華中科技大學

本科生毕业设计

面向容器的虚拟机性能分析及优化

院　　系　计算机科学与技术

专业班级　1307 班

姓　　名　王镇宇

学　　号　U201314969

指导教师　吴　松

2017 年 05 月 22 日

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包括任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本声明的法律后果由本人承担。

作者签名：　　年　月　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保障、使用学位论文的规定，同意学校保留并向有关学位论文管理部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权省级优秀学士论文评选机构将本学位论文的全部或部分内容编入有关数据进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于 1、保密口，在　　年解密后适用本授权书

2、不保密口　。

（请在以上相应方框内打“√”）

作者签名：　　年　月　日

导师签名：　　年　月　日

摘　要

云计算技术自产生以来，一路迅猛发展，但随着运行在容器中应用规模变得庞大，逻辑也越发复杂，产品的更新与技术的迭代越来越频繁。容器技术的出现解决了虚拟机在安全和资源调度方面的问题，同时提供高效的资源互操作和保持程序间的相对独立性。

公有云平台上的容器服务普遍基于虚拟机来提供，因此设计一种针对容器场景的高效虚拟机系统对提高容器公有云服务的整体性能至关重要。在现今软件配置和管理部署都逐渐复杂化的背景下，以减少传统虚拟化技术对计算机硬件资源的浪费和简化应用部署管理为目的，基于 Docker 实现了对承载容器的虚拟机系统的简化，使其更效率地运行容器。首先对承载容器的虚拟机系统进行性能分析，找出可能的性能瓶颈，然后对虚拟机客户操作系统或者主机操作系统进行优化，寻找相应的解决方案，针对容器进行虚拟机操作系统内核的定制化。

针对承载容器的虚拟机启动时间进行优化，精简虚拟机系统的内核模块，优化启动过程中的挂载磁盘操作（Initial RAM Disk，initrd）及时钟同步操作（Real Time Clock，RTC），使得虚拟机在启动速度得到提升的同时，能为容器提供稳定的服务。

关键词：Docker；云计算；虚拟机；操作系统

Abstract

Since the birth of Cloud computing technology, it has been developing rapidly all the way, but with the scale of application has become increasingly large, more complex logic, product updates and iteration more and more frequent. The emergence of container technology solves the problem of virtual machine security and resource scheduling, while providing efficient resource interoperability and maintaining relative independence between programs.

The container service on the public cloud platform is generally based on the virtual machine, so it is very important to design a highly efficient virtual machine system for the container scene to improve the overall performance of the container's public cloud service. In order to reduce the waste of computer hardware resources and simplify the application deployment management in the context of the complexity of the current software configuration and management deployment, this paper realizes the simplification of the virtual machine system carrying the container based on Docker, Make the virtual machine system more efficient to run the container. First, analyses the performance of the virtual machine system carrying the container to identify possible performance bottlenecks, and then optimize the virtual machine guest operating system or host operating system. Find the appropriate solution for customizing the container for the virtual machine operating system kernel.

Optimize the virtual machine startup time of the container and simplify the kernel module of the virtual machine system.Optimize the mount disk operation (Initial RAM disk, initrd) and clock synchronization operation (Real-Time Clock, RTC) during startup to make the virtual machine in the start speed is improved while the container can provide a stable service at the same time.

Key Words：Docker; Cloud Computing; Virtual Machine; Operating System

目　　录

1 绪论

1.1 课题研究的背景

随着计算机技术的不断迅猛发展，用户数量的持续增加，为了满足用户提出需求的多样性和复杂性，云计算技术横空出世。它的出现打破了传统网络服务的服务端-客户端（Client Server，CS）架构，云计算可以快速提供用户所需要的资源如存储，网络服务等，用户可以不需要与服务提供商进行直接交互，因此在管理层面来说也是相当便捷。同时，计算机技术的发展也给云计算产业带来了不小冲击，传统云平台的三种基础架构如下图 1-1 所示，这三种模式都在实际生产应用中暴露出一些缺陷，基础架构即服务（Infrastructure as a Service, IaaS）的资源利用率低、平台即服务（Platform as a Service，PaaS）的环境局限性过强等等。

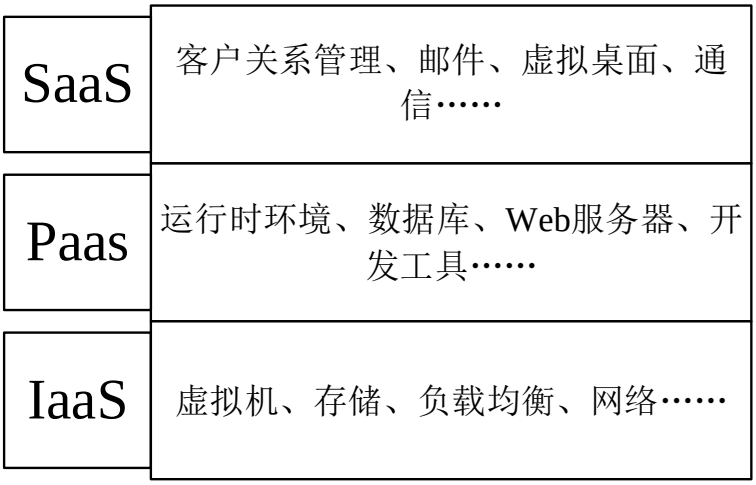

图 1-1 传统云平台架构

Docker 的出现很好的解决这两者的问题，它以容器为资源调度单位，很好的封装软件的运行环境，同时使用 Linux 操作系统内核的控制组（cgroups）与命名空间（namespace）技术提供了很好的资源隔离和限制。在资源不断稀缺的实际生产环境下，如何高效地配置虚拟机来提供容器所需要的运行环境，提高虚拟机的资源利用率，给大规模容器集群的部署和编排带来便捷，同时还能在虚拟机内稳定运行容器相关的服务是一件十分紧要的任务。

因此，面向容器的虚拟机性能优化就是针对容器运行环境，设计一个定制的虚拟机内核，缩减不必要的模块，在提高虚拟机相关性能的同时可以让容器能够高效地在里面运行。

1.2 国内外研究现状

容器技术的出现为用户开辟了一方新大陆，但随着用户在其上初步探索之后，发现容器技术庞大的生态系统给应用带来了相当难度。在生产使用中，个人和企业都期望使用容器解决复杂问题。这个时候我们面对的不是一个容器的管理，而是多个跨主机跨平台的容器集群的管理，我们需要考虑到各种类型的工作负载，企业在使用容器技术进行开发部署时，需要实现支持团队合作、实时反馈的平台。虽然启动一个 Docker 只需要一个 docker run 指令，但我们一旦要将其部署到实际开发与测试场景中去时，多需要考虑的到的就非常多，必须面面俱到才能不出纰漏，如集群的快速部署、容器网络、存储安全、隔离性等一系列问题。

针对以上的问题，容器云的概念应运而生。容器云提供了一个用于构建、发布和运行分布式应用的平台，软件开发者在其上以容器为单位进行资源分割和调度，封装软件运行时环境。

Docker 的出现让人们意识到了轻量级虚拟化技术的价值。Docker 在最初发布时只是一个单机下的容器管理工具，随后 Docker 公司发布了 Compse、Swarm 等一系列编排部署工具。在现阶段的大规模集群部署中，Docker 采用了 Google 的 Kubernetes。容器技术的生态系统十分庞大，因此各种新技术的更新与迭代十分的快，而 Docker 也在不断采取最前沿的技术完善自身。

传统的操作系统镜像不如 Docker 这么轻量化，也有很多企业尝试针对 Docker 开发出面向容器的操作系统，以下是目前一些选择：

1. CoreOS：以容器为中心的操作系统，配置管理、自动扩容、安全等方面有一套完整的工具。

2. Project Atomic：一个轻量级的操作系统，可以运行 docker、kubernetes、rpm 和 systemd。

3. Ubuntu Core：轻量级 Ubuntu 操作系统，适合运行物联网（Internet of Things, IoT）设备或者容器集群。

4. Rancher OS：为运行 docker 容器打造的操作系统。它可以让系统容器和用户容器运行在不同的 Docker Daemon 上，从而实现隔离效果。

5. Project Photon：VMware 开源的项目，旨在提供极简化的容器主机系统。

1.3 研究内容与目标

云计算技术自产生以来，一路迅猛发展，但随着应用规模日益变得庞大，逻辑也越发复杂，产品的更新与迭代越来越频繁。容器技术的出现解决了虚拟机在安全和资源调度方面的问题，同时提供高效的资源互操作和保持程序间的相对独立性。

公有云平台上的容器服务普遍基于虚拟机来提供，因此设计一种针对容器场景的高效虚拟机系统对提高容器公有云服务的整体性能至关重要。在现今软件配置和管理部署都逐渐复杂化的背景下，以减少传统虚拟化技术对计算机硬件资源的浪费和简化应用部署管理为目的，本文基于 Docker 实现了对承载容器的虚拟机系统的简化，使其更效率地运行容器，实现以下设计目标：

1. 首先对承载容器的虚拟机系统进行性能分析，找出性能瓶颈
2. 然后对 Docker 运行在物理机和虚拟机下不同性能进行测试
3. 接着找出 Docker 运行所依赖的内核模块和功能
4. 对虚拟机客户操作系统进行裁剪，优化内核模块，提高系统性能

1.4 文章组织结构

本章介绍了云计算背景及容器技术，简单阐述了目前国内外的研究现状，具体分析了虚拟机性能优化的方案，明确了本课题研究的重点和目标。

第二章先介绍了 Docker 的背景、架构和生态系统，并介绍了相关 Linux 内核的架构知识，理解了内核模块的作用。

第三章在明确课题目标的基础上，对比选取选取测试环境，包括物理机配置、虚拟机配置、容器镜像选取等，之后对性能测试的工具和技术进行深入学习，分别使用各种方法对虚拟机性能和 Docker 容器性能进行测试，收集优化前的各项测试数据，主要从 CPU 使用率、系统平均负载、磁盘 I/O 和网络传输方面来分析。

第四章由测试数据结合 Docker 知识，结合实际生产环境，确定虚拟机性能优化的方向，从优化虚拟机的启动速度来提升其部署速度，从而完成目标。最根本是进行内核的优化，从内核入手优化虚拟机操作系统的模块，针对 Docker 的使用依赖项，对内核各模块进行了裁剪和优化，同时对系统的 initrd 中脚本文件

linuxrc 进行优化。

第五章测试裁剪优化过后的系统在自身启动时间上是否得到优化，再在其上跑 Docker，运行之前的任务进行性能测试，收集数据与之前的数据进行比对。分析虚拟机的启动和部署速度是否得到提升，结果是否符合预期。

第六章对本文做总结，分析了虚拟机性能优化所取得的成果和不足，并指出未来工作的方向。

1.5 本章小结

本章介绍了云计算背景及容器技术，阐述了本课题目前国内外的研究现状，具体分析了虚拟机性能优化的方案，明确了本课题研究的重点和目标，最后对全文组织结构进行介绍。

2 Docker 及 Linux 内核

2.1 Docker 简介

2.1.1 Docker 背景介绍

Docker 诞生与 2013 年，是一个跨平台、可移植并且简单易用的容器解决方案。Docker 可以在容器内部快速自动化地部署应用，并通过操作系统内核技术（namespace、cgroups 等）为容器提供资源隔离和安全保障[12]。Docker 选择容器作为核心和基础，迅速博得世界各地各大云计算厂商及开发者手中的青睐。

Docker 通过 Go 语言开发，源码托管在 GitHub 上。从 GitHub 中 commit 的次数来看，Docker 自开源以来热度一直相当的高，见图 2-1。

