学位论文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。

作者签名： 日期：2018 年 6 月 24 日

学位论文使用权限

学位论文是研究生在哈尔滨工业大学攻读学位期间完成的成果，知识产权归属哈尔滨工业大学。学位论文的使用权限如下：

（1）学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文，并向国家图书馆报送学位论文；（2）学校可以将学位论文部分或全部内容编入有关数据库进行检索和提供相应阅览服务；（3）研究生毕业后发表与此学位论文研究成果相关的学术论文和其他成果时，应征得导师同意，且第一署名单位为哈尔滨工业大学。

保密论文在保密期内遵守有关保密规定，解密后适用于此使用权限规定。

本人知悉学位论文的使用权限，并将遵守有关规定。

作者签名：  日期：2018 年 6 月 24 日

导师签名：  日期：2018 年 6 月 24 日

致　谢

光阴荏苒，在哈工大就读研究生这两年很快就要结束了。在这期间，“规格严格，功夫到家”的校训的精神时时刻刻提醒着我砥砺前行，在实验室老师、同学的帮助下，我在学术研究和人生阅历上，受益匪浅。毕业在即，我要向他们表示感谢。

首先感谢我的导师张伟哲教授，在两年的学习生涯中，老师在学术上的造诣深深地影响了我。每当我在课题研究的过程中举步维艰时，老师总是能够准确地指出我困惑的原因，启发我找到新的思路和解决问题办法。老师在学术上始终保持严谨的态度，对理论的推导要有理有据，对实验的设计要全面并且实际，对文章的表述要清晰准确。在为人处世方面，老师一直严格要求我们要求实务真，明明白白，这样才能在学术的道路上稳步前行，有所成就。在老师的影响之下，我对科学研究有了更加深刻的理解，在成长的道路上始终保持热情。

感谢实验室张宏莉老师在学习和生活上对我的指导和帮助，感谢何慧老师对我的亲切关怀，感谢翟老师对我还有Lilac社团的帮助和支持，感谢小赵老师给了我们一个舒适的学习工作环境，感谢实验室的小伙伴们陪伴我度过两年的愉快生活。

感谢白恩慈师兄、王德胜师兄和郝萌师兄在科研工作中对我的关照和帮助，感谢王焕然、吴毓龙在我面临困难之时给我的帮助和分担，感谢姜哲、丁泽宇两位师弟对我的协助，有了你们，我的毕设才能够顺利进行下去。

感谢我的父母、家人和所有关注着我的人，是你们的支持，我才有勇气和力量走到今天，也将在未不停前行，永不止步。

最后，愿所有在我生命中伴我同行之人，都将有一个闪光的未来，一路幸福快乐。我会继续努力，向着未来前行。