

分类号

密级

UDC ^{注 1}

学 位 论 文

网络功能虚拟化平台研究

郭帅

指导教师

章小宁

教授

电子科技大学

成 都

申请学位级别

硕士

学科专业

通信与信息系统

提交论文日期

2018.5.15

论文答辩日期

2018.5.23

学位授予单位和日期

电子科技大学

2018 年 6 月

答辩委员会主席

评阅人

注 1：注明《国际十进分类法 UDC》的类号。

**Research on Network Function Virtualization
Platform**

A Master Thesis Submitted to
University of Electronic Science and Technology of China

Discipline:	Communication and Information Systems
Author:	Guo Shuai
Supervisor:	Professor Zhang Xiaoning
School:	School of Communication & Information Engineering

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

作者签名： 郭帅 日期： 2018 年 6 月 20 日

论文使用授权

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

作者签名： 郭帅 导师签名： 章小宁

日期： 2018 年 6 月 20 日

摘要

最近几年来,个人用户网络流量需求越来越多,但是传统移动运营商的营收却不容乐观,究其原因是因为用户越来越高的网络要求使得运营商不断增加布置新设备的成本,控制底层设备成本的增加以降低成本成了运营商们普遍关心的话题,网络功能虚拟化(Network Function Virtualization, NFV)技术随之兴起。其作为一种可以广泛应用的技术手段,通过将传统运营商持有的网络功能硬件解耦和,将网络功能从专有的硬件上剥离开来,部署在 X86 通用服务器上,并依靠软件来实现相应的虚拟网络功能,以向外提供服务。这样可以根据实时的需求进行资源的有效调度,达到降低设备成本的需要。

而在实际应用部署中,具体的 NFV 使用还有很多的问题尚未解决,比如通用服务器一般用于处理传统的 IT 业务,其相比于专有硬件,在处理 NFV 中的高速率转发业务的时候,面临着严重的性能瓶颈,需要有针对性地分析在实际使用中如何提高 NFV 的使用效率。在单服务器中,数据包的整个转发流程会有多个过程会对网络功能的使用效率造成影响,而采用不同的虚拟化方式,也将对后续网络功能的部署、迁移产生不同的影响。

本文针对 NFV 在实际部署中的高性能需求,对实际的部署架构进行了层层分析,并对可能造成性能瓶颈的数据转发平面和虚拟化平面进行了分析,力图实现高性能地在通用服务器上进行 NFV 部署。在虚拟转发层,选取了业内应用广泛的三种软件交换技术 DPDK-OVS、NetMap 和 Click 来进行部署,针对 DPDK 和 OpenvSwitch 两种工程实现技术整合以及性能优化,针对 NetMap 在 NFV 场景下的使用改进了相关的内部数据结构进行优化,通过构建 Click 基础组件链构建 Click 软件路由器,实现了三种数据转发平面后,对三种数据平面的性能进行了对比测试和分析。在虚拟化方式的选择上,选用了 KVM 虚拟化技术和 Docker 容器技术进行分析,在实际的宿主机上部署了两种虚拟化方式,优化了相关了网络连接模式,并对两种虚拟化方式进行了性能测试,对结果进行了分析。最后,搭建大规模实验平台,针对不同的数据转发平面和虚拟化方式,部署了两种 NFV 实验平台,对虚拟网络功能进行了功能测试,对网络性能和容灾性能进行了测试对比分析,最后对三种数据转发平面和两种虚拟化方式在 NFV 技术落地中的选择提出了建议,以为后续 NFV 技术高效率地部署落地提供更好的支持。

关键词: 网络功能虚拟化, 数据转发平面, 虚拟化, 性能

ABSTRACT

In recent years, the demand for individual users' network traffic has been increasing, but the revenue of traditional mobile operators is not optimistic. The reason is that the higher and higher network requirements of users have caused the cost of deploying new equipment to be continuously increased in operations. Controlling the increase in the cost of the underlying equipment to reduce costs has become a topic of common concern to operators, and Network Function Virtualization (NFV) technology has emerged. As a widely used technical method, it decouples the network function hardware held by traditional operators and separates the network function from the proprietary hardware. It is deployed on X86 general-purpose servers and depends on the software. Implement corresponding virtual network functions to provide services outward. In this way, the resources can be effectively scheduled according to real-time requirements and the need for reducing equipment costs can be achieved.

However, in actual application deployment, there are still many problems in the use of specific NFVs. For example, general-purpose servers are generally used to process traditional IT services. Compared to proprietary hardware, they handle high-rate forwarding services in NFVs. Facing severe performance bottlenecks, it is necessary to analyze in a targeted manner how to increase the efficiency of NFV use in actual use. In a single server, there are multiple processes in the entire forwarding process of a data packet that can affect the use efficiency of network functions. Using different virtualization methods will also have different effects on the deployment and migration of subsequent network functions.

This thesis focuses on the performance requirements of NFV in actual deployment, analyzes the actual deployment architecture, and analyzes the data forwarding plane and virtualization plane that may cause performance bottlenecks. In the virtual forwarding layer, three software switching technologies, DPDK-OVS, NetMap, and Click, which are widely used in the industry, were selected for deployment. The technologies of DPDK and OpenvSwitch were integrated, and the use of NetMap in the NFV scenario was improved. The internal data structure, which builds the Click software router by constructing the Click infrastructure chain, implements three data forwarding planes, and compares the performance of the three data planes. In the selection of virtualization

methods, KVM virtualization technology and Docker container technology were selected for analysis. Two virtualized methods were deployed on the actual host machine, and related network connection modes were optimized, and two virtualization methods were implemented. A performance test was conducted and the results were analyzed. Finally, a large-scale experimental platform was set up. Two NFV experimental platforms were deployed for different data forwarding planes and virtualization methods. The virtual network functions were tested, and the network throughput performance and disaster recovery performance were compared and analyzed. Three data forwarding planes and two virtualization methods have been proposed in the selection of NFV technology.

Keywords: Network Function Virtualization, Data Forwarding Plane, Virtualization, Performance

目录

第一章 绪论	1
1.1 研究工作的背景与意义	1
1.2 国内外研究历史与现状	2
1.3 论文工作和内容组织	4
1.4 本章小结	4
第二章 网络功能虚拟化平台概要	5
2.1 网络功能虚拟化总体架构	5
2.1.1 NFV 系统架构模型	5
2.1.2 NFV 部署架构模型	7
2.2 数据转发平面	9
2.2.1 DPDK-OVS 的概述	9
2.2.2 Netmap 的概述	10
2.2.3 Click 的概述	11
2.3 虚拟化实现平面	12
2.3.1 KVM 虚拟化技术	12
2.3.2 Docker 容器技术	13
2.4 虚拟网络功能	15
2.5 本章小结	15
第三章 面向网络功能虚拟化的数据平面的对比	16
3.1 网络功能虚拟化部署框架	16
3.1.1 设备转发层	16
3.1.2 虚拟交换机层	17
3.1.3 虚拟网络功能层	18
3.2 实现 DPDK-OVS 数据面	18
3.2.1 部署 DPDK	18
3.2.2 整合 DPDK 与 OpenvSwitch	22
3.3 实现 NetMap 数据面	27
3.3.1 NetMap 架构	27
3.3.2 NetMap 初始化配置以及结构改进	29
3.4 实现 Click 数据面	31

3.4.1 Click 体系结构	31
3.4.2 Click 的部署以及调试	32
3.5 性能测试	34
3.5.1 测试环境介绍	34
3.5.2 测试结果分析	35
3.6 本章小结	41
第四章 面向网络功能虚拟化的虚拟方式对比	42
4.1 虚拟机技术与容器技术的对比	42
4.2 基于容器的网络功能虚拟化	43
4.2.1 Docker 容器架构	43
4.2.2 部署 Docker 容器及虚拟网络功能	44
4.2.3 优化容器网络连接	45
4.3 基于 KVM 的网络功能虚拟化	49
4.3.1 KVM 部署架构	49
4.3.2 创建 KVM 虚拟机	50
4.3.3 构建虚拟机网络	51
4.3 性能测试	52
4.3.1 测试环境	52
4.3.2 测试结果	52
4.4 本章小结	58
第五章 平台搭建与测试	59
5.1 实验平台	59
5.1.1 环境配置	59
5.1.2 环境搭建	59
5.2 实现虚拟网络功能	63
5.2.1 虚拟机署虚拟网络功能	63
5.2.2 容器部署虚拟网络功能	65
5.3 虚拟网络功能测试	65
5.4 性能测试	67
5.4.1 网络吞吐性能测试	67
5.4.2 容灾性能测试	70
5.5 本章小结	72
第六章 总结与展望	73

6.1 本论文工作总结	73
6.2 下一步工作展望	74
致谢	75
参考文献	76
攻硕期间取得的成果	79

图目录

图 1-1 NFV 的发展愿景	2
图 2-1 NFV 系统架构模型	5
图 2-2 NFV 部署模型	8
图 2-3 DPDK 软件组件	9
图 2-4 OVS 软件架构图	10
图 2-5 NetMap 架构图	11
图 2-6 Click 组件连接图	12
图 2-7 KVM 的虚拟化组件图	13
图 2-8 Docker 架构图	14
图 3-1 网络虚拟化部署架构图	16
图 3-2 SDN 交换机结构	17
图 3-3 DPDK 架构图	19
图 3-4 EAL 初始化流程	20
图 3-5 创建 Hugepage 文件系统	21
图 3-6 使用 MemSeg 结构	21
图 3-7 MemSeg 内存结构	22
图 3-8 传统 OpenvSwitch 部署与 DPDK-OVS 部署对比	23
图 3-9 对 OpenvSwitch 进行源码编译及配置	23
图 3-10 绑定网卡至 DPDK	24
图 3-11 创建网桥并加入 DPDK 端口	24
图 3-12 查看 CPU 资源	25
图 3-13 CPU 资源绑定	25
图 3-14 Filter 配置优化	26
图 3-15 OpenvSwitch 配置优化过程	26
图 3-16 NetMap 组件结构图	28
图 3-17 Netmap 全局配置	29
图 3-18 无锁队列伪代码操作。(a)添加数据操作；(b)删除数据操作	30
图 3-19 新 netmap_ring 结构图	31
图 3-20 NetMap 部署测试用例	31
图 3-21 Click 标准路由器配置	33