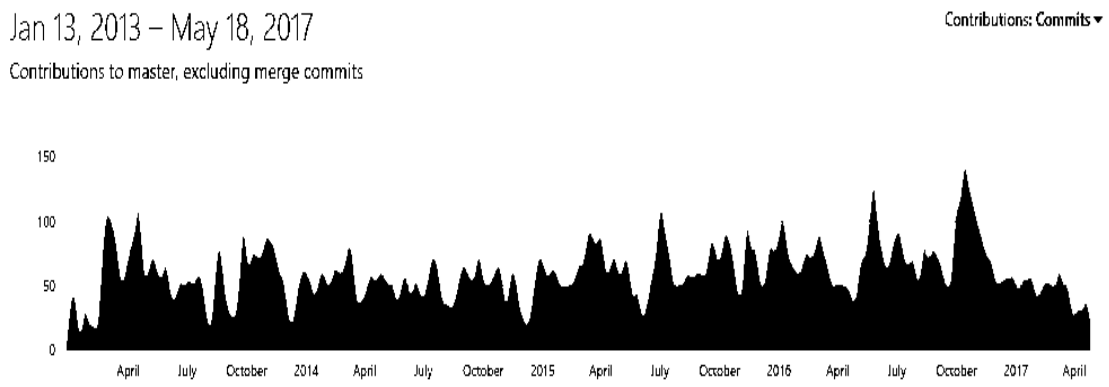

图 2-1　GitHub 上 Docker 的 commit 统计数

截止 2017 年 5 月：GitHub 中 Docker 被标心关注（star）的多达四万三千人次，被 fork 开发分支的次数接近一万三千次，贡献者更是不计其数。

Docker 官方的镜像库也得到使用者和开发者的大力支持，每天都有很多开发者不断向官方的镜像库中提交自己的镜像，作为一个开源项目，这让世界上所有有志于 Docker 项目开发者都处在了一个社区里，更促进了容器技术迅猛发展，使得容器技术更加多样化。

容器技术的出现为用户开辟了一方新大陆，但随着用户在其上初步探索之后，发现容器技术庞大的生态系统给应用带来了相当难度，容器的生态系统如图 2-2。从图中可以得出，容器技术的生态系统涵盖了资源调度、编排、部署、监控、配置管理、存储、网络安全以及安全等。

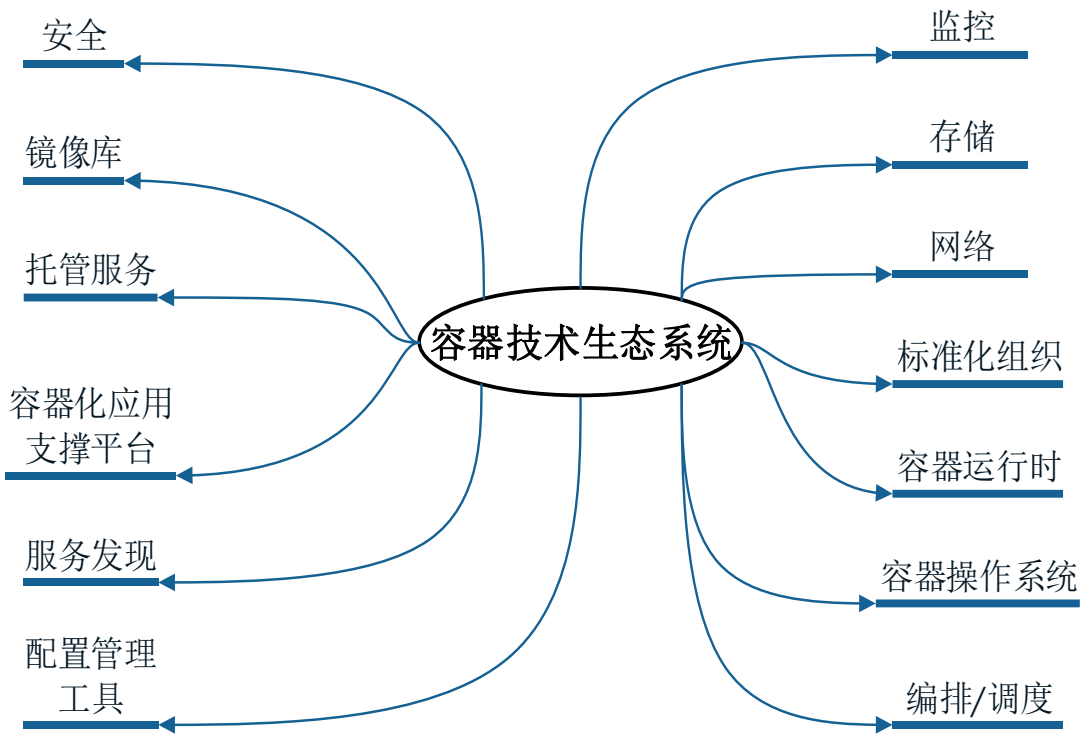

图 2-2　容器技术生态系统

容器技术带来的优势主要有以下几点：

1. 环境标准化与版本控制。相比于代码的 Git 机制，容器环境得到标准化之后，我们可以利用这种机制在发生故障后迅速进行回滚等操作。

2. 跨云平台支持。容器的适配性很强，越来越多的云平台选用它的原因在于其通用性，让应用多平台部署很方便。

3. 高资源利用率及隔离。容器本质上相当于运行在宿主机上的一个进程，与宿主机共享操作系统，可以更加充分的利用系统资源。

4. 容器跨平台性。原先的 Linux 容器由于缺少跨平台性，难以推广。容器在这之上进行了大胆创新，将运行依赖的环境打包成镜像，使其可以在各个平台快速简介的部署。

5. 简单易用。Docker 字面意思是码头工人，它的图标就是一个鲸鱼载着各种集装箱，很容易让人将一个个集装箱理解成一个个 Docker 容器，将鲸鱼理解成容器的宿主机。

6. 易用镜像仓库。目前社区中的镜像仓库都有好几种，管理方式都和 GitHub

很类似，大大方便了开发者。

2.1.2 Docker 架构

Docker 采用的架构模式是 CS 模式，架构如所图 2-3 示。用户通过 client 客户端和服务器 server 端建立通讯，当用户的请求发送给服务端 Docker daemon 之后，后端的各个功能独立但相互紧密结合的模块一起完成用户发来的请求。

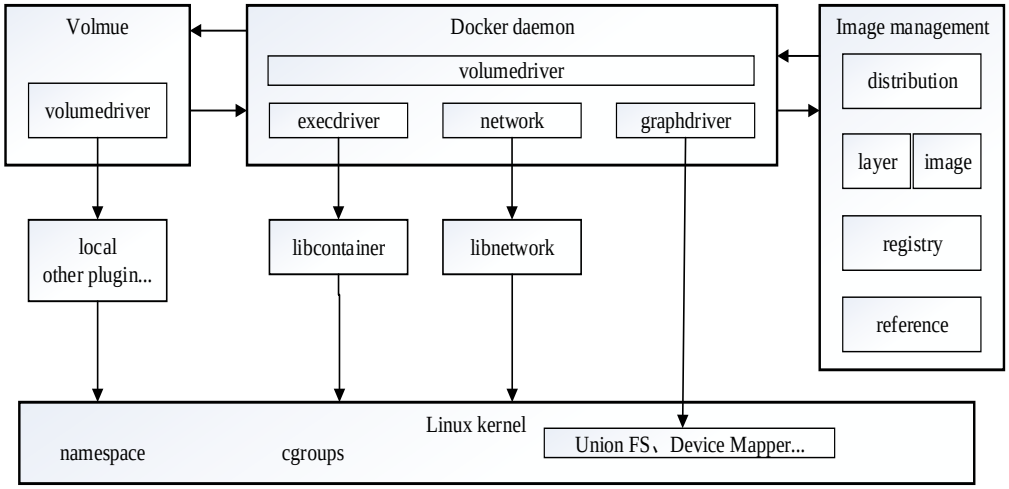

图 2-3 Docker 架构

根据上架构图可以将 Docker 分为五个部分：服务端 docker daemon、用户端 docker client、镜像管理、网络和驱动。

1. Docker daemon

这是 Docker 运行的核心进程，就像 CPU 之于 PC 一样，它负责整个容器接受任务、处理任务等一系列事务的处理。当 Docker daemon 在后台运行时，它会启动一个服务器用来接收用户端发来的请求，并分发调度请求，有具体的函数来执行请求，满足用户的需求。

2. Docker client

这就是 Docker 客户端，用户在这里进行各种操作，每一种操作都会产生服务请求发给服务端，当服务端相应并完成请求之后，用户端也负责接收服务端发送过来的结果数据。目前社区的客户端版本已经十分完善。

3. 镜像管理

Docker 通过 distribution、registry、layer、image、reference 等模块实现了 Docker 镜像的管理，我们将这些模块统称为镜像管理（image management）。

（1） Distribution 负责与 Docker registry 交互，上传下载镜像以及存储与 v2

registry 有关的元数据。

（2） Registry 模块负责与 Docker registry 有关的一系列交互操作，包括身份和镜像的验证等。

（3） Image 模块负责与镜像元数据有关的存储、查找，镜像层的索引、查找以及镜像 tar 包有关的导入、导出等操作。

（4） Reference 负责存储本地所有的镜像的 repository 和 tag 名，并维护与镜像 ID 之间的映射关系。

（5） Layer 模块负责与镜像层和容器层元数据有关的增删查改，并负责将镜像层的增删查改操作映射到实际存储镜像层文件系统的 graphdriver 模块。

4. Network

Libnetwork 抽象出了一个容器网络模型，并给调用者提供了一个抽象接口，其目标并不仅限于 Docker 容器。CNM 模型对真实的容器网络抽象出了沙盒（sandbox）、端点（endpoint）、网络（network）这三种对象，由具体的网络驱动（包括内置的 Bridge、Host、None 和 overlay 驱动以及通过插件配置的外部驱动）操作对象，并通过网络控制器提供一个统一接口供调用者管理网络。

5. Drivers

Docker 也为用户操作系统调用提供了三种驱动，包括容器执行驱动（execdriver）、卷存储驱动（volumedriver）、镜像驱动（graphdriver）。

2.2 Linux 内核架构

2.2.1 Linux 内核简介

Linux 内核是一种以 C 语言和汇编语言写成的计算机操作系统内核。由芬兰人 Linus Torvals 开发而成。

本质上 Linux 系统就是一个内核，但严格来说内核只是操作系统的一部分，它对下承接了计算机硬件层，对上为各种库提供 API 接口。内核能提供硬件抽象层、磁盘及文件系统控制、多任务等功能。然而拥有一个内核并不能被称之为操作系统。Linux 系统指的就是运行 Linux 内核的操作系统。

Linux 内核与 Docker 一样源码开源，托管于 GitHub，在世界各地 Linux 爱好者的贡献下不断发展壮大。

2.2.2 Linux 内核模块

1. 模块概述

Linux 为了高效利用微内核的优点但不降低系统的性能，使用了模块这一方法。通常，系统程序员都偏向于将新代码作为一个模块实现。因为模块可以根据需要进行链接，这样内核就不会因为装载很多很少使用的程序而变得非常庞大。几乎 Linux 内核的每个高层组件——文件系统、设备驱动程序、可执行程序、网络层等等都可以作为模块进行编译。

有些 Linux 代码必须被静态链接，也就是说相应组件或者被包含在内核中，或者根本不需要被编译。典型情况下，这发生在组件需要对内核静态链接的某个数据结构或函数进行修改时。

2. 模块的链接和取消

模块是作为可执行可链接格式（Executable and Linkable Format，ELF）对象文件存在文件系统中的，可以通过执行 insmod 外部程序将一个模块链接到正在运行的内核中，sys_init_module()服务是实际执行者，它执行如下操作：