图 3-22 测试平台	34
图 3-23 测试带宽	36
图 3-24 OpenvSwitch 优化前后吞吐性能	36
图 3-25 NetMap 优化前后吞吐性能	37
图 3-26 纯数据平面转发性能	37
图 3-27 网络时延对比	38
图 3-28 OpenvSwitch 整合优化前后最大转发能力对比	39
图 3-29 NetMap 优化前后最大转发能力对比	39
图 3-30 纯数据平面最大处理能力	40
图 3-31 增加虚拟 I/O 后数据平面最大处理能力	40
图 4-1 KVM 架构与 Docker 架构	42
图 4-2 Docker 组件架构图	44
图 4-3 创建新的 Nginx Dockerfile	44
图 4-4 启动 Docker	45
图 4-5 查看 Docker 启动效果	45
图 4-6 使用一个 Linux bridge 来构建网络	46
图 4-7 改进后的容器网络	47
图 4-8 使用共享内存的容器网络	48
图 4-9 字符设备实现 mmap 函数主要过程	48
图 4-10 KVM 组件架构图	49
图 4-11 virt-manager 管理 KVM 虚拟机	51
图 4-12 KVM 虚拟机网络	52
图 4-13 三个 VNF 构成功能链时性能	53
图 4-14 六个 VNF 构成功能链时性能	53
图 4-15 CPU 性能对比	54
图 4-16 多个客户机时 CPU 性能对比	55
图 4-17 内存性能对比	55
图 4-18 网络吞吐性能对比	56
图 4-19 Docker 容器与 KVM 虚拟机重启时间对比	57
图 5-1 系统连接图	60
图 5-2 OpenStack 节点网络连接图	61
图 5-3 节点服务配置	62
图 5-4 OpenStack 服务	62

图 5-5 安装 Docker 实现过程	63
图 5-6 查看 Docker 版本	63
图 5-7 配置镜像制作文件	64
图 5-8 制作的镜像文件	64
图 5-9 OpenStack 镜像库	65
图 5-10 创建配置 Snort 软件的 Dockerfile	65
图 5-11 发送端服务器 ping 接收端服务器	66
图 5-12 VNF 中 Snort 告警规则	66
图 5-13 VNF 中 Snort 告警信息	67
图 5-14 包转发能力测试结果	68
图 5-15 转发性能测试结果	69
图 5-16 网络时延测试结果	69
图 5-17 创建 VNF	70
图 5-18 VNF 信息	71
图 5-19 迁移控制指令	71
图 5-20 迁移后的 VNF	71

表目录

表 2-1 软件交换机技术特性对比.....	8
表 5-1 物理网卡配置表.....	61
表 5-2 集群配置信息.....	62

第一章 绪论

1.1 研究工作的背景与意义

近些年来,各种各样的网络服务功能不断出现,个人用户的网络流量也不断激增,但是传统移动运营商却陷入了营收不断减小的困境。这是因为随着大规模新网络功能的部署,需要运营商不断加大底层专有设备的投入,这使得运营商的设备成本不断上升。与此同时,如果不能对这些设备采用有效的规划,这些设备产生的电力支出、运维成本也是困扰传统运营商的问题^[1]。网络功能虚拟化(NFV, Network Function Virtualization)技术被认为是解决上述问题的一个行之有效的方法。

传统的网络运营商提供的网络功能通常部署在专有的硬件设备上,当客户提出的需求增加,运营商就必须对现有的网络设备进行升级,这使得设备集群的集成度不断上升,并且大大加大了集群控制运维的复杂程度。专有设备一方面对于功能的部署存在严格的周期限制,整个流程需要不断地对规划、设计、开发、维护进行迭代。与此同时,专有硬件内部的芯片集成度越来越高,这使得芯片开发成本也越来越高^[2]。故而针对专有网络硬件设备的改进势在必行。

随着通用芯片的计算能力在近些年来的巨大提升,这使得通用芯片的发展和使用愈发地得到重视。IT行业所使用的服务器由标准IT组件(如X86架构的通用芯片)构成,目前全球标准服务器的销售量已突破百万台。这些具有极强兼容性的基础硬件平台的繁荣发展为NFV的落地打好了坚实的基础。

NFV通过将虚拟网络功能部署在X86等通用服务器上,以及充分利用虚拟化技术,将传统专有网络硬件设备解耦和,实现软件和硬件的分离,可以充分实现物理资源的灵活共享,实现面向新网络功能需求的功能快速开发和自动化部署,并基于实际业务需求提供弹性扩容缩容、错误清除功能以及自动化升级维护等功能。并且NFV可以通过在底层物理设备的远程进行批量式的对现有设备做出更新和维护,更加灵活地控制和调度了现有的资源,这也使得运营商的运营以及维护成本大大降低。

如图1-1所示,NFV改变了传统电信设备的形态和实现方式,基于通用标准X86服务器^[3]、存储设备和网络设备以软件的方式实现了现有传统电信设备的功能。这些软件功能上与专有硬件设备所能提供的功能是具有一致性的^[4]。

由于NFV具有以上这些相对于传统电信网络专有设备的优势,使得NFV的发展备受关注。欧洲电信标准协会(European Telecommunications Standards Institute, ETSI)发起成立了一个新的NFVISG(Network Function Virtualization Industry

Specification Group), 目前已有超过 200 家传统网络运营商、网络专用设备供应商、新兴 IT 设备供应商等相关机构参加该工作小组^[1]。ETSI 机构成立 NFVISG 最初的主要目的是在 NFV 的发展过程中提供早期的架构设计经验分享, 并输出相关的标准化文档。在这个基础上, ETSI 虽然已经将 NFV 基本架构确定下来, 但整个 NFV 的发展仍然面临多种考验。

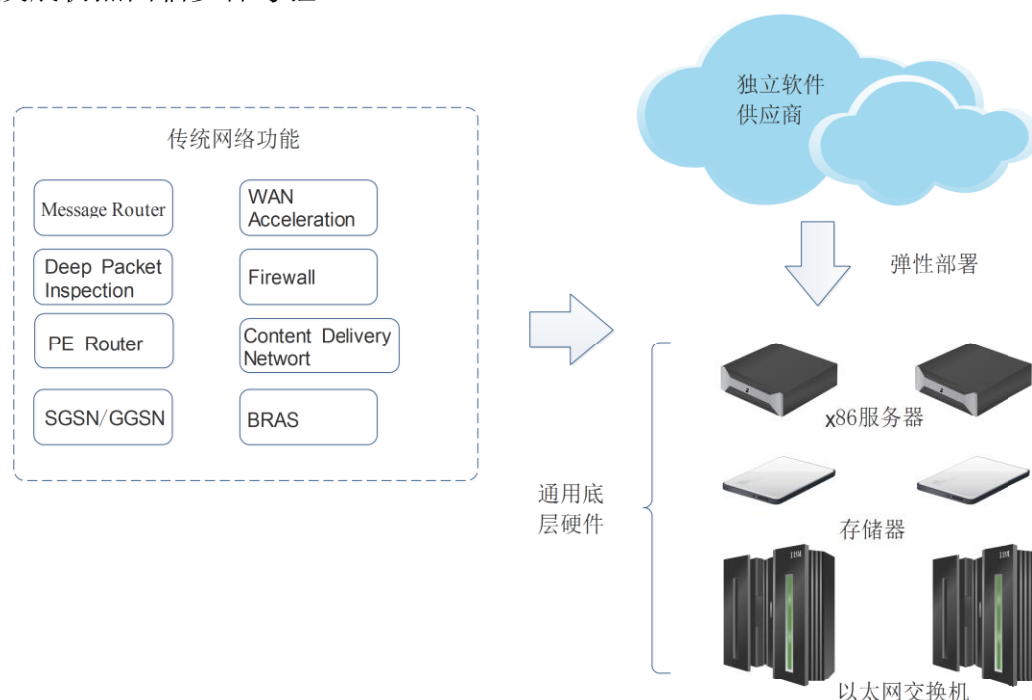


图 1-1 NFV 的发展愿景

1.2 国内外研究历史与现状

2010 年 10 月, 世界上几个巨头运营商共同合作推出了第一份 NFV 白皮书, 使 NFV 正式走到了世人面前^[6]。2012 年 11 月, 在 ETSI 的支持下, BT、Deutsche Telekom、AT&T 等 7 家运行商牵头成立了 NFVISG 组织, 为 NFV 的进一步规范化做出了巨大贡献。2013 年 AT&T 公司率先推出了 Domain2.0 计划, 以针对用户应网络云项目开展进一步研究。2014 年华为公司在世界移动通信大会上正式推出基于开源技术的 FusionSphere 云平台, 通过之前在软硬件方面研究的积累, 利用不同软件功能和虚拟化方案来实现网络功能的虚拟化^[3]。

当前国内外对 NFV 的研究多集中于将 NFV 在云环境中落地这一方面, 厂商通过对现有技术的升级, 对云平台的功能进行扩展以适应 NFV 发展的需要。ETSI 定义下的 NFV 标准架构主要由 NFV 基础设施 (NFV Infrastructure, NFVI)、虚拟网络功能 (VNFs) 和 NFV 管理与编排 (NFV Management and Orchestration, NFV MANO) 三个部分组成^[7]。其中, NFVI 组件负责提供上层应用所必须的各种物理

资源和虚拟资源，VNFs 包含了各种虚拟网络功能组成的电信业务，NFVI 和 VNFs 这两部分对应于传统的电信业务网络。在这个基础上，NFV 新提出了 MANO 模块来对业务网络域中所有的设备、资源以及业务进行统一化的部署与管理^[8]。

在具体的 NFV 使用中，需要结合底层通用服务器的技术特点来进行 NFV 技术的部署与实现。在部署过程中，业内根据通用服务器的特点针对其中的数据转发平面和虚拟化方式展开研究。其中数据转发平面决定了网络数据包在单服务器内各个 VNF 之间转发的效率，虚拟化方式决定了 VNF 计算能力、IO 性能、隔离安全性等关键指标。

在传统 Linux 服务器自带的 Linux Bridge 以外，国内外公司研究了多种数据转发平面以供开发者选择，比如 OpenvSwitch 可以提供灵活的转发规则控制但由于自身设计造成了糟糕的性能^[9]，CuckooSwitch 数据平面可以提供高效率的吞吐性能但是管理十分复杂^[10]，NetMap 依靠零拷贝技术可以实现网络包从物理网卡到用户空间的快速转发^[7]，但仍具有提升性能的空间。Netgraph 和 Netfilter 等软件通过数据包过滤钩子的技术，将特定的勾取数据包的代码插在数据处理连里，实现从网卡驱动程序截获数据包，从而实现不需要拷贝数据包来进行包的转发。

虚拟化方式根据不同的目标可以分为不同的虚拟化级别，KVM 所代表的全虚拟化方式和 Docker 容器技术代表的操作系统级虚拟化方式也被广泛用于软件功能的部署和应用，从而提高单服务器的利用率。其中 KVM 虚拟化技术由 Qumranet 公司开发，2007 年被集成进入到 Linux 内核代码树中，它继承了 Linux 大部分优势，可以充分利用 Linux 内核不断优化优势。而 Docker 容器技术诞生于 2013 年，是一种基于 Linux Container 的系统级虚拟化技术，可以更加方便快捷地完成对应用的隔离部署。此外还有 Xen、Z/VM 等虚拟化技术也在飞速发展。

在实际应用中，具体的 NFV 使用还有很多的问题尚未解决，比如通用服务器在处理 NFV 中的高速率转发业务的时候，面临着严重的性能瓶颈，需要有针对性地分析在实际使用中如何提高 NFV 的使用效率。在单服务器中，数据包的整个转发流程会有多个过程会对网络功能的使用效率造成影响，而采用不同的虚拟化方式，其网络连接方式也会产生一定的性能瓶颈，不同的虚拟化技术特点也将对后续网络功能的部署、迁移产生不同的影响。

由此可见，当前业界对于 NFV 中的 VNF 编排领域研究较多^[11]，而对于将具体的虚拟网络功能如果落地，如何提高虚拟环境与物理环境的资源利用率，如何正确选择数据面和虚拟化方式使网络功能高效率、高可靠性地运行起来研究内容较少。

因此，本文从数据面入手，自底向上对比分析选择不同数据面和不同虚拟化方

式对 NFV 使用时所产生的差异，在研究过程中对数据面的瓶颈进行优化处理，并优化容器网络部署方案，从而对后续 NFV 技术落地过程中的高性能部署方案提供更多的帮助与支持。

1.3 论文工作和内容组织

本文的各章节主要内容安排如下：

第一章为绪论。首先介绍了本文的研究背景，阐述了网络功能虚拟化技术的出现原因以及发展历程，然后介绍了当前技术条件下国内外对于 NFV 的研究现状，以及本文将会进行的研究内容。

第二章是本文的相关研究工作。首先介绍了 NFV 的相关基础知识，对 NFV 系统中涉及到了三个重要模块 NFVI、VNF 以及 VNF MANO 进行了详细阐述。接着阐明了 NFV 技术落地中影响 NFV 运行效率中的几个关键部分，包括数据面，虚拟化方式等相关知识，介绍了三种常用的数据面和两种常用的虚拟化方式。最后对常用的两种虚拟网络功能软件进行了简单的阐述。

第三章是对数据面的研究。详细阐述了 DPDK-OVS、NetMap 以及 Click 三种数据面的数据加速原理以及部署方式，并对 DPDK-OVS 数据面以及 NetMap 数据面进行优化，接着搭建了实验测试平台，对三种数据面的数据转发性能进行了测试，并对测试结果进行分析。

第四章是对虚拟化的研究。详细阐述了当前两种主流虚拟化方式的虚拟原理，优化了 Docker 容器网络连接方案，搭建实验测试平台，对两种虚拟化方式进行了性能测试，详细分析了产生差异的原因以及后续使用的适用场景。

第五章在前两章的基础上，搭建了大规模测试平台，通过将不同数据平面和虚拟化方式结合，部署虚拟网络功能，对 NFV 平台进行功能测试和性能测试。

第六章对所做工作进行了总结，并对 NFV 的发展进行了展望。

1.4 本章小结

本章节首先介绍了网络功能虚拟化的发展历史以及研究现状，接着介绍了本文的主要研究内容，最后列出全文的整体结构。

第二章 网络功能虚拟化平台概要

2.1 网络功能虚拟化总体架构

ETSI 在 2012 年的 NFV 标准化大会上推出了 NFV 的具体架构，本节以 ETSI 提出的 NFV 架构为基础模型，对 NFV 的各个功能模块进行详细阐述，并对 NFV 具体在物理环境下的部署模型进行分析。

2.1.1 NFV 系统架构模型

NFV 系统架构主要由三个重要模块组成，分别为 NFV 基础设施模块（Network Function Infrastructure, NFVI）、虚拟网络功能模块（Virtualized Network Functions, VNFs）和管理与编排模块（NFV Management and Orchestration, MANO）三个组成部分^[10]。

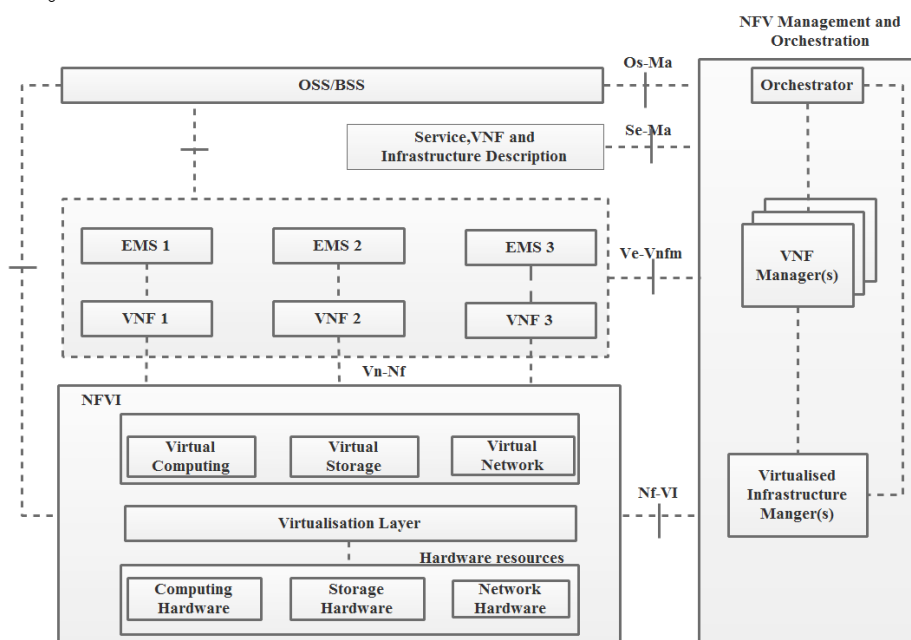


图 2-1 NFV 系统架构模型

其中 NFVI 作为整个网络功能虚拟化架构的基础设施层，可以对外提供虚拟计算资源、虚拟存储资源和虚拟网络资源三种资源，这些功能是由其中的 X86 通用服务器中的硬件资源（Hardware resources）、虚拟化抽象层（Virtualisation Layer）和虚拟机资源（Virtual Machine）三个部分共同对外提供的。

VNFs 作为一整套虚拟网络功能的集合，是由虚拟网络功能组件（VNF）、网元管控系统组件（Element Management System）和维护与商业支撑系统组件

(Operation and Business Support System) 三个子系统组合而成的。其中 VNF 根据最新的网络架构还可以细分为 IP 多媒体子系统和演进分组核心等网络结构。

而 MANO 作为管理编排域, 由资源编排组件 (Orchestration)、虚拟网络功能管控组件 (VNF Management) 和虚拟基础设施管理组件 (Virtualised Infrastructure Management) 三个子系统构成^[10], 根据上层的指令, 负责合理调配底层资源、资源额度管理和故障处理等需求。

(1) NFVI

NFVI 模块作为 NFV 整体架构的基础设施, 是网络功能虚拟化对外提供服务的基础执行者, 向外提供的基础资源有通用硬件、虚拟层和虚拟资源三部分。

通用硬件诸如 X86 服务器等标准化服务器, 包括了计算硬件, 存储硬件和网络硬件。X86 等标准化服务器除了近些年来计算性能得到了飞速的提高, 还具有便宜实惠的特点, 可以使得运营商摆脱传统专有硬件的束缚, 运营商可以根据现有的资源合理的实行资源的有效调度。虚拟化层是负责将底层的物理硬件实体资源提供抽象服务, 将其抽象为对应的虚拟资源。其实现的主体是在标准化服务器上的虚拟机监视器, 运行在操作系统之上, 可以将底层的实体计算、存储、网络等资源抽象化, 对上层模块提供相关接口以供调用。虚拟层可以使用技术手段使得硬件软件解耦和, 使得资源利用率得到大幅提高。虚拟资源抽象于底层物理资源, 对别对应底层的计算、网络以及存储资源, 将其以虚拟化的形式对外提供。虚拟资源根据虚拟化方式的不同对外提供服务, 可以使用虚拟机, 也可以使用容器来将物理资源虚拟化。

(2) VNFs

VNF 作为具体的虚拟网络功能, 其用软件来实现, 部署和运行在底层的 NFVI 之上, 且不应依赖于硬件资源。每一个物理网元都可以看做一个具体的 VNF, 每一个 VNF 又可以由一个或者多个网络功能模块组成, 该 VNF 实例可以被一一映射到单位个虚拟化环境比如虚拟机或者容器之中。VNF 的管理执行者是网元管理系统 (Element Management System, EMS), 其负责对 VNF 的安装、监控、配置等多个运维步骤进行有效的监督和控制。在运营商的部署中, EMS 涵盖了业务支撑系统和操作支撑系统所需要的基本信息。