（1） 首先确定所需要的模块名。

（2） 模块对应的代码一般存放在文件系统相关模块的文件夹中。

（3） 从磁盘读入存有模块目标代码的文件。

（4） 调用 init_module()系统调用，传入参数。

（5） 结束。

对应的取消链接可以使用 rmmod 指令，相应的 sys_delete_module()服务是实际执行者。

3. 按需要链接模块

模块可以在系统需要其提供的动能时自动进行链接，之后也可以自动删除。例如，假设一个 MS-DOS 文件系统没有被静态链接，也没有被动态链接。如果用户师徒装载 MS-DOS 文件系统，那么 mount()系统调用通常就会失败，返回一个错误码，因为 MS-DOS 没有被包含在已注册文件系统的 file_system 链表中。然而，如果内核已经配置为支持模块的动态链接，那么 Linux 就是图链接 MS-DOS 模块，然后再扫描已经注册过的文件系统的列表。如果该模块成功地被链接，那么 moun()系统调用就可以继续执行，就好像 MS-DOS 文件系统从一开始

就存在一样。

4. 内核的编译

（1） 大概步骤：

[1] 安装开发包组，下载源码文件

[2] .config：准备文本配置文件

[3] make menuconfig：配置内核选项

[4] make [-j #]：编译

[5] make modules_install：安装模块

[6] make install ：安装内核相关文件

[7] 编辑 grub 的配置文件

（2） 编译配置选项

配置内核选项：

[1] make config：基于命令行以遍历的方式去配置内核中可配置的每个选项

[2] make menuconfig：基于 curses 的文本窗口界面

[3] make gconfig：基于 GTK (GNOME）环境窗口界面

[4] make xconfig：基于 QT(KDE)环境的窗口界面

[5] make defconfig：基于内核为目标平台提供的“默认”配置进行配置

[6] make allyesconfig: 所有选项均回答为“yes“

[7] make allnoconfig: 所有选项均回答为"no“

（3） 编译

全编译:make [-j #] （-j 表示可选核数）。

（4） 编译内核

没有特殊要求，直接执行 make 指令即可。

（5） 清理删除

在已经执行过编译操作的内核源码树做重新编译，需要事先执行 make clean 指令来清理大多数编译生成的文件，但会保留 config 文件等

2.3 本章小结

本章主要介绍了 Docker 的架构组织、生态系统，详细阐述了 Linux 内核的架

构知识，理解了内核模块的作用以及如何对模块进行链接、卸载等操作，同时介绍了内核编译的相关操作步骤。

3 配置及测试方法

3.1 配置简介

从课题本质入手，面向容器的虚拟机性能优化，那我最先要着手的就是对承载容器的虚拟机进行性能测试，从容器和 VM 的架构区别之处入手，我决定从优化虚拟机的 boottime 入手，缩减 VM 的各个模块，让 Docker 运行所不需要的模块从内核去除，再根据特定场景下 Docker 所需要的模块将其编译成 module，实现动态加载，让整个虚拟机操作系统更加精简，启动时间得到提升，对 Docker 的支持更加全面。

3.1.1 物理机与虚拟机配置

物理机与虚拟机的配置如下表 3-1：

表 3-1 物理机与虚拟机配置

	操作系统	硬盘空间	Core 数	内存	内核版本
物理机	Fedora 25	180G	2	8G	4.4.0
虚拟机	CentOS-7-x86_64-Minimal	20G	2	2G	3.10.0

物理机操作系统选用的是 Fedora 25，是目前 Linux 具备 UI 界面发行版中很受个人用户欢迎的版本，它简单易用，具有很强的稳定性，能为各类开发者和创客提供整套开发工具。

在虚拟机操作系统的选择上做了很多考究，首先针对实际的应用场景，虚拟机 OS 在物理机上部署时，肯定不是一个虚拟机那么简单，涉及到大规模的集群部署时，虚拟机 OS 的大小肯定有决定性的作用，这决定了一台物理机部署的虚拟机台数，从而决定了它的性能和资源利用率。

因此，在选用虚拟机 OS 的决定上，我经过诸多比较，决定选用 CentOs-7-minimal，这是基于 CentOS7 的最精简发行版。CentOS 是 Linux 发行版之一，它是由 Red Hat Enterprise Linux 开放的源码编译而成。因为源代码相同，因此很多个人用户倾向于它的高效稳定性而选用其作为 RHEL 的替代。相较于 CentOS7 完整版 1.2G 的 ISO，精简版大小只有 700M 不到，因此选用更为精简的虚拟机系统镜像可以更好的提升实际部署时的性能。

3.1.2 容器镜像选取

为比较不同发行版 Linux 在 Docker 容器中的运行性能，我选择了 Docker Hub 官方的 fedora 和 ubuntu 镜像，如下图 3-1：

```
[root@localhost /]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker.io/fedora    latest              15895ef0b3b2        4 weeks ago         230.9 MB
docker.io/ubuntu    latest              6a2f32de169d        5 weeks ago         117.2 MB
```

图 3-1 容器镜像

3.2 虚拟机性能监控

想要对虚拟机的性能进行优化，必须先进行性能测试，虚拟机在物理机上大规模部署时，最直观反映其性能的就是响应速度，例如当一个使用手机 app 的用户需要使用云服务时，提供相关云服务的虚拟机的启动时间越快，则容器启动到准备好客户所需要的服务的时间会越短，这样可以大大提升云服务的质量。

3.2.1 Linux 内核启动相关测试

下面是通过代码时间戳、工具和系统内置命令等方法来测试 Linux 启动时间的测试结果。

1. printk.time

“printk.times”是一种简单的时间戳技术，它将代码添加到标准的内核 printk 例程中，以便每个消息输出定时数据。虽然不是很精确，但可以用来了解内核初始化的时间相对较长的区域。启动时间工作组使用此功能来确定需要进行的工作 Linux 内核区域，以提高启动时间，并衡量工作组所做更改的优化程度。

该功能的技术包括补丁和实用程序，补丁会更改内核中的 printk 代码以发出计时数据。

在打印时间打开的情况下，系统会在打印机启动的时间内以时间数据（以微秒为单位）发出定时数据。实用程序显示呼叫之间的时间，或者可以显示相对于特定消息的时间。这使得更容易看到内核代码的特定段的时间。

2. bootchart2

bootchart2 是一个由 python 编写的启动项监测工具，当安装 bootchart2 后，修改内核启动的配置文件 grub2.cfg，如图 3-2，使得引导过程中运行此脚本而不

是 init，并测量此过程中的时间。

```
### BEGIN /etc/grub.d/00_header ###
initcall_debug printk.time = y quiet init = /sbin/bootchartd_
set pager=1
```

图 3-2 修改 grub2.cfg

测试数据保存在/boot/var/log 目录下，使用其提供的组件 bootchartd 可以将收集的 log 数据生成图表，显示每个进程何时启动和停止，并记录了每个进程的名字及启动耗费的时间。运用 bootchart2 可以非常直观的看出内核启动过程中都加载了哪些 module 以及各个 module 的启动时间。可以与 printk.time 测得数据相关联，得出更精确的启动时间。

3.2.2 Linux 模块管理命令

如 2.3.2 章节所述，为提高系统性能，充分利用 Linux 微内核的特点，开发者将很多系统功能编译成模块，下面介绍一下在实验过程中用到的模块管理命令。

1. Lsmod：等效指令是 cat /proc/modules，列出已经加载的内核模块。
2. Modinfo：该指令查看模块的信息，明确模块作用。
3. modprobe：挂载新模块以及新模块相依赖的模块。
4. rmmod：移除已挂载模块。
5. insmod：挂载模块，相较没有 modprobe 功能强大。
6. 与内核模块加载相关的配置文件

模块的配置文件 modules.conf 或 modprobe.conf。Linux 内核启动是所挂载的模块信息都存放在配置信息中。

3.3 容器性能监测

Docker 的出现给人们带来了一个部署应用的新平台，随着它的迅猛发展，开发者们也将目光转向了容器运行的性能方面，因此对容器进行性能监控显得尤为关键。下面简单介绍我在实验中用到一些工具。

3.3.1 Docker 性能监控指令

Docker 自带了很多在实验过程中主要用到了几个指令：

1. docker ps 命令

列出当前主机上的容器信息，包括容器 ID、镜像名、创建时间、状态、端口信息和容器名字。

2. docker images 命令

如图 3-1 所示，该指令可以查看当前主机上的所有镜像信息，包括 pull 下来的和自己 commit 的镜像，详细显示镜像大小标签等属性。

3. docker stats 命令

用于统计容器的状态信息，可以实时监控容器的 CPU、内存、块设备 I/O 和网络 I/O 信息，并定期刷新显示在命令行。和监控工具比较起来，自带的功能虽然强大，但是记录和实时反馈的效果还是不如工具。

4. docker inspect 指令

这条指令可以说是最全面的一条指令的，它将容器底层的详细配置信息全部列出来，包括它的基础配置、网络环境、状态信息等，当我需要在容器中运行某一项特殊的网络服务时，就可以从该条指令中获取这个 Docker 容器的相关网络信息如端口地址等等。

3.3.2 Datadog Agent

Datadog Agent 是一款运行在主机上的轻量级软件。它可以在后台运行，收集系统的各项事件和性能指标，生成可视化视图，通过网页用户可以观测这些数据，从而监测系统性能。

Datadog Agent 提供了监控大规模集群、云主机、Docker 容器，支持多种操作系统、数据库等，在数据采集、计算和展现之上，还拥有事件流展现、报警功能。

3.4 本章小结

本章在明确课题目标的基础上，对比选取测试环境，包括物理机虚拟机配置、容器镜像选取等，之后对性能测试的工具和技术进行深入学习，分别使用各种方法对虚拟机性能和 Docker 容器性能进行测试，收集优化前的各项测试数据，主要从 CPU 使用率、系统平均负载、磁盘 I/O 和网络传输方面来分析。

4 虚拟机性能优化步骤

4.1 虚拟机内核启动项分析

1. printk.time 测试数据

在 Linux 内核版本高于 2.6.11 的版本下运行，在启动系统的时候，在命令行加上“printk.time = 1”即可将时间戳打印在每个启动项之前。启动后通过命令“dmesg”收集 printk 数据，具体指令 dmesg > time。在通过内核文件夹/scripts/show_delta 的脚本文件将将收集的时间更加细化，可以再每一行前显示当前操作耗费的时间，通过数据重定向在输出到 boottime，具体指令 show_delta < time > boottime。