(3) MANO

NFV 的 MANO 模块主要由三大组件构成, 分别为虚拟化的基础设施管理器组件 (Virtualized Infrastructure Manager), 虚拟化的网络功能模块管理器组件 (Virtualized Network Function Manager) 和网络功能虚拟化编排组件 (Network Function Virtualization Orchestration) ^[11]。

VIM 负责管理 VNFI 中底层的物理和虚拟的计算资源、存储资源以及网络资源，同时可以实现一些 NFV 的服务功能链。

NFVO 可以针对整个 MANO 平台根据上层需要进行有效的调度，负责在操作环境下针对虚拟网络功能服务的生命周期和维护方法做出相应的策略。。它对 OSS/BSS 开放接口来匹配二者的需求，调配合适的 VNF、物理网络功能和安排虚拟链路，从而提供符合需求的网络功能服务。

VNFM 主要负责 VNF 进行管理，尤其是对 VNF 生命周期进行管理，如相应功能的部署和进一步升级等，以及负责对 VNF 运营期中出现的问题如监控、维护等指标进行管理。

2.1.2 NFV 部署架构模型

ETSI 自上而下设计了 NFV 的系统架构模型，而针对整个 NFV 系统的性能瓶颈分析，需要结合通用服务器的具体架构自下而上地逐层分析，以解决整个 NFV 系统在面对高速网络数据包转发业务时可能会遇到的性能瓶颈。

在 NFV 的使用场景中，虚拟网络功能一般被部署于 X86 通用服务器的虚拟化环境之中，网络包的流程与传统 IT 环境下一致。网络数据包从一个环境到达另一个环境需要经过多个处理流程，比如虚拟化实现层、宿主机以及虚拟环境的 I/O 接口、操作系统内核协议栈等等。而在多个网络功能之间，可以根据需要搭建合适的虚拟网络构建连接，这被称之为服务功能链技术。因此，要针对 NFV 部署环境下的性能分析，需要针对这两个方面来进行。

单服务器的转发性能瓶颈是整个业务瓶颈的主要问题，其原因主要是因为通用服务器系统中采用了软件实现的转发和交换技术。如图 2-2 所示，如果使用虚拟机的方式实现虚拟化，业务流量在 X86 服务器之间从虚拟化环境到另一个虚拟化环境通过时，需要依次通过虚拟机中的虚拟网卡、宿主机中的数据转发平面和宿主机中的物理网卡三个部分。而在软件结构上，报文的收发需要经过虚拟机用户态应用、虚拟机内核态网络协议栈、虚拟机虚拟网卡驱动程序、内核态虚拟交换层、宿主机内核网络协议栈和物理网卡驱动程序等多个中间过程，这一过程存在着大量的虚拟化指令封装、内存内容复制、内核上下文切换、系统中断等大量 CPU 动作 [12-14]。

针对上述数据平面转发数据包性能影响的因素，业界针对部署在宿主机中的数据转发平面提出了多种改进方法，如针对数据包在通过操作系统内核协议栈时的多次复制过程和内核上下文频繁切换问题，提出了零复制技术^[15]。这种零复制技术的出现可以使得位于用户空间的进程可以直接读取物理网卡中的设备缓冲区，

并且可以通过设置内核空间中的 IO 驱动, 使用内核的虚拟内存映射技术, 在内核态和用户态之间构建映射关系, 这样一来, 运行在用户态的进程可以通过使用标准的读写技术实现对物理网卡的缓冲区中的内容进行零复制收取和发送^[16-18]。

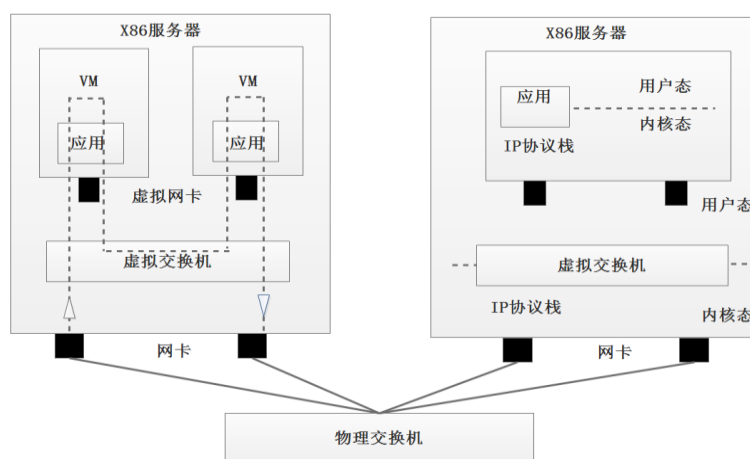


图 2-2 NFV 部署模型

此外还提出了 PF 环技术。PF 环技术在宿主机内核空间中划定一块固定的区域, 创建成一个环形的缓冲区域, 接着在原有的套接字之外新定义一种 PF 套接字, 使得运行在用户态的进程可以直接利用该技术手段规避开宿主机操作系统内核协议栈, 直接对内核中的环形缓冲区域进行数据访问^[19]。除了这些技术, 针对转发过程中影响性能的因素的具体原因, 各大开源社区推出了针对不同瓶颈的软件交换机, 分别采取不同的方式以降低软件系统层面带来的性能消耗。它们之间的对比如表 2.1 所示。本文选取了其中三种目前最常用的三种开源软件 DPDK-OVS, NetMap 和 Click 来进行对比分析。

表 2-1 软件交换机技术特性对比

	Batching	Multi-queue	Zero-copy	Portability	Domain
Raw Socket	支持	不支持	不支持	支持	User
NetMap	支持	支持	支持	不支持	User
Click	不支持	不支持	不支持	支持	Both
PacketShader	支持	支持	不支持	不支持	User
Kernel-bypass	不支持	支持	支持	支持	User
DPDK	支持	支持	不支持	不支持	User
PF_RING	不支持	支持	不支持	支持	User
RouteBricks	不支持	支持	不支持	不支持	Kernel

2.2 数据转发平面

如表 2.1 所示，当前业界对于软件交换机提出了多种的技术方案，DPDK-OVS^[20]、NetMap^[21]以及 Click 作为当前应用最广的三种软件交换机有着各自的技术特点，具有极佳的研究对比价值。

2.2.1 DPDK-OVS 的概述

DPDK (Dataplane Packet Development Kit)，是 intel 公司对外开放的数据平面开发工具套装，使用 IA (Intel architecture) 处理器架构的用户可以在宿主机中的用户空间中对数据包进行高效的处理，库函数和驱动由 DPDK 负责维护。它的设计初衷与通用化服务器的目的不同，它的提出主要是针对于网络应用中数据包的高性能处理^[22]。这个目的的实现具体体现在 DPDK 应用程序运行在用户空间，使用了自己自带的数据平面库来对网络数据包进行收发，从而绕过 Linux 内核协议栈中网络数据包繁琐的处理过程。

DPDK 提供了对数据包进行高效率处理的库文件和 PMD (Physical Media Dependent) 驱动的集合。其架构组成主要由两部分构成：环境抽象层 ELA (Environment Abstraction Layer) 和核心组件部分，在这之中核心组件部分主要包括包括内存管理、无锁环形队列、PMD 驱动、Hash 算法等，ELA 则是为核心组件的开发提供了更加友好的接口，以屏蔽底层物理硬件系统带来的差异^[23]。DPDK 软件组成如图 2-3 所示^[23]。

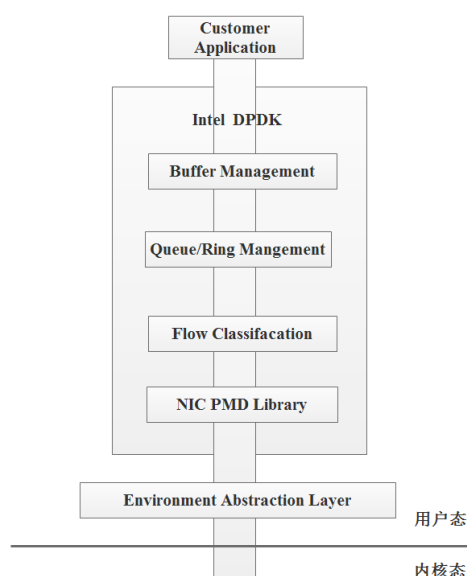


图 2-3 DPDK 软件组件

OVS (OpenvSwitch) 是一个高质量的，多网络分层的虚拟交换机，其目的是

实现控制平面和转发平面的解耦和，使得大规模网络自动化可以通过控制器之上的编程来进行扩展。同时 OpenvSwitch 还支持标准的接口管理和协议。OpenvSwitch 提供的转发服务具有两大优点：一是可以进行相对灵活的配置，因为交换机中的转发功能是通过相关的软件技术来完成的，可以在一个宿主物理服务器中部署数十个虚拟交换机，端口可以多种搭配，可以添加虚拟网卡和物理网卡，也支持远程的端口流表控制；另一大优点是其成本低廉，仅仅依靠软件的方式就可以轻易达到很快的交换速度。

OpenvSwitch 主要由三大模块构成：内核空间中的核心模块 `openvswitch.ko`、部署于用户空间中的守护进程 `ovs-vswitchd` 以及交换机数据库服务器 `ovsdb-server`^[24]，OVS 的软件架构如图 2-4 所示。

DPDK 和 OpenvSwitch 由设计目的不同，可以针对不同的实际性能瓶颈进行优化。基于 DPDK 的优化技术方案，对传统的 OpenvSwitch 中的数据路径进行了改写，使用了轮询模式驱动程序，即 PMD 来对报文进行接受和发送，使用大页内存对数据包进行缓存，使用无锁队列对数据包有效的规划排列等，从而大大优化了 OpenvSwitch 数据平面的包处理效率，极大地提高了 OpenvSwitch 的转发性能和吞吐量。

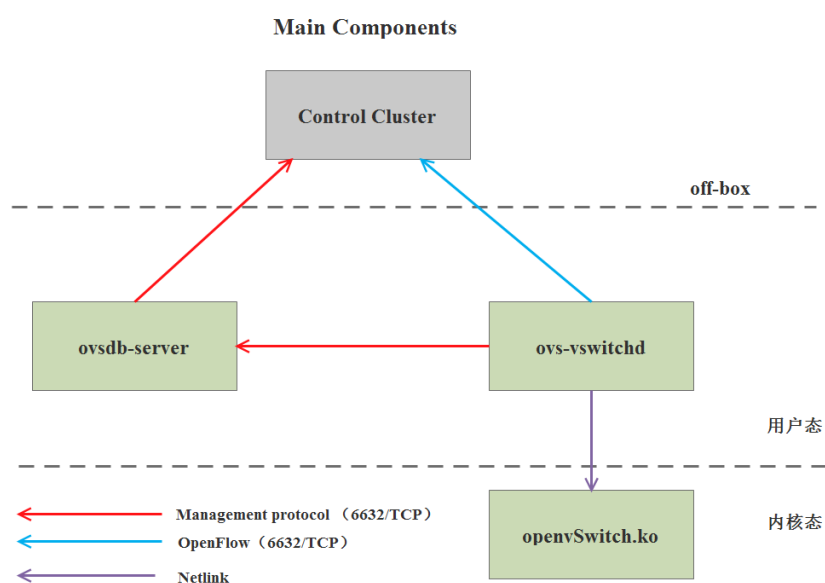


图 2-4 OVS 软件架构图

2.2.2 Netmap 的概述

Netmap 技术是一个针对原始网络流量包进行收发的高性能架构，由相关内核模块和用户态函数库组成^[25]。其设计初衷是在不对现有操作系统软件和硬件进行修改的基础上，使到达物理网卡的网络流量包可以规避开操作系统内核协议栈，由

运行在用户空间的进程同物理网卡之间直接进行相关交互动作，进而实现用户空间中的进程和网卡之间数据包的高性能传递。

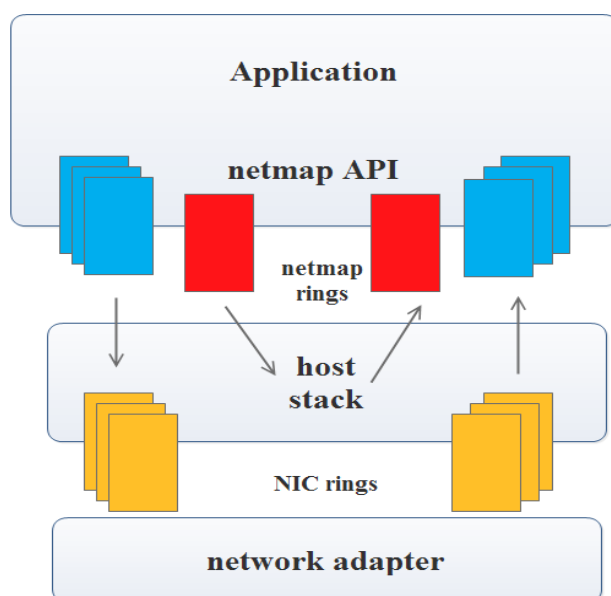


图 2-5 NetMap 架构图

Netmap 实现的方式是基于零拷贝思想的高速网络 I/O 架构，其架构如图 2-5 所示。

当宿主机部署了 NetMap 框架后，可以使物理网卡进入 NetMap 模式。这时传统宿主机中 NIC ring 会与操作系统中的网络协议栈解耦和，并拷贝一份 NIC ring 交给 NetMap 虚拟交换机进程使用。NetMap 在拥有 NetMap 环后，还会新建一个内存结构，即一对环并负责对其维护，这对环将负责与宿主机网络协议栈进行网络通信。NetMap 环和新建的一对通信环均在内存的共享空间实现，可以用来对网络数据包进行缓存。用户空间中的进程可以通过调用 NetMap 向外提供的 API 来获取 NetMap 环中的数据包内容。

2.2.3 Click 的概述

Click 是一个模块化的可定制路由器，它是由一些组件按照特定的方式组合而成的^[26]。组件是 Click 路由器中最基本的功能单元，其本身只能执行一些最基本的功能，如复制、分类、排队和调度等功能，复杂的功能需要用若干个组件共同组合来完成^[27]。基础行为组件可以由 element 类对象、输入输出接口以及配置连接串三个部分组成。如图 2-6 所示，FromDevice、counter 和 discard 模块都作为一个 element 类对象，其中的需要通过在输入输出接口之间完成，配置串即用户下达给 Click 的参数配置。

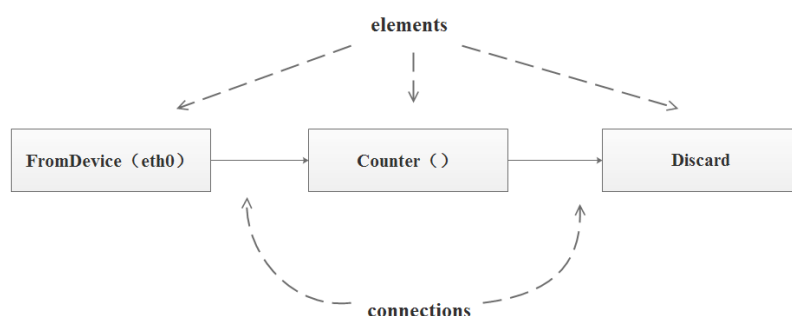


图 2-6 Click 组件连接图

在整个 Click 功能框架中，开发者已经通过 C 语言实现好很多的基础组件，对其进行有效的规划和连接则可以通过基础组件构建高级的网络功能组件。比如如果将特定多个简单的组件相连接并加以声明，就可以构建成用户所需的特定的路由功能，在图 2-6 所示的 Click 连接中，这个路由配置可以写为如下形式：FromDevice(eth0)->Counter()->Discard()，建立一个文本将这个配置保存在 Click 配置目录下即可将这个网络功能运行起来^[26]。

2.3 虚拟化实现平面

虚拟化作为这个时代最热门的 IT 技术之一，近些年来得到了飞速的发展。虚拟化将物理服务器所能提供的资源抽象成独立的虚拟资源，从而为不同需求的用户提供服务，同时保证了服务的隔离性和安全性。KVM (Kernel-based Virtual Machine) 虚拟化技术和 Docker 容器技术作为业界实现虚拟化的主流方式，从不同层面实现了虚拟化。

2.3.1 KVM 虚拟化技术

KVM 是由相关虚拟化扩展技术发展而来的，其通常部署于 X86 标准化服务器上，运行在 Linux 系统环境中，是该系统的原生全虚拟化解决方案^[28]。在 KVM 技术中，运行在 Hypervisor 之上的客户虚拟机通常表现为一个的 Linux 进程，由 Linux 操作系统中的调度程序进行统一管理。虚拟机出的每个虚拟 CPU 从系统用户的角度来看是一个常规的 Linux 进程，宿主服务器中的 Linux 操作系统内核中的相关功能也可以被 KVM 虚拟机进行直接调用。

KVM 在当前 Linux 操作系统中被实现为一个内核中的基本模块，现在市面上很多的操作系统都可以在 KVM 的支持下完成部署；通过将 KVM 实现为底层操作系统中的一个模块，其运行时表现为一个基本的进程，这使该虚拟化技术能够更加贴近操作系统，可以直接调用 Linux 内核的系统函数。KVM 模块是 KVM 虚拟机

架构中的核心模块，它负责整个 KVM 虚拟机从启动到运行的全部管理，首先其需要对底层物理硬件进行初始化操作，使宿主机进入支持虚拟化的模式，然后创建虚拟机并完成部署，之后为客户虚拟机在运行过程中所需要的资源进行支持调度。然而，KVM 本身不执行任何操作指令集模拟，用户空间的程序需要通过相关的接口映射到一个内核空间，向它提供模拟的 I/O，并将它的内容映射回宿主机的内容，实现这个功能的应用程序就是 QEMU。

如图 2-7 所示，在实际的宿主机中，底层硬件作为基础资源的支撑者，向上层提供当前条件下所拥有的物理资源；Linux 系统内核加载 KVM 模块，加载了 KVM 模块后即可创建/dev/kvm 设备来提供内存虚拟化。由 Hypervisor 虚拟机监控层创建虚拟机，并在虚拟机中加载用户所需的特定虚拟机操作系统，Hypervisor 虚拟机监控层会将客户机作为一个进程运行并为它们映射相应的地址空间^[29]。QEMU 将会被部署并运行在虚拟机操作系统中，其中虚拟机的所有 IO 请求都会被 Hypervisor 截获并转发给用户空间的 QEMU 进程^[30]。GuestOS 可以在两种模式下运行，一种是在 Guest Mode，这种模式下运行的 GuestOS 可以支持标准的内核；另一种是在 User Mode，这种模式下运行的 GuestOS 则支持自己的内核和 User-space(Applications)。



图 2-7 KVM 的虚拟化组件图

2.3.2 Docker 容器技术

传统的虚拟化方式如 KVM 虚拟化技术都是在一台物理服务器添加一层虚拟机监控器（Hypervisor），通过 Hypervisor 虚拟机监控器和 QEMU 进程的指令集模拟，虚拟化出多台虚拟机以供上层用户来使用，在虚拟机中的客户端操作系统中运行的 GuestOS 执行环境完全独立。但是随着当前网络功能虚拟化的发展要求，这

种虚拟化方式所带来的性能瓶颈也越来越大，而容器技术的出现则为提高网络功能虚拟化的运营效率提供了另一种解决思路。

容器技术是 linux 平台下一种轻量级的虚拟化技术，相比于传统的 KVM 虚拟化方式不同，容器技术不需要添加一层 Hypervisor 虚拟机监控器的辅助，不需要全虚拟化的指令集模拟，一个宿主机内核可以供容器和宿主机共同使用，因此所带来的性能损耗和运营效率在某些方面上要优于 KVM 虚拟化方式^[31]。