虚拟机的 CentOS7 的内核版本为 3.10.0，经多次启动后测得 boottime 文件部分截图如下图 4-1，两个时间分别表示执行到该步操作经历的总时间和该步操作所耗费的时间，在 vim 定位处“Freeing unused kernel memory：1609K\n”，为内核初始化结束的标志。

```
('[0.789240 < 0.000007 >] usbserial: USB Serial support registered for generic\n',)
('[0.789326 < 0.000086 >] i8042: PNP: PS/2 Controller [PNP0303:KBC,PNP0f13:MOUS] at 0x60,0x64 irq 1,
12\n',)
('[0.790447 < 0.001121 >] serio: i8042 KBD port at 0x60,0x64 irq 1\n',)
('[0.790454 < 0.000007 >] serio: i8042 AUX port at 0x60,0x64 irq 12\n',)
('[0.790635 < 0.000181 >] mousedev: PS/2 mouse device common for all mice\n',)
('[0.791417 < 0.000782 >] rtc_cmos 00:01: rtc core: registered rtc_cmos as rtc0\n',)
('[0.791590 < 0.000173 >] rtc_cmos 00:01: alarms up to one month, y3k, 114 bytes nvram, hpet irqs\n'
,)
('[0.793048 < 0.001458 >] input: AT Translated Set 2 keyboard as /devices/platform/i8042/serio0/inpu
t/input1\n',)
('[0.793637 < 0.000589 >] input: VirtualPS/2 VMware VMMouse as /devices/platform/i8042/serio1/input/
input4\n',)
('[0.794017 < 0.000380 >] input: VirtualPS/2 VMware VMMouse as /devices/platform/i8042/serio1/input/
input3\n',)
('[0.795385 < 0.001368 >] hidraw: raw HID events driver (C) Jiri Kosina\n',)
('[0.795456 < 0.000071 >] usbcore: registered new interface driver usbhid\n',)
('[0.795456 < 0.000000 >] usbhid: USB HID core driver\n',)
('[0.795614 < 0.000158 >] Initializing XFRM netlink socket\n',)
('[0.795718 < 0.000104 >] NET: Registered protocol family 10\n',)
('[0.796098 < 0.000380 >] Segment Routing with IPv6\n',)
('[0.796108 < 0.000010 >] NET: Registered protocol family 17\n',)
('[0.796312 < 0.000204 >] registered taskstats version 1\n',)
('[0.796313 < 0.000001 >] Loading compiled-in X.509 certificates\n',)
('[0.797959 < 0.001646 >] alg: No test for pkcs1pad(rsa,sha256) (pkcs1pad(rsa-generic,sha256))\n',)
("[0.798637 < 0.000678 >] Loaded X.509 cert 'Build time autogenerated kernel key: e8458161dd6cc0dd7f
df5b363d262285fab4b6e4'\n",)
('[0.798661 < 0.000024 >] zswap: loaded using pool lzo/zbud\n',)
('[0.808850 < 0.010189 >] Key type big_key registered\n',)
('[0.810973 < 0.002123 >] Key type trusted registered\n',)
('[0.814007 < 0.003034 >] Key type encrypted registered\n',)
('[0.814011 < 0.000004 >] ima: No TPM chip found, activating TPM-bypass! (rc=-19)\n',)
('[0.814056 < 0.000045 >] evm: HMAC attrs: 0x1\n',)
('[0.817995 < 0.003939 >] rtc_cmos 00:01: setting system clock to 2017-05-11 08:57:29 UTC (149449304
9)\n',)
('[0.818967 < 0.000972 >] Freeing unused kernel memory: 1609K\n',)
                                                                1351,60       88%
```

图 4-1 boottime 部分测试结果

2. bootchart2 测试结果

通过配置 bootchart2，在启动项中添加 bootchart 组件收集启动项和启动时间，结果如图 4-2，图中蓝色部分是组件所产生的耗费，可以看出组件耗费时间加起来对时间测试还是有影响的，但是 bootchart2 很准确地收集了内核在启动期间的各个启动项。可以与 printk.time 收集的数据结合起来进行分析。

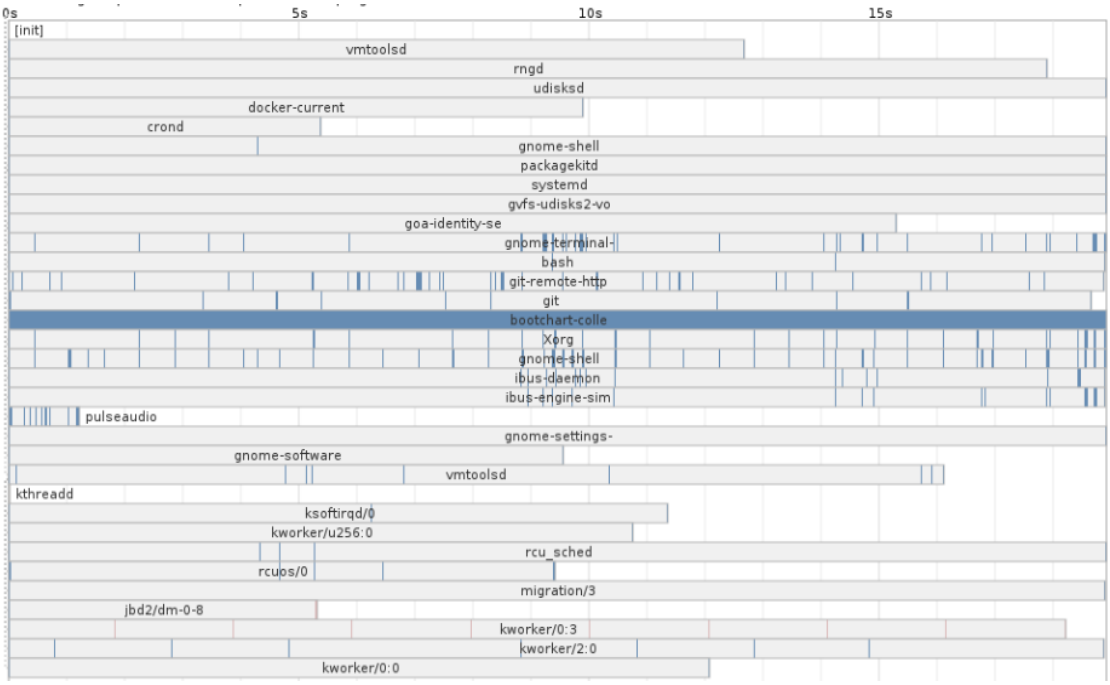

图 4-2 bootchart2 测试数据

5. 数据分析

结合 bootchart2 测得图像，阅读 boottime 文件可以初步分析出如下结果：

（1） 1～408 行启动总时间为 0.000000s：这一过程主要操作是 OS 从 ACPI 表中取得寄存器信息，OS 的 AML 解释器解释执行 AML 代码，读取多处理器配置信息（MADT）、NUMA 配置信息（SRAT、SLIT）等信息。

（2） 409～464 行总启动时间为 0.092238s：这一过程主要是挂载进程哈希表（PID hash table）、初始化 SELinux 组件、按需挂载 ACPI 表。

（3） 465 ~ 1276 行总启动时间为 0.650559s：这一过程主要是初始化 PCI 总线，挂载 pci_bus、pcieport 等驱动。

（4） 1276 ~ 1353 行启动时间为 0.168749s：这一过程是初始化 usb driver，到这一步完成时，内核的启动完成。

（5） 1277 ~ 1531 行总耗费时间为 33.348506s：这一段时间就是挂载各种

模块个驱动，如加载 SELinux 模块、初始化网卡等。

4.2 虚拟机内核模块分析

在 4.1 章节中我们对虚拟机内核启动项及其耗时进行了仔细的分析，通过对收集到的启动项日志 boottime 的分析，我们可以进一步来确定针对 Docker 的内核模块精简方案，这一小节内容是将启动项的具体分析表列出来，筛选出在启动过程中占据大量时间的启动项，再分析其与 Docker 的依赖关系，从而进一步寻找优化该启动项的方案。

4.2.1 启动项耗时分析

对启动项日志 boottime 进项逐行分析之后，取出耗时量大的启动项操作进行具体比对，如表 4-1。

表 4-1 boottime 分析

总耗时<耗时	具体操作
('[0.670842 < 0.274363 >]	Freeing initrd memory: 19784K\n',)
('[1.097196 < 0.241483 >]	usb 2-1: new full-speed USB device number 2 using uhci_hcd\n',)
('[1.283262 < 0.152542 >]	FUJITSU Extended Socket Network Device Driver version 1.2 - Copyright\n'）
('[2.040404 < 0.154262 >]	usb 2-1: New USB device found, idVendor=0e0f, idProduct=0003\n',)
('[2.284108 < 0.129853 >]	usb 2-2: New USB device found, idVendor=0e0f, idProduct=0002\n',)
('[2.580774 < 0.287216 >]	SGI XFS with ACLs, security attributes, no debug enabled\n',)
('[2.743259 < 0.159033 >]	XFS (dm-0): Ending clean mount\n',)
('[3.134521 < 0.391262 >]	systemd-journald[135]: Received SIGTERM from PID 1 (systemd).\n',)
('[3.925067 < 0.584710 >]	SELinux: 32768 avtab hash slots, 104710 rules.\n',)
('[4.483597 < 0.454862 >]	ip_tables: (C) 2000-2006 Netfilter Core Team\n',)
('[7.368422 < 2.868098 >]	systemd-journald[538]: Received request to flush runtime journal from PID 1\n',)
('[8.297472 < 0.929050 >]	shpchp: Standard Hot Plug PCI Controller Driver version: 0.4\n',)
('[8.615189 < 0.303103 >]	Floppy drive(s): fd0 is 1.44M\n',)

续表 4-1

('[9.328425 < 0.226803 >]	ppdev: user-space parallel port driver\n',)
('[9.802030 < 0.381788 >]	AVX version of gcm_enc/dec engaged.\n',)
('[11.425187 < 1.227307 >]	audit: type=2000 audit(1494493048.590:1): initialized\n',)
('[16.483706 < 4.850277 >]	ip6_tables: (C) 2000-2006 Netfilter Core Team\n',)
('[17.638831 < 0.224376 >]	nf_conntrack version 0.5.0 (16384 buckets, 65536 max)\n',)
('[17.992210 < 0.353379 >]	bridge: filtering via arp/ip/ip6tables is no longer available by default. \n',)
('[18.444496 < 0.452286 >]	IPv6: ADDRCONF(NETDEV_UP): ens33: link is not ready\n',)
('[32.371510 < 13.844570 >]	loop: module loaded\n',)
('[33.697281 < 1.325771 >]	Bridge firewalling registered\n',)

4.2.2 容器在虚拟机下性能分析

以下分别从 CPU 使用率、系统平均负载、磁盘 I/O 速率和网络传输速率来分析 Docker 在虚拟机下的性能表现。任务容器的性能由 Datadog Agent 所跑的容器 docker-dd-agent 收集，可以在监测界面上观测到相关性能数据。

在容器内运行 Fedora 镜像、Ubuntu 镜像和 docker-dd-agent 镜像，在 Fedora 容器内使用脚本运行 yum update、yum install python 和 yum install git 三个指令，其余两个容器处于挂起状态，测试 10min 内系统的各性能如下：

1. CPU 使用率分析

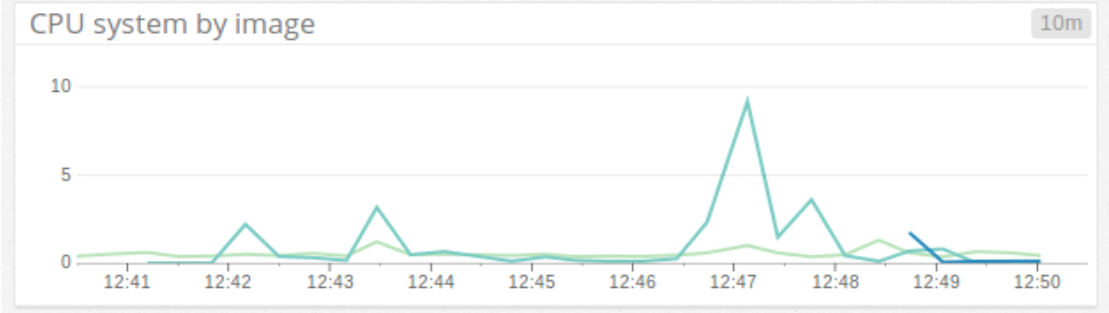

图 4-3 虚拟机 CPU 使用率

如图 4-3，淡绿色线表示 docker-dd-agent，运行在 Docker 容器中，用来向 datadog 传输测试数据的进程，淡蓝色表示 Fedora 容器在启动后 CPU 使用率，