作为容器技术中的领头羊，Docker 与 2013 年 3 月正式发布开源版本。Docker 的思路与 KVM 不同，它从 Linux 内核入手，利用 Linux 中的 Namespace^[32]和 Cgroups^[33]等技术，为用户提供了一种轻量级的虚拟化技术^[34]。Docker 由于不需要 Hypervisor 的辅助，大大减小了部署时的额外开销，可以直接和宿主机的底层操作系统共享内核，在性能上几乎没有任何损耗，这样以来即可以更加充分地调用系统资源^[35]。同时，Docker 也提供了一定程度上的隔离化环境，保证部署在之上的业务相互之间不受影响。

如图 2-8 所示，相对于 KVM 虚拟化方式而言，Docker 容器技术用 Docker Engine 层取代了 Guest OS 层和 Hypervisor 层。虚拟机实现资源隔离的方式是使用各自独立的 OS，并利用 Hypervisor 和其他技术虚拟化 CPU、内存、IO 设备底层物理设备。而对比 KVM 虚拟化方式，Docker 容器技术从原理上来看则显得简练很多。Docker Engine 可以简单看做对 Linux 的 Namespace、Cgroup、镜像管理文件等 Linux 系统内核功能的封装。Docker 与 KVM 虚拟化技术不同，它没有加入 Hypervisor 虚拟监控层来进行环境模拟，而是类似一个进程直接使用 Linux 操作系统的相关功能来完成，其中 Namespace 负责实现系统环境的隔离，Cgroup 实现资源的限制，镜像实现目录环境的隔离^[36]。

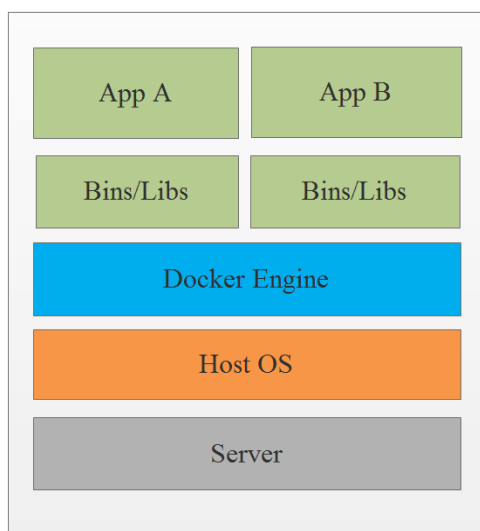


图 2-8 Docker 架构图

2.4 虚拟网络功能

在传统的运行商网络中,大量的专用设备提供着各自的网络功能,这些专有的硬件设备一般使用特定的 ASIC 芯片,且由于设备专有性强,控制性和扩展性较差,所以 NFV 针对这一情况将虚拟网络功能利用软件来实现,并部署在通用服务器上。

NFV 是将传统网络中部署在专有硬件设备的网络功能部署在通用服务器上,网络功能提供的服务由软件来实现。在一个传统网络中,专有的网络设备常包含路由器(Router)、交换机(Switch)、防火墙(Firewall)、负载均衡器(Load balancer)、代理服务器(Proxy)和网络地址转换器(NAT)等设备。这些专用设备一般分别位于不同位置,来满足客户不同的需求。

而对于虚拟网络功能,常见的实现软件有 Nginx、Snort、Firestorm 等软件。本文选取了两个常用软件 Nginx 和 Snort 来提供网络功能实验。Nginx 是一个可以对外提供高性能服务的 HTTP 服务器和反向代理服务器,通常可以被用来作为一个负载均衡功能的实现者对外提供服务^[37]。Snort 是一个网络入侵检测系统的软件实现,它可以对数据流量进行实时的检测分析,可以对监测到的流量包进行内容匹配,也可以根据不同的攻击方式即时报警。此外,Snort 是开源的软件,使用时十分简单方便^[38]。

2.5 本章小结

本章首先介绍了 NFV 的设计架构,对 NFV 系统中的三个重要组成部分 NFVI、VNFs 以及 VNF MANO 进行了详细阐述,接着分析了在具体的部署环境下,NFV 面临的性能瓶颈,并对其中主要的影响因素——数据转发面进行了阐述。随后介绍了当前两种常用的虚拟化方式 KVM 虚拟化技术和 Docker 容器技术,阐述了两者的实现原理。最后介绍了在 NFV 环境中两种常用的虚拟网络功能软件 Nginx 和 Snort。

第三章 面向网络功能虚拟化的数据平面的对比

数据平面作为网络功能虚拟化实际部署环境中至关重要的一层，其性能表现将对整个平台的性能表现有着重要的影响，如何选择合适的平面部署在 NFV 实际场景中，需要综合考虑其性能表现和使用条件。

3.1 网络功能虚拟化部署框架

在 NFV 的实际应用中，部署虚拟网络功能需要结合 X86 通用服务器的实际物理环境来综合考虑，典型的网络功能虚拟化部署架构图如图 3-1 所示。在网络流量通过一台通用服务器时，逻辑上会经过三个层次：底层设备转发层、虚拟交换层和虚拟网络功能层。底层设备转发层一般部署传统物理交换机或 OpenFlow 交换机，它们可以根据特定的转发规则将其他设备转发来的包传递给宿主机中的物理网卡。虚拟交换层可以将宿主机中的物理网卡接收到的报文转发给上层具体的网络功能。在虚拟网络功能层，运营商可以部署相应的虚拟网络功能，依靠如 Nginx、Snort、Firestorm 等软件来实现。

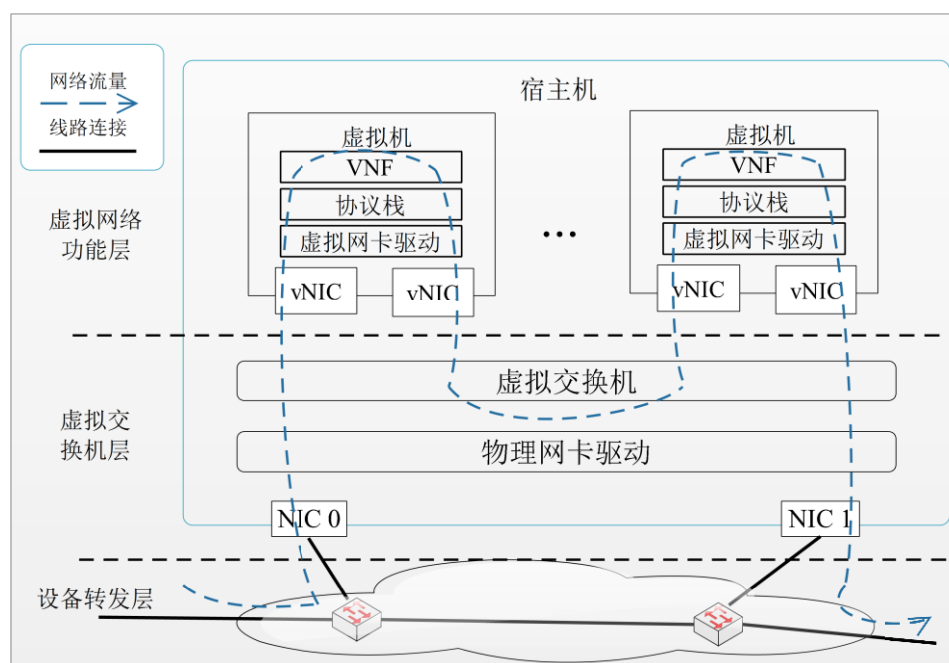


图 3-1 网络虚拟化部署架构图

3.1.1 设备转发层

在设备转发层，既可以使用传统的转发设备，也可以使用支持 OpenFlow 协议

的特定 SDN 交换机。SDN 交换机作为白牌交换机，将控制面与转发面解耦和，控制交由远端的 SDN 控制器完成，交换机本身只专注于数据的转发，所以其数据处理的性能比较高，多用于 NFV 场景下。

如图 3-2 所示，SDN 交换机主要两部分组成：安全通道和流表。SDN 交换机通过安全通道与特定的 SDN 控制器进行通信，通过流表中的流表项来对流量进行特定规则的转发。

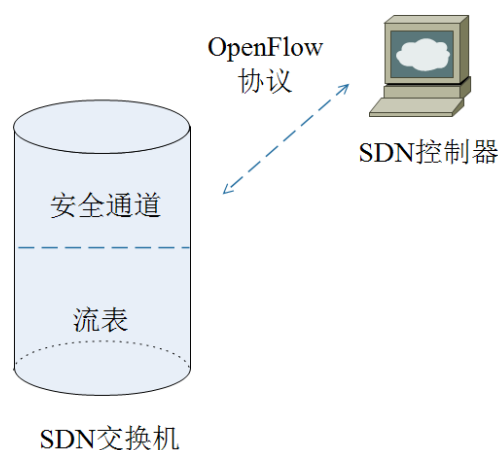


图 3-2 SDN 交换机结构

3.1.2 虚拟交换机层

虚拟交换机层是部署在 NFV 虚拟网络功能和物理设备层之间的桥梁，它负责将物理网络适配器收到的包转发给上层的虚拟机或容器环境。在使用 Hypervisor 的虚拟机环境中，虚拟交换机层在 Hypervisor 和虚拟机之间运行，提供一个 I/O 虚拟化层。Hypervisor 通过内核网卡驱动程序接收到来自网络适配器的数据包，再通过虚拟交换机重定向转发给虚拟机的虚拟网卡。虚拟网卡和物理网卡的端口都连接在虚拟交换机上，通过设定特别的转发规则，可以完成数据的传输。

因为虚拟交换机使用了由软件实现的转发技术，所以在宿主服务器中的转发性能就成为了造成 NFV 实际部署中的主要性能瓶颈。网络数据流量流经物理网卡，通过虚拟交换机到达虚拟网卡这个过程中，操作系统会进行大量的中断、内核上下文切换、复制内存内容等大量的 CPU 动作，这些动作造成了极大的性能损耗。

在当前技术中，物理网卡在接收到数据包后，一般采用直接存储器读取的方式直接写入宿主机内存并强迫 CPU 产生中断。在 NFV 的应用场景中，如果在多块网卡连接在同一个 CPU 内核情况下，这将导致 CPU 频繁进入中断状态。因此，针对网卡性能的改进主要通过关闭或者减少中断，或使用多核 CPU 依靠负载均衡来提高 NFV 在宿主机部署中的性能。

所以，在 NFV 场景下，当物理网卡捕获到数据包后，通用服务器上 Linux 系统内核利用物理网卡驱动读取网络数据，然后内核将读取到的网络数据复制给运行在用户态的 Hypervisor 进程，Hypervisor 再将数据转发给虚拟环境。虚拟机中的虚拟网卡默认 tap 模式，利用 tap 网卡驱动，虚拟网卡可以将内核协议栈中处理好的网络报文转发给另一个使用 tap 驱动程序的进程，如 Hypervisor 和 VM 对应的进程。DPDK-OVS, NetMap 和 Click 三种开源软件作为当前技术中最常使用的虚拟交换机，具有提高数据包转发性能的能力。

3.1.3 虚拟网络功能层

虚拟网络功能通过软件的形式对外提供服务，通常被部署在虚拟机中或容器中。在当前的应用中，Nginx 和 Snort 作为两个常用的网络功能软件，在 NFV 的部署中得到了广泛的使用。也可以使用一些其他的软件分别来实现 Router、Switch、Firewall、Load balancer、Proxy 和 NAT 等网络功能。

3.2 实现 DPDK-OVS 数据面

DPDK 和 OVS 作为两种目标不同的技术手段，它们的设计初衷是不同的。

DPDK 是 Intel 公司开发的数据平面开发工具套装，主要部署于 X86 通用平台，提供库函数和驱动以使用户空间可以高效地处理数据包。在宿主机部署了 DPDK 模块后，物理网卡接受到的数据包可以不经过操作系统内核协议栈的处理直接与用户空间进行交互，从而实现数据包的转发加速目的。而 OpenvSwitch 作为一个高质量的，多层虚拟交换机，旨在解决现有物理交换机的一些瓶颈，比如虚拟交换机利用了软件开发，可以同时对外提供多个端口，相比于传统交换机具有更好的扩展性。同时可以支持多种协议，对转发功能进行保证。

而二者的缺点也很明显，OpenvSwitch 的转发效率较低，无法满足在 NFV 的应用场景下网络数据包的快速转发，而 DPDK 提高了网络数据包的转发速率，但是流量控制手段单一，流量控制结构灵活性差，以及流量无法实现细粒度分类。因此，将二者结合在一起使用可以满足当前环境下 NFV 的快速转发以及流量控制的需求。

3.2.1 部署 DPDK

如图 3-3 所示，在实际部署中，DPDK 主要由两部分组成：环境抽象层（Environment Abstraction Layer）和基本组件部分，其中基本组件包括内存管理组件、无锁环形队列组件、PMD（Physical Media Dependent）驱动组件、数据包分流

组件等。

环境抽象层为用户进程提供了一个通用接口，屏蔽掉了与底层库与设备通信的相关细节。EAL 执行了 DPDK 运行过程中的初始化工作，基于 Hugepage 的内存分配，多核属性设置，原子化的锁操作，并完成 PCI 物理地址到用户空间的映射，方便应用进程使用。

内存管理组件提前在 EAL 中获得设置好大小的内存对象，避免了在 DPDK 运行过程中需要即时执行内存分配和垃圾回收，这样大大提高了数据的转发效率，通常被当做数据包的缓存来进行使用。

无锁环形队列组件进一步优化的无锁先入先出模式的环形队列，在生产者消费者模型中，他可以使竞争资源避免等待，并且支持高并发的无锁操作。

PMD 驱动组件在 Intel 1Gb、10Gb 以及 40Gb 等产品的使用中，可以根据轮询来进行包的收发，大幅提高了网卡收发数据包的性能。

数据包分流组件利用 Intel SSE 根据多元组进行了高效率的哈希算法，这样可以将数据包进行更加快速的分类，以便后续处理。

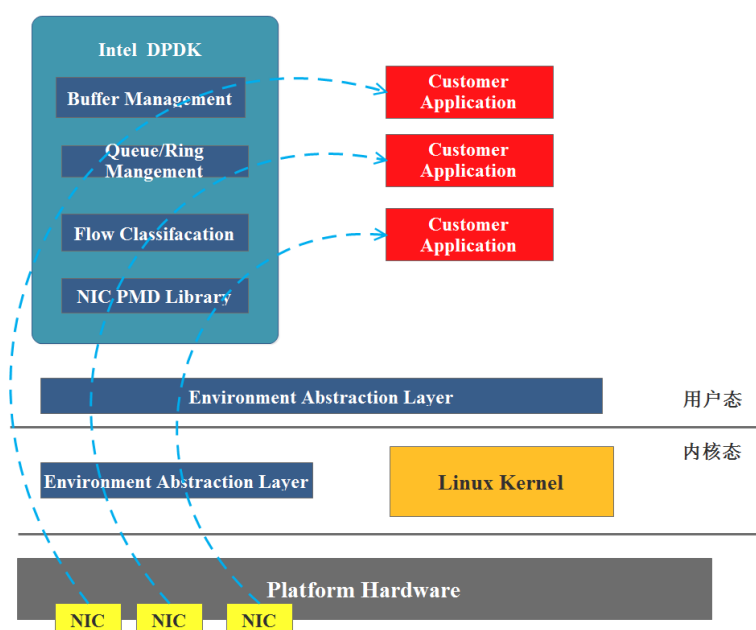


图 3-3 DPDK 架构图

部署 DPDK 第一步要对环境抽象层进行初始化，环境抽象层初始化步骤如图 3-4 所示。

第一步，设置环境抽象层初始化动作；

第二步，设置环境抽象层的 CPU 子组件，设置 CPU 各个核心的相关配置，同时进行配置信息的保存；

第三步,设置环境抽象层的 PCI 子组件,创建 PCI 设备队列和 PCI 驱动队列,轮询 PCI 总线,以及设备文件中的设备;

第四步,设置 EAL 内存管理子组件,预设大页内存,在物理设备上留取一些连续的内存空间,为之后设置 MemZone 数据结构;

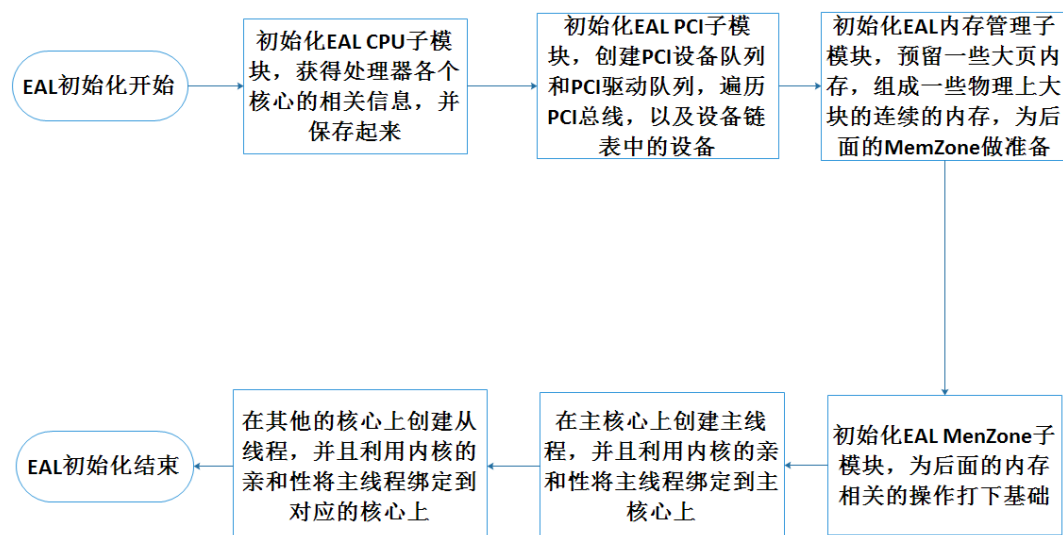


图 3-4 EAL 初始化流程

第五步,配置 EAL MenZone 数据结构,准备后续的内存操作;

第六步,创建主线程,并将主线程绑定到主核心之上运行;

第七步,在其他的核心上创建从线程;

第八步,设置环境抽象层初始化结束。

在其中, Hugepage 配置初始化和 MemZone 配置初始化是环境抽象层配置的关键步骤。

3.2.1.1 Hugepage 初始化

HugePage 初始化配置可以分为三个步骤:

第一步,同步系统大页内存的相关配置信息。这一步是找到系统中的 hugetlbfs 目标挂载点,在一个支持大页内存的系统中,只有挂载了对应的文件系统,才能使用大页内存。具体过程如图 3-5 所示。

第二步,利用 hugetlbfs 对连续空闲的大页内存页进行指派。即在 hugetlbfs 文件系统下根据文件对内存做映射,DPDK 中首先将单个目标文件映射大小等于一个巨页的单位大小,可以为 1M 大小,也可以是 1G 大小。其次在内存映射时会优先选择映射物理内存地址连续的空间,其次再去选择虚拟内存地址连续的空间。完成之后在 hugetlbfs 文件系统目录下会对应生成一些文件,这些文件都是映射了实

际内存的。

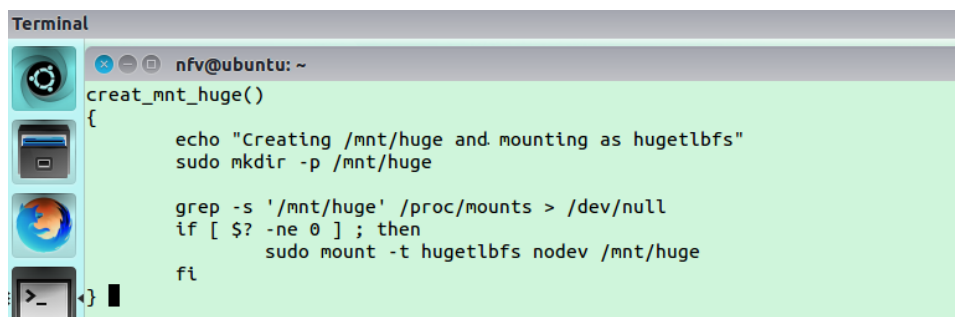


图 3-5 创建 Hugepage 文件系统

第三步，将所有可以使用的内存一并分类、整理，作为使用 MemSeg 结构记录，之后在第二步映射好的内存中寻找连续地址的内存空间，然后将这部分空间用 MemSeg 数据结构来表示。其过程如图 3-6 所示。