运行脚本的两分钟时间内占用率峰值达到了 10%，深蓝色表示后来启动的 ubuntu 容器，发现在启动的十几秒内，CPU 使用率达到了 2%左右。

2. 用户使用负载分析

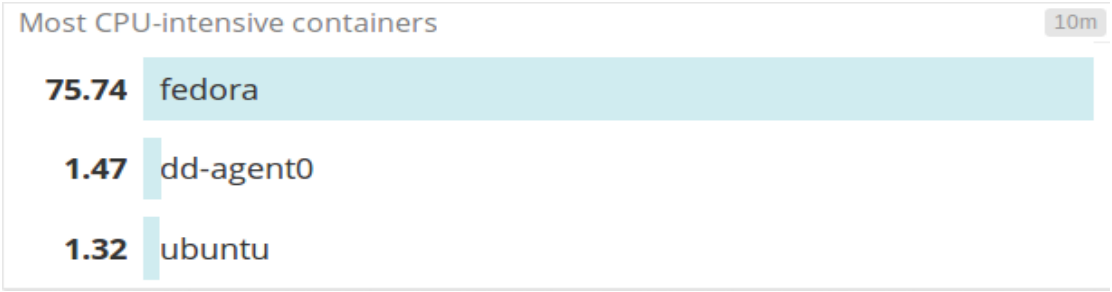

图 4-4 用户使用负载分析

如图 4-4 所示，docker-dd-agent 监控程序和 ubuntu 容器启动后处于后台挂起状态，用户未进行操作，此时两个容器对 cpu 负载很小，而进行大量 I/O 操作的 feodra 容器的用户使用负载相当高。

3. 磁盘传输速率分析

如图 4-5 所示，深蓝色表示 Fedora 容器，发现其读写操作速率都不是很理想，读操作速率峰值约为 1.5M/s，写操作速率峰值约为 3M/s，写操作一般会落后于读操作，蓝色和黄色分别表示 docker-dd-agent 和 ubuntu 容器，docker-dd-agent 有较明显的读操作，是用与读取 Fedora 容器的各操作数据，而蓝色和黄色的写操作的速率都很低可以忽略。

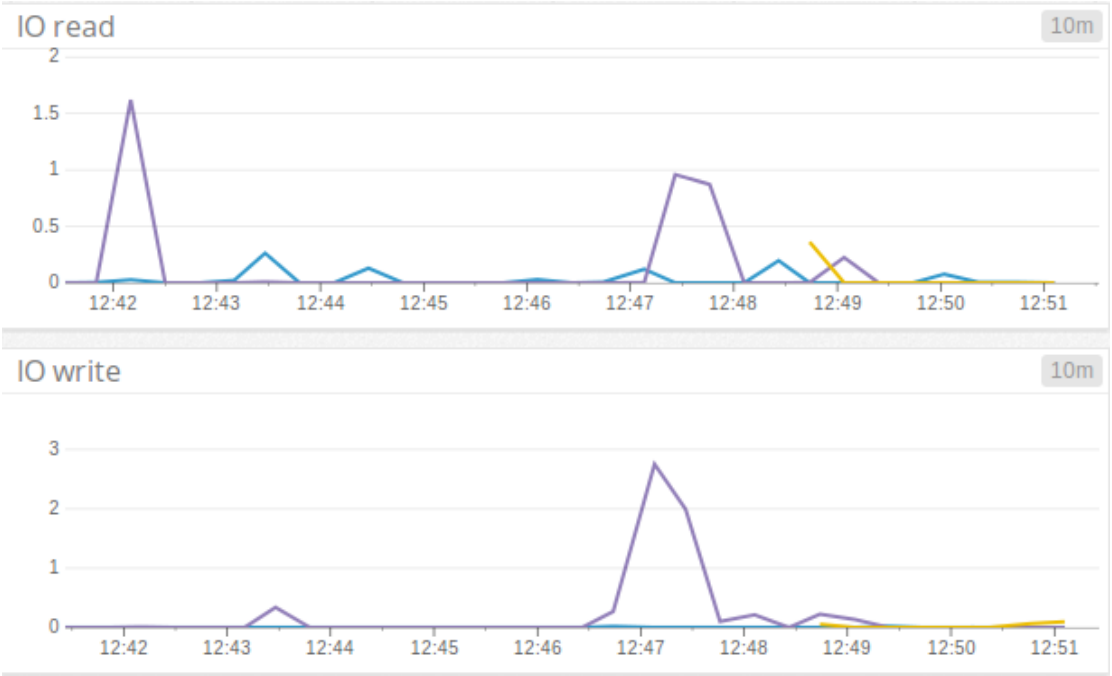

图 4-5 磁盘传输速率分析

4. 网络传输速率分析

如图 4-6 所示，网络传输数据方面，接受和发送数据都与进行的操作相关，波峰波谷与网络速度直接相关，关注紫色区域容器发出数据，紫线表示发送数据总速率，紫色块状区域则表示 fedora 容器发送数据速率，推测之间的灰色区域差值大部分用于 docker-dd-agent 向外发送收集的数据，可以消除此误差得到更精确结果。

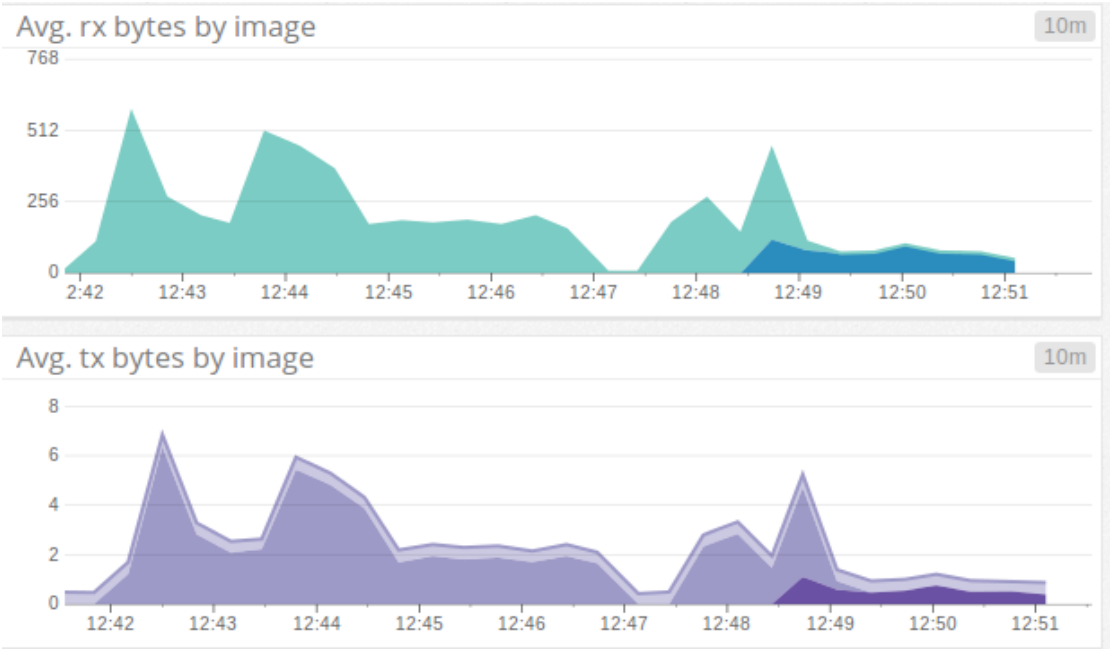

图 4-6 网络传输数据分析

4.2.3 分析依赖关系

根据 2.2 章节介绍的 Docker 架构相关知识，结合 4.2.2 章节对 Docker 在虚拟机下性能的测试结果，对 boottime 中启动时间耗费巨大的操作进行分析，得出以下的结果：

1. 内核启动时间大部分用于初始化驱动和模块
2. 挂载 usb 驱动等外部接口驱动对于实际环境下部署 Docker 没有意义
3. SELinux 等模块的挂载对 Docker 的意义不大但占据大量的启动时间
4. 初始化 RAM 磁盘释放内存操作耗时过长
5. 内核与时钟同步操作（Real Time Clock，RTC）可以延后进行
6. 网卡的初始化占据了内核启动的大部分时间

4.2.4 优化方案

（1） 对于如 usb 驱动这样 Docker 完全不需要依赖的驱动，可以完全在内核启动时不去初始化，或者在内核编译时直接将 usb 模块不编译。

（2） SELinux 安全模块启动耗时明显，但其影响对于 Docker 运行可有可无，可以选择编译内核时就用该模块，或者动态加载该模块，让用户选择是否启用该功能。

（3） 精简 initrd 中的启动脚本 linuxrc

（4） 考虑 RTC 时钟同步操作延后的优化方案，可以屏蔽启动过程中的同步操作，让它在启动后进行，或者去掉该过程。

4.3 优化虚拟机内核配置

根据 4.2 章节提出的（1）、（2）优化方案，先对 Linux 内核进行优化，将不需要的内核功能禁用，尽可能多的将需要的功能编译成模块，方便之后加载进行测试。内核选用的是 www.kernel.org 下载的最新内核版本 linux-4.10.11。

4.3.1 make menuconfig 简介

make menuconfig 是一种对内核进行配置的命令，在字符终端下，它可以让用户基于文本选单进行界面配置，方便删除不必要的文件和目录，之后将用户的设置保存在.config 文件中。进入 mennuconfig 配置界面后如下图 4-7：

```
[*] 64-bit kernel
    General setup  --->
[*] Enable loadable module support  --->
-*- Enable the block layer  --->
    Processor type and features  --->
    Power management and ACPI options  --->
    Bus options (PCI etc.)  --->
    Executable file formats / Emulations  --->
[*] Networking support  --->
    Device Drivers  --->
    Firmware Drivers  --->
    File systems  --->
    Kernel hacking  --->
    Security options  --->
-*- Cryptographic API  --->
[*] Virtualization  --->
    Library routines  --->
```

图 4-7 menuconfig 配置

使用 make menuconfig 进行配置时共有三个选择方式：Y 表示将该功能放入

内核，N 表示不将该功能放进内核中，M 表示将该功能编译成模块动态加载。

4.3.2 内核配置优化

根据 menuconfig 界面提供的信息，我们对[*]built-in 的部分进行部分修改，对于其余部分不需要的直接不选或进行模块化，以下是具体优化配置：

1. 常规内核选项

进入该选项后，界面如图 4-8 所示：

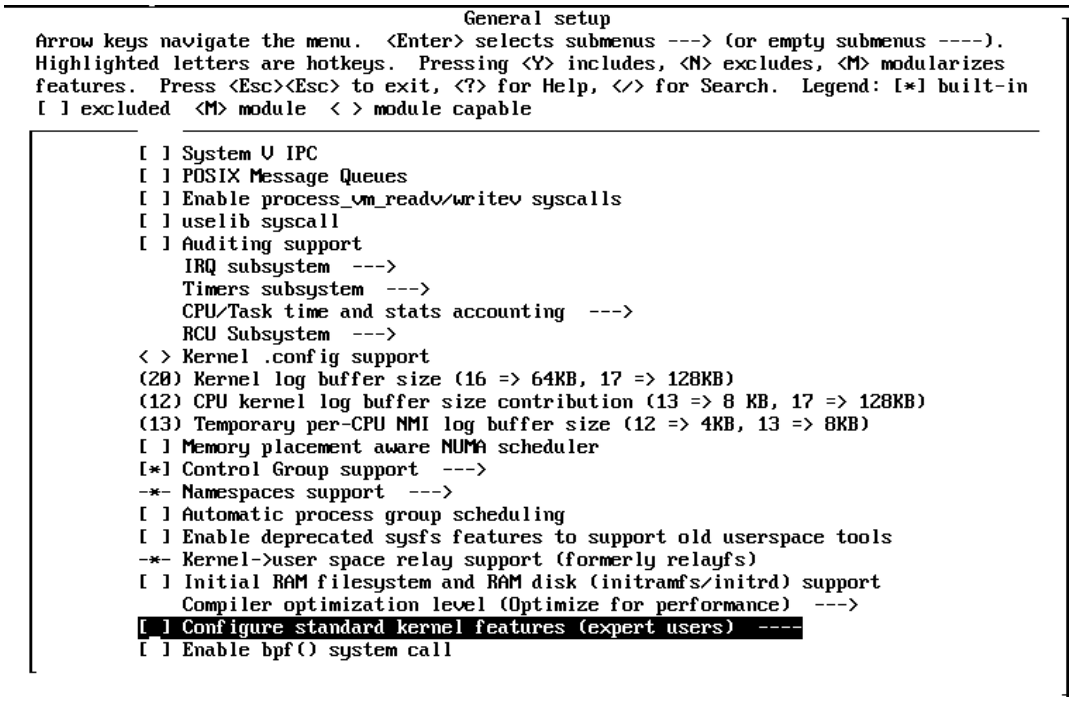

图 4-8 general setup

该选项中默认的 built-in 内容只有 cgroups support、namespace support 和 user space relay support，Docker 是通过与宿主机共享内核的方式来工作的，因此资源的管理都是与 Linux 内核方式一致的，所以这里保持默认选项。

2. 对模块的支持

Linux 内核模块在之前 2.3.2 中已经详细介绍过，我要对内核编译过程中大量模块进行精简必须要启用该模块，进入该选项界面如图 4-9 所示。

这一部分我去除了 Automatically sign all modules（自动挂载所有模块），安装卸载模块所使用的空间可以重新分配并得到利用，但是编译和挂载模块同样需要消耗大量内存，禁用自动注册所有模块是可以将内核的大量内存拿出来进行其他操作，而不是耗费在挂载不必要的模块上。

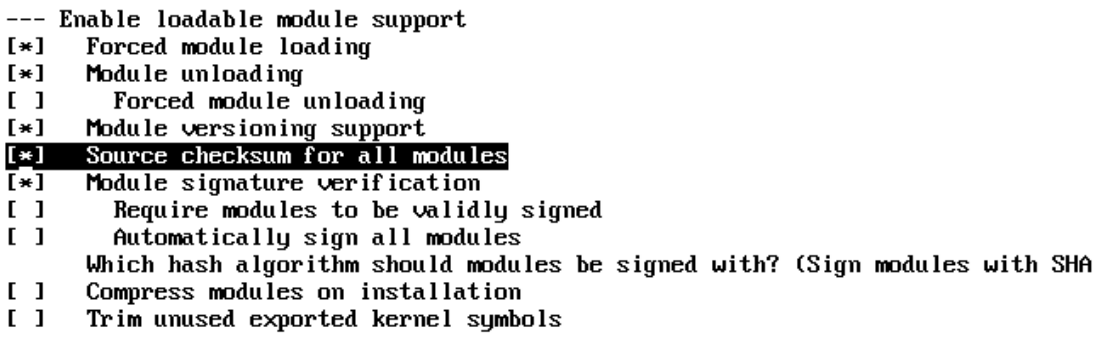

图 4-9 module support 配置

3. 处理器类型和特色

该部分介绍了各种处理器的类型和特色，可以对各种情况下处理器的行为动作作出判断，配置界面图 4-10 如所示：

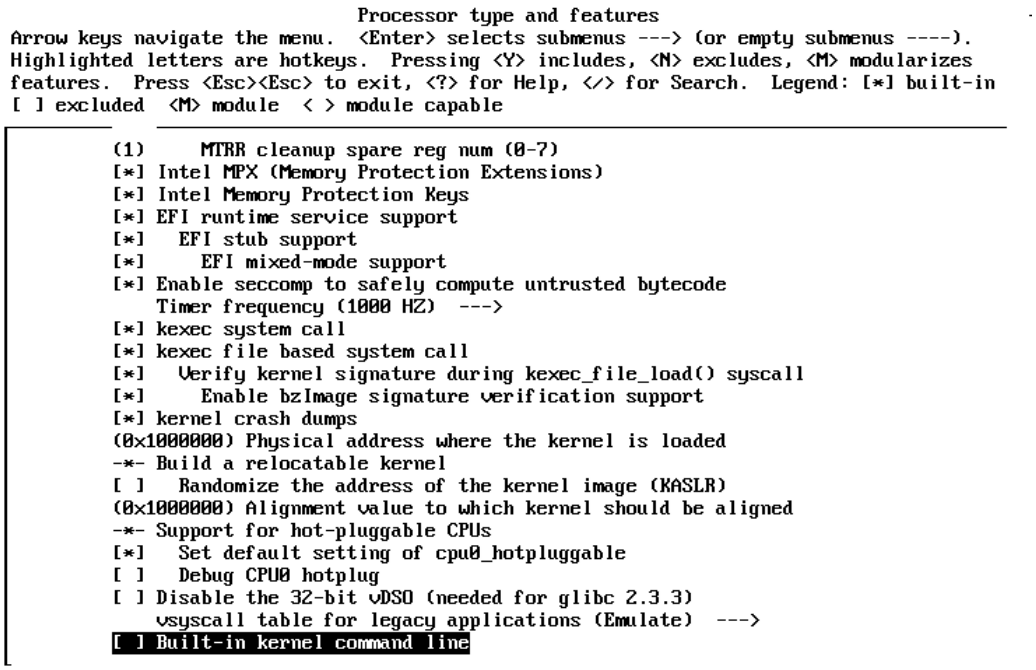

图 4-10 处理器类型特色配置

将 Processor family 选择为 Gerneric-x86-64，这样可以使 CPU 做最佳优化，使它运行得更快更稳定，可能编译出来的内核体积会大一些但性能会更好，根据实际 Docker 应用场景，还可以选择将 High memory support（高内存支持）、Symmetric multi-processing support（多核处理器支持）等功能关闭，这些都是根据你所分配的虚拟机内存以及核数来确定，根据实验用虚拟机配置，内核相关配置未做更多改动。

4. 网络配置

网络配置里主要关注的是 Network options 选项，在 support 界面我去掉了蓝牙子系统（Bluetooth subsystem support）、广播频率子系统（RF switch subsystem Support）、近场通信设备支持（NFC subsystem support）和无线网（Wireless）等相关模块和功能。

之后进入 Network options 界面，如图 4-11，对 TCP/IP 相关功能进行配置，相较于外部接口设备，网络协议层的功能很复杂，修改必须基于对相关代码的了解才能进深入展开，因此不做过多修改。

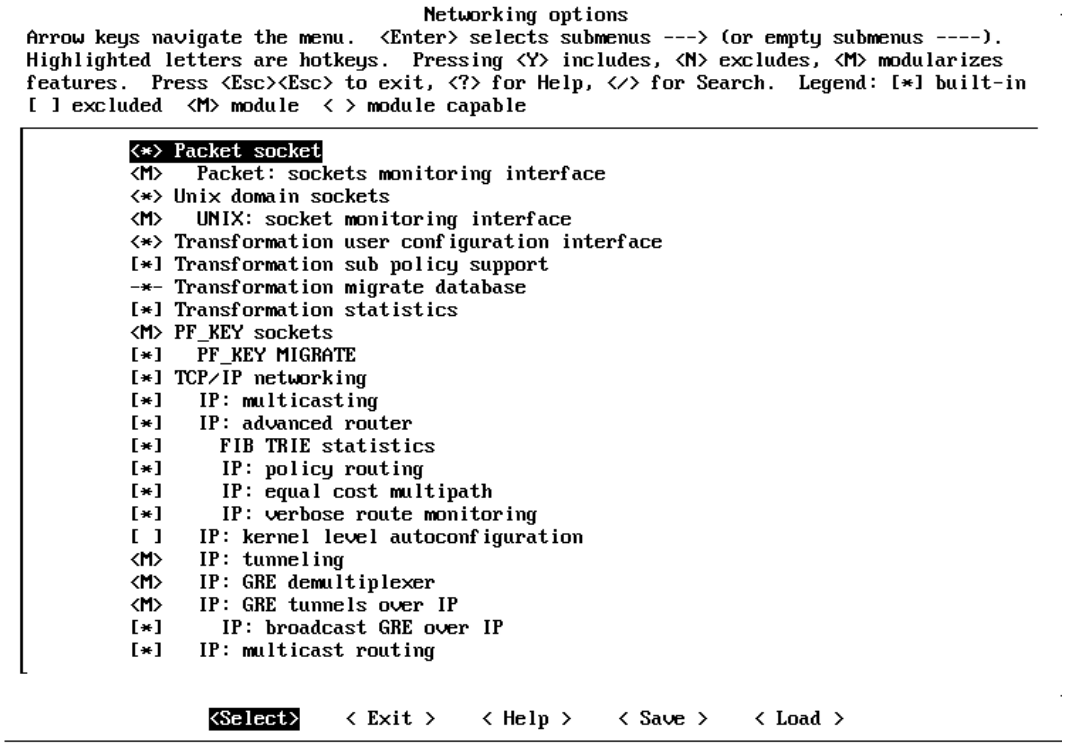

图 4-11 网络选项细节

（1） Network packet filtering 涉及到防火墙的开关，内核级的防火墙固然很优越，但是之前测试数据 SELinux 模块对内核的占用过多，于是决定不去开启，而实际情况下用户一般都是默认禁用 SELinux 的。

（2） 网络配置的过程和操作都非常的繁杂，因为都是对涉及网络安全传输的选项，且大多数都是模块“M”安装，因此其余部分没有做过多细化，保留原始配置。

5. 设备驱动

设备驱动界面如图图 4-12，针对 4.2.3 节提出的优化方案，禁用了 USB support，在对设备驱动列表分析之后，启用了 Parallel port support（并行接口支

持），禁用了 Multimedia support（多功能媒体支持）、Sound card support（声卡支持）等驱动，禁用的驱动都是 Docker 在虚拟机中运行中所不需要的驱动，如声卡驱动、usb 驱动等，因次可以再内核编译阶段就直接去除。

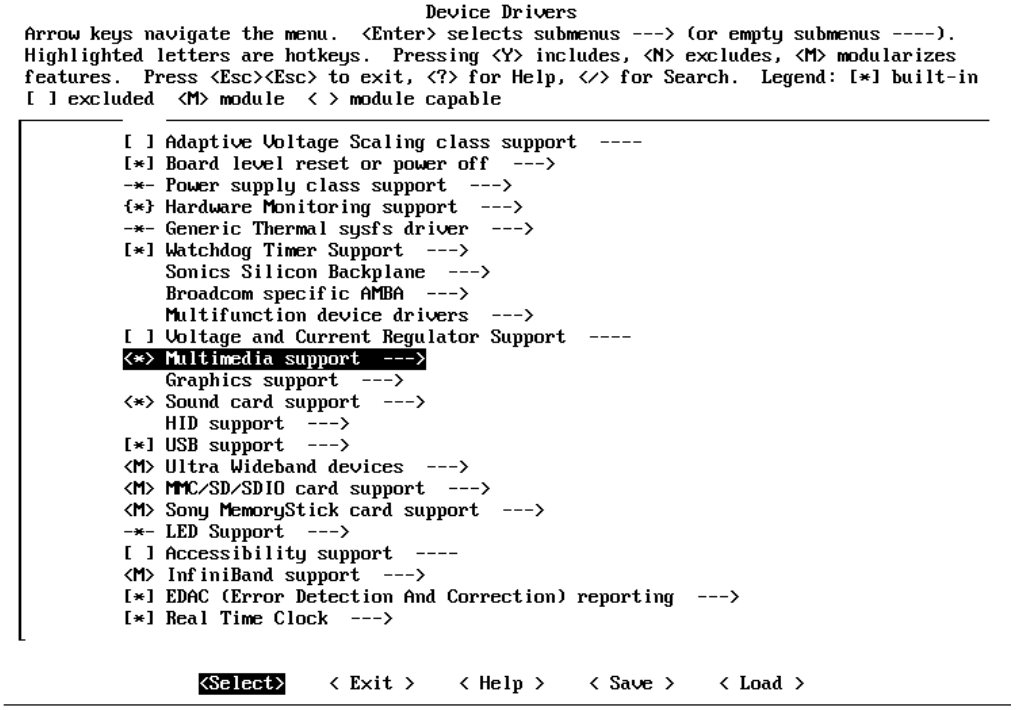

图 4-12 设备驱动配置

这里注意到了 Real Time Clock（RTC），是上面 RTC 提到的内核时钟同步功能，在这里不赘述，具体在 4.5 章节介绍。

6. 内核启动监测

这部分需要设置 printk and dmesg options，进入该模块后选项如图 4-13，其余的都禁用即可，我们需要 printk time 和 dmesg 来提取新内核的启动时间，而不需要再重新配置时间戳函数。当然如果追求内核的极限启动，可以将 printk 函数直接在内核编译时去除。

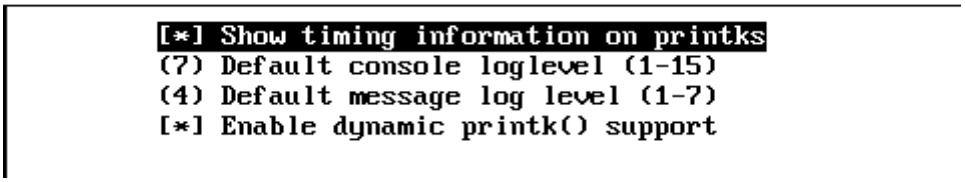

图 4-13 printk 和 dmesg 配置

7. 安全选项

安全选项界面如图 4-14 所示，这部分是对内核的安全模块进行配置，主要是选择是否应用安全加密的文件系统等操作，SELinux 模块的选择项在这一部分，

设置为 disable。

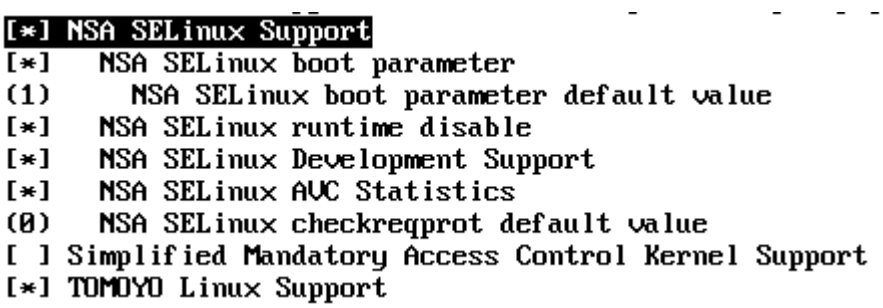

图 4-14 SELinux 模块配置

8. 其他配置

（1） Enable block layer：启用块层，包括添加对块设备的写回支持等功能，保持默认配置；

（2） Power management and ACPI options：电源管理和 ACPI 选项，保持默认配置；

（3） File systems：文件系统，保持默认配置；

（4） Virtuliazation：包括启用 KVM 核心模块等，保持默认配置。

（5） Crypto API：Crypto API 是 Linux 内核中的加密框架，适用于处理加密的内核的各个部分，保持默认配置。

4.3.3 新内核编译

（1） 退出 menuconfig 后，所有配置保存在.config 文件中。

（2） 执行 make 命令，进项内核编译，单核虚拟机系统的编译时间平均为一个半小时左右。

（3） 执行 make module_install 命令安装模块 。

（4） 执行 make install 指令安装内核相关文件，将编译之后的文件拷贝到相应的目录。

（5） 配置引导文件 grub2.cfg，重新启动系统选用新内核，如。

4.4 重制 initrd

分析 boottime 启动时间过程中，在内核启动完成之前有一个操作：Freeing initrd memory: 19784K\n',) 花费了相当长的时间，耗费时间为 0.274363s。initrd 全

称为 initial ram disk，是一个提供启动过程中所需要的内核模块，缺少它系统将无法正常启动。

4.4.1 initrd 和 linuxrc

在系统启动的引导过程中，内核会检测到存在 initrd 映像，将其从 RAM 中的指定物理地址处将这个文件复制到内核 ramdisk 中，并将其挂载为根文件系统（/）。当内核挂载这个模块初始 ramdisk 时，它会查找并执行 linuxrc 脚本文件，当该脚本文件执行完毕后，内核会卸载 initrd 并继续进行系统开机的引导[13]。一个以 initrd 作为根文件系统的 Linux 系统不会尝试将其他文件系统挂载为根文件系统。根据这一特性，开发者可以重制 initrd。

4.4.2 优化 linuxrc

BusyBox 是一个以自由软件形式发行的应用程序。Busybox 在单一的可执行文件中提供了精简的 Unix 工具集，因此我思考是否可以采用 busybox 的 linuxrc 替换原系统的该文件，提供最精简的指令集即可。

BusyBox 可以被自定义化以提供一个超过两百种功能的子集。在 BusyBox 的网站上可以找到所有功能的列表。使用 busybox 官网提供的最新版本如图 4-15。

图 4-15 busybox 版本信息

下载 busybox 后，使用 make enuconfig 操作将 busybox 编译成内核功能模块用生成的 linuxrc 文件替换原系统中文件，重启查看 initrd 释放内存的操作时间是否得到优化。

4.5 RTC 驱动优化

根据 4.2.4 章节提出的优化方案（4），考虑 RTC 时钟同步操作延后的优化方案，可以屏蔽启动过程中的同步操作，让它在启动后进行，或者去掉该过程。

4.5.1 直接从内核中禁用 RTC

如图 4-12 设备驱动列表中显示，在 Real Time Clock 前[*]选 N，让内核不对

RTC 驱动进行编译，可以直接去掉该过程。

4.5.2 采用 RTCNoSync

在内核启动过程中可能需要很长时间的一个例程是 get_cmos_time()。当内核引导时，此例程用于读取外部实时时钟（RTC）的值。目前，该例程延迟到下一次第二次翻转的边缘，以确保内核中的时间值相对于 RTC 是准确的。

但是，此操作可能需要一整秒的时间才能完成，从而在总启动时间内引入高达 1 秒的可变性。此例程中的同步很容易删除。可以通过目录 include / asm-i386 / mach-default / mach_time.h 中的函数 get_cmos_time()中的前两个循环来消除它，在其他架构的内核源代码树中也有类似的例程。

进行此修改的一个原因是，Linux 内核存储的时间不再与机器的实时时钟硬件中的时间完全同步。某些系统在系统关机时将系统时间保存回硬件时钟。经过大量启动和停机后，这种缺少同步的操作会导致实时时钟值与正确时钟时间有误差。由于每次引导周期的不同步数量高达一秒，所以这个差值可能很大。

使用这种方法的前提是在 RTC 驱动编译在内核中了，这个时候将下载的 B 补丁编译安装到已经编译完的新内核中，再修改内核启动的引导文件 init，启用“CONFIG_RTC_NO_SYNC_ON_READ”后重新启动，观测 RTC 是否被成功禁用。

4.6 本章小结

本章由测试数据结合 Docker 架构知识，确定虚拟机性能优化的方向，从优化虚拟机的启动速度来提升其部署速度，从而完成目标。最根本是进行内核的优化，从内核入手优化虚拟机操作系统的模块，针对 Docker 的使用依赖项，对内核各模块进行了裁剪和优化，同时对系统的 initrd 中脚本文件 linuxrc 进行优化。

5 测试数据分析

5.1 虚拟机模块及驱动优化分析

5.1.1 initrd 启动时间

优化之前：

```
('[2.038730 < 1.024451 >] Freeing initrd memory: 19784K\n',)
```

优化之后：

```
('[0.398107 < 0.188621 >] Freeing initrd memory: 19836k freed\n',)
```

优化结果：

initrd 完成释放内存操作用时 1.024451s，优化之后用时 0.188621s，使用了 busybox 的脚本 linuxrc 脚本代替原系统相关脚本，使 initrd 启动时间提升了将近 5.43 倍。

5.1.2 SELinux 模块

优化之前 bootlog 中的启动项 SELinux 相关部分如下代码：

```
('[0.069168 < 0.000001 >] SELinux:    Initializing.\n',)
('[0.069173 < 0.000005 >] SELinux:    Starting in permissive mode\n',)
('[3.925067 < 0.584710 >] SELinux: 32768 avtab hash slots, 104710 rules.\n',)
('[3.940810 < 0.015743 >] SELinux: 32768 avtab hash slots, 104710 rules.\n',)
('[3.966159 < 0.025349 >] SELinux:    8 users, 14 roles, 4975 types, 301 bools, 1 sens, 1024 cats\n',)
('[3.966162 < 0.000003 >] SELinux:    91 classes, 104710 rules\n',)
('[3.968291 < 0.002129 >] SELinux:    Permission validate_trans in class security not defined in policy.\n',)
('[3.968298 < 0.000007 >] SELinux:    Permission module_load in class system not defined in policy.\n',)
('[3.968361 < 0.000063 >] SELinux:    Class binder not defined in policy.\n',)
('[3.968362 < 0.000001 >] SELinux:    Class cap_userns not defined in policy.\n',)
```

('[3.968362 < 0.000000 >] SELinux: Class cap2_userns not defined in policy.\n',)
('[3.968362 < 0.000000 >] SELinux: the above unknown classes and permissions will be allowed\n',)
('[3.968368 < 0.000006 >] SELinux: Completing initialization.\n',)
('[3.968368 < 0.000000 >] SELinux: Setting up existing superblocks.\n',)
('[4.028735 < 0.060367 >] systemd[1]: Successfully loaded SELinux policy in 201.013ms.\n',)

优化后从内核模块中删除了 SELinux 模块，找不到与其相关的启动信息。实际生产环境中可以将 SELinux 模块编译，按需加载来提高安全系数。

5.1.3 kernel hacking 模块

kernel hacking 模块中包含了 printk 和 dmesg 两个重要的功能。在收集测试数据阶段，这两个功能是最重要的。

根据 3.4.1.2 节 bootchart 收集的数据，结合它的工作原理，可以看到启动项过程中的蓝色部分表示 printk 函数返回所耗费的时间，为了追求内核启动的极限速度，可以禁用 printk 函数来达到这个要求，相比直接在启动项源码中修改禁用 printk 函数，直接在内核配置过程中直接禁用更加快捷。

这样做带来的后果就是无法直观收集系统启动项时间，因此这一步优化是在最后所做，给内核打上 patch，禁用 kernel hacking 模块，使用 bootchart 和 grabserial 工具收集启动时间，基本符合预期结果。

5.1.4 RTC 驱动

优化之前 RTC 驱动挂载时间如下：

('[0.791417 < 0.000782 >] rtc_cmos 00:01: rtc core: registered rtc_cmos as rtc0\n',)
('[0.791590 < 0.000173 >] rtc_cmos 00:01: alarms up to one month, y3k, 114 bytes nvram, hpet irqs\n',)
('[0.817995 < 0.003939 >] rtc_cmos 00:01: setting system clock to 2017-05-11 08:57:29 UTC (1494493049)\n',)

RTC 驱动在内核启动时挂载，从图中可以看出 get_rtc_cmos()函数从执行到

返回正确时间需要经历多次调用，大大增加内核的启动速度，结合实际生产环境，承载容器的虚拟机在启动时并不需要这一步。

在实验阶段首先是采用 RTCNoSync 方案，直接禁用 get_rtc_cmos()函数，其次在内核配置阶段将 RTC 选项置否，结果都符合预期，在优化后的内核启动信息中没有 rtc 相关信息。

考虑到现实生产环境中执行一些操作时，主机的时间不同步会造成不可预知的错误，所以 RTC 驱动可以编译但在配置文件中设置不在内核启动时加载。

5.1.5 usb 驱动

优化之前 bootlog 中的启动项 usb 相关部分信息如下：

```
('[0.788529 < 0.000210 >] usb usb2: New USB device found, idVendor=1d6b, idProduct=0001\n',)
('[0.788531 < 0.000002 >] usb usb2: New USB device strings: Mfr=3, Product=2, SerialNumber=1\n',)
('[0.788532 < 0.000001 >] usb usb2: Product: UHCI Host Controller\n',)
('[0.788533 < 0.000001 >] usb usb2: Manufacturer: Linux 4.10.11 uhci_hcd\n',)
('[0.788534 < 0.000001 >] usb usb2: SerialNumber: 0000:02:00.0\n',)
('[0.788708 < 0.000174 >] hub 2-0:1.0: USB hub found\n',)
('[0.788726 < 0.000018 >] hub 2-0:1.0: 2 ports detected\n',)
('[0.789218 < 0.000492 >] usbcore: registered new interface driver usbserial\n',)
('[0.789233 < 0.000015 >] usbcore: registered new interface driver usbserial_generic\n',)
('[0.789240 < 0.000007 >] usbserial: USB Serial support registered for generic\n',)
```

log 中与 usb 相关相关信息达 71 条，图中截取了其中一部分，当在内核配置时去除了 usb support 之后，优化后 log 中找不到相关启动信息，符合预期结果。

5.1.6 parallel port 驱动

优化之前 bootlog 中的启动项 parallel port 相关信息如下：

```
('[9.328425 < 0.226803 >] ppdev: user-space parallel port driver\n',)
```

在内核配置阶段去除了 parallel port support 之后，优化的启动信息中找不到相关启动信息，符合预期目标。

5.1.7 其他模块

对照 4.3.2 章节做出的修改，使用内核模块指令 lsmod 查看相关模块是否存在，若不存在则表明在内核编译过程中就没有编译相关模块，结果符合预期。

5.2 虚拟机的启动时间数据分析

5.2.1 内核启动时间对比

优化之前：

```
('[0.818967 < 0.000972 >] Freeing unused kernel memory: 1608K\n',)
```

优化之后：

```
('[0.476708 < 0.000120 >] Freeing unused kernel memory: 1680k freed\n',)
```

优化程度：

优化之前的内核平均启动时间约为 0.818967s，优化后启动平均时间约为 0.476708s，启动速度提升 1.72 倍。

5.2.2 系统 boot 总时间

优化之前：

```
('[47.715360 < 1.090887 >] IPv6: ADDRCONF(NETDEV_UP): docker0: link is not ready\n',)
```

优化之后：

```
('[38.097239 < 0.493399 >] IPv6: ADDRCONF(NETDEV_UP): docker0: link is not ready\n',)
```

优化程度：

优化之前的总启动时间约为 47.715360s，优化之后总启动时间约为 38.097239s，启动速度提升 1.25 倍。

5.2.3 挂载模块时间

优化之前：

```
('[42.234142 < 19.178696 >] loop: module loaded\n',)
```

优化之后：

```
('[35.628861 < 13.851692 >] loop: module loaded\n',)
```

优化结果：

执行挂载模块的 loop 执行完成时间，优化之前为 19.178689s，优化之后为 13.861692s，优化了内核启动模块后，挂载模块速度提升了 1.38 倍。

5.3 本章小结

本章测试裁剪优化过后的系统在自身启动时间上是否得到优化，再在其上跑 Docker，运行之前的任务进行性能测试，收集数据与之前的数据进行比对。分析虚拟机的启动和部署速度得到提升，结果符合预期。

6 总结与展望

6.1 工作总结

云计算技术不断的发展和完善，推动了云服务的不断进步。本文就着重于云平台上为容器提供服务的虚拟机性能分析，得出其性能存在的瓶颈，针对其存在的问题提出解决方案，并提出优化虚拟机性能的可行方案。

本文在虚拟机性能分析及优化上所做主要工作如下：

1. 首先介绍了云计算背景及虚拟化技术，阐述了目前国内外的研究现状，具体分析了虚拟机性能优化的方案，明确了本课题研究的重点和目标。与此同时，也深入了解了 Docker 的架构组织、生态系统，更详细地了解了 Linux 内核的架构知识，理解了内核模块的作用。

2. 随后在明确课题目标的基础上，对比选取选取测试环境，包括物理机虚拟机配置、容器镜像选取等，之后对性能测试的工具和技术进行深入学习，最终选取内外相结合的方法，即从内部系统原有功能层面，添加代码来获得测试时局，外部借助相关性能监控工具，将测试数据具体化、可视化，便于性能的量化分析。

3. 接着收集优化前的各项测试数据，同时在物理机和虚拟机上运行 Docker 进行相同的任务，收集物理机上和虚拟机上 Docker 各项性能的差别，主要从 CPU 使用率、系统平均负载、磁盘 I/O 和网络传输方面来分析。

4. 由测试数据结合 Docker 知识，结合实际生产环境，确定虚拟机性能优化的方向，决定从优化虚拟机的启动速度来提升其部署速度，从而完成目标。最根本是进行内核的优化，从内核入手优化虚拟机操作系统的模块，针对 Docker 的使用依赖项，对内核各模块进行了裁剪和优化。这是非常庞大的一项工程，难点在于分析各模块作用以及容器对其依赖性，期间也经历过将文件系统编译成模块导致系统无法启动等等问题。

5. 最后测试裁剪优化过后的系统在自身启动时间上是否得到优化，再在其上跑 Docker，运行之前的任务进行性能测试，收集数据与之前的数据进行比对。虚拟机的启动和部署速度得到了大幅提升，同时也能很好地为 Docker 容器提供相关服务。

6.2 未来展望

目前的研究工作初步完成了一个面向容器的虚拟机设计，使其在启动速度大幅提升的基础上能为 Docker 提供稳定的服务。

本文知识针对 Docker 来优化了虚拟机的性能，并未对 Docker 本身做更多优化，未来可能有着针对某项特定任务或者特定平台的定制化容器化出现，当这样的容器运行在特定的虚拟机内时，其性能将会得到质的飞跃，当集群部署时，它能完成的任务不是一般云平台能望其项背的。

从单个虚拟机性能的运行结果来看，缩减虚拟机操作系统的体积可以使得物理机能部署更多虚拟机，提高物理机的资源利用率和性能。虚拟机针对容器进行了相关的优化，能够很好的提供 Docker 服务。当虚拟机在物理机上集群部署时，便可以将容器的优势发挥到最大，看似只有一台机器，其实其中跑着若干虚拟机，而虚拟机中又运行着更多容器，每个容器都相当于一个配置好的操作系统。可以进行的各种各样的操作，这就是云计算技术强大与魅力所在。

正如上述内容所展现的，当涉及到大规模集群部署时，资源的分配、容器的编排都会是很棘手的问题，当前社区中很火的 Swarm、Kubernetes 等技术都针对这些问题非常快的发展着。技术前沿的 Project Atomic、CoreOS 等都是面向容器设计的虚拟机 OS 的杰出产品，相信随着 Docker 日益壮大的生态圈，面向容器的虚拟机技术将会得到长足发展。

致谢

时间如白驹过隙，匆匆这四年，一晃又夏天。回想起四年前那个夏天，抱着重重行李来到华科的第一天，那些情景仍历历在目，在离别之际，我要深深感谢这个让我迷过无数次路的美丽校园。

非常感谢指导我毕设的吴松教授，想起大四上学期抱着对云计算浓厚的兴趣去听了吴教授的云计算基础导论，对教授渊博的学识，生动的教学方式印象深刻。非常感谢吴教授在毕设期间对我的关心，在百忙之中仍不忘抽出时间具体指导我课题的研究方向，当课题研究遇到瓶颈的时候，吴教授总能给我指明方向。

此外，我还要感谢课题研究期间悉心指导我的两位博士学长樊浩和黄卓，每周的例会上，他们总能及时地解答我的问题，对我的研究方向提出十分重要的指导意见，让我在不断地拨乱反正过程中得到了能力的提升。平时有问题向他们反映时，他们总能不厌其烦地向我详细解释，很幸运在毕设阶段能遇到如此认真负责的两位学长！

同时，我要感谢伴我走过四年时光的每一位老师和同学，尤其感谢李克松、李艺璇、付斌、汪雨昕、古文、韩云同学，一起走过的路，一起看过的景，我将一直珍藏在心，衷心祝愿各位前程似锦！

最后，感谢我的父母对我一直以来的养育与付出，感谢朋友们对我的支持与帮助，我将一直铭记在心。

参考文献

[1] 金海,廖小飞. 面向计算系统的虚拟化技术[J].中国基础科学.2008(06)

[2] 吴革,李健,赖英旭. 基于操作系统容器虚拟化技术的 JBS 模型的研究[J]. 网络安全技术与应用. 2010(04)

[3] 吴义鹏. 基于容器的虚拟机调度算法优化及实现[D]. 北京邮电大学 2011

[4] 李安伦. 基于 Xen 隔离的嵌入式 Linux 系统安全增强技术[D]. 南京理工大学 2013

[5] 薛海峰卿斯汉,张焕国. XEN 虚拟机分析[J]. 系统仿真学报. 2007(23)

[6] 浙江大学 SEL 实验室.Docker——容器与容器云[M].北京：人民邮电出版社，2016.10

[7] 鸟哥，王世江改编.鸟哥的 Linux 私房菜[M].（第三版）北京：人民邮电出版社，2010.7

[8] 博韦（美），陈莉君，张琼声译.升入理解 Linux 内核[M].（第三版）北京：中国电力出版社，2007

[9] Stephen J. Vaughan-Nichols.New Approach to Virtualization Is a Lightweight[J]. Computer . 2006

[10] Xavier M G,Neves M V,Rossi F D,et al.Performance evaluation of container-based virtualization for high performance computing environments[R]. Proc of 21st Euromicro international conference on parallel,distributed and network-based processing . 2013

[11] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi HArpaci-Dusseau. “Slacker: Fast distribution with lazy docker containers”[R]. In 14th USENIX Conference on File and Storage Technologies, 2016：181 ~ 195

[12] Biederman E,Networx L.Multiple instances of the global linux namespaces[J]. Proceedings of the Linux Symposium . 2006

[13] Anil Madhavapeddy,Richard Mortier,Charalampos Rotsos,David Scott,Balraj Singh,Thomas Gazagnaire,Steven Smith,Steven Hand,Jon Crowcroft. Unikernels[J]. ACM SIGPLAN Notices . 2013 (4)

[14] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM[R]. In Proceedings of the linux sym-posium, pages 19–28, 2009.

[15] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Oper-ating Systems:

Three Easy Pieces[M]. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[16] Jens Axboe, Alan D. Brunelle, and Nathan Scott. blktrace(8) - Linux man page[DB]. http://linux.die.net/man/8/blktrace, 2006.

[17] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multipro-cessors[R]. In Proceedings of the 16th ACM Symposium on Operat-ing Systems Principles (SOSP '97), pages 143–156, Saint-Malo, France, October 1997.

[18] Jeremy Elson and Jon Howell. Handling Flash Crowds from Your Garage[R]. In USENIX 2008 Annual Technical Conference, ATC'08, pages 171–184, Berkeley, CA, USA, 2008. USENIX Association.

[19] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing Performance Isolation Across Virtual Ma-chines in Xen[R]. In Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'2006), Mel-bourne, Australia, Nov 2006.

[20] Tyler Harter. HelloBench[A]. http://research.cs.wisc.edu/ adsl/Software/hello-bench/, 2015.

[21] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, Scalable Disk Imaging with Frisbee[J]. In USENIX An-nual Technical Conference, General Track, pages 283–296, 2003.

[22] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Gh-odsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center[M]. In NSDI, volume 11, pages 22–22, 2011.

[23] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg[J]. In Proceedings of the Eu-ropean Conference on Computer Systems (EuroSys), Bordeaux, France, 2015.

[24] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers[R]. In Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07), San Jose, Califor-nia, February 2007.