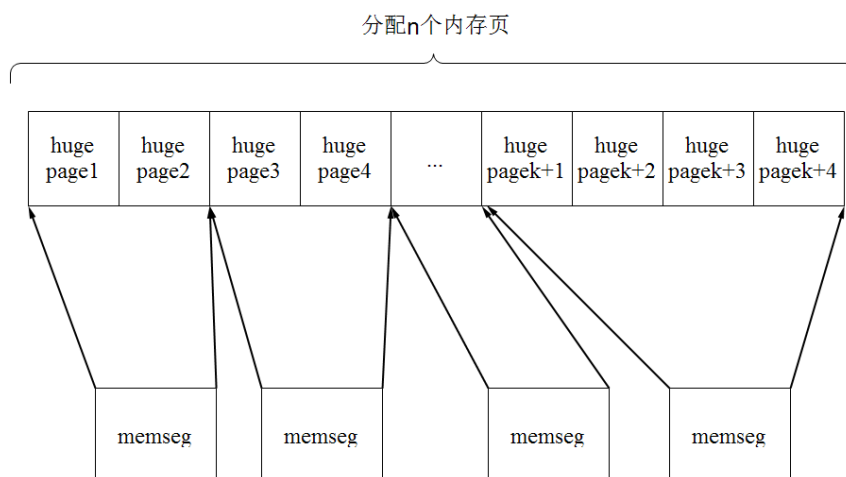


图 3-6 使用 MemSeg 结构

3.2.1.2 MemZone 初始化

在 MemZone 子模块的初始化配置中，最关键的是生成如图 3-7 所示的内存结构，定义一个 head 和 end 来表示 MemSeg 所表示的内存区域，接着使用一个内存链表，将这些 MemSeg 结构放入到这个内存链表之中。DPDK 中按照大小的不同，将内存区域分成了 13 类，使用一个 free_head 的数组保存了 13 个内存链表的入口，每个链表会在初始化配置时装入相应的 MemSeg，如果没有使用 DPDK 时，这些结构是放入内存堆中，DPDK 中将堆用链表的方式实现了。

内存的初始化结构为 MemSeg，由 MemSeg 来搭建成内存堆，使用 MemZone 来管理内存的分配与释放，通过 MallocHeap 堆来分配内存。即 MemZone 记录了哪些内存已经被分配使用。在 DPDK 中，所有内存管理的实现都依赖于上述机制，

这个机制是其内存管理的核心所在。

使用 MemZone 来进行内存分配的步骤为：第一步，在内存堆中寻找容量合适的内存块；第二步，对找到的内存块进行分配，如果分配完成后，这个内存块仍然有剩余的容量，则将剩余容量作为新的内存加入到内存堆中以供后续使用；最后一步，查询 MemZone 的可用位置，将已分配的内存配置信息写到 MemZone 中。

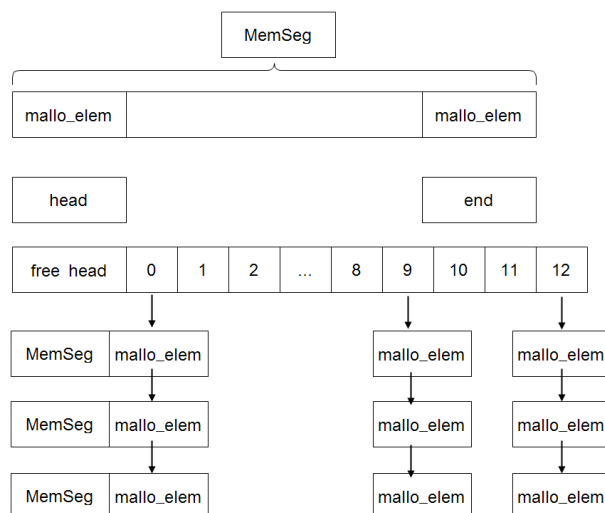


图 3-7 MemSeg 内存结构

使用完 MemZone 后，其内存释放过程步骤为：第一步，查询指定指针映射到的 MemZone 并将 MemZone 信息置零；第二步，将该指针指向的内存清空后重新加入内存堆中，如果可以进行合并，则将物理地址连续的内存空间合并，即多个连续地址的空间内存之间需要整合成一块大的内存加入内存堆中。

3.2.2 整合 DPDK 与 OpenvSwitch 及优化

DPDK 的部署，可以使得物理网卡和宿主机用户空间之间的数据包交互更加快速，绕开操作系统内核态的处理，从而实现加速转发。而 OpenvSwitch 较物理交换机而言有着更低的成本和更高的工作效率。而二者的缺点也很明显，OpenvSwitch 的转发效率较低，无法满足在 NFV 的应用场景下网络数据包的快速转发，而 DPDK 提高了网络数据包的转发速率，但是流量控制手段单一，流量控制结构灵活性差，以及流量无法实现细粒度分类。因此，将二者结合在一起使用可以满足当前环境下 NFV 的快速转发以及流量控制的需求。

如图 3-8 所示，单独部署 OpenvSwitch 时，其工作在操作系统的内核空间中，在与客户机的虚拟化 I/O 进行数据交互的时候会经过多次内核态和用户态的上下文切换，这导致了额外的性能开销。而借助 DPDK 与 vhost-user 的优势，将其与 OpenvSwitch 结合部署，可以实现从虚拟机到宿主机物理网卡间的数据零拷贝。组

合之后虚拟机之间的数据交互以及物理网卡到虚拟机虚拟网卡之间的数据交互都将工作在用户空间中，使数据转发的性能大大提高。

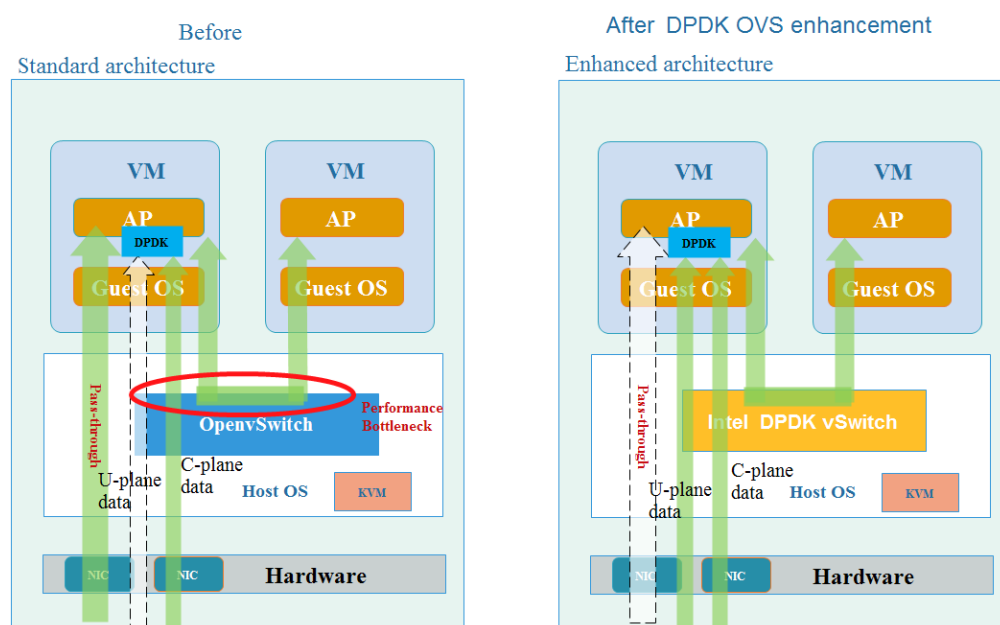


图 3-8 传统 OpenvSwitch 部署与 DPDK-OVS 部署对比

在实际的工程上，DPDK 和 OpenvSwitch 作为两个独立的组件，需要按照特定的步骤进行整合才可以实现对外提供预期的功能。其整合步骤如下：

第一步，需要进行如上节所示的 DPDK 的初始化部署；

第二步，为方便后续的修改使用，这里需要对 OpenvSwitch 进行源码的编译安装，过程如图 3-9 所示；

```
# 解压编译
[nfv@nfv]# tar -zxvf openvswitch-2.7.0.tar.gz
[nfv@nfv]# cd openvswitch-2.7.0/
[nfv@dpdk openvswitch-2.7.0]# ./boot.sh
[nfv@dpdk openvswitch-2.7.0]# ./configure \
--with-dpdk=/usr/src/dpdk \
--prefix=/usr \
--exec-prefix=/usr \
--sysconfdir=/etc \
--localstatedir=/var
[nfv@dpdk openvswitch-2.7.0]# make
[nfv@dpdk openvswitch-2.7.0]# make install
```

图 3-9 对 OpenvSwitch 进行源码编译及配置

第三步，解压后进入相关的目录对相关的组件进行编译安装，完成 OpenvSwitch 的部署；

第四步，将所需要使用的物理网卡绑定到 DPDK 进程中，为之后的使用做好准备，我们将 eth0 和 eth1 绑定到 DPDK 中，使用相关系统终端指令将无法看到这两个网卡配置，只有进入 DPDK 信息栏才可以看到，绑定成功如图 3-10 所示：

```
Network devices using DPDK-compatible driver
=====
0000:03:00.0 '82571EB Gigabit Ethernet Controller' drv=vfio-oci unused=
0000:03:00.0 '82571EB Gigabit Ethernet Controller' drv=vfio-oci unused=

Network devices using kernel driver
=====
0000:00:1f.6 'Ethernet Connection (2) I219-V' if=enp0s31f6 drv=e1000e unused=vfio-pci
0000:02:00.0 'BCM4352 802.11ac Wireless Network Adapter' if= drv=bcma-pci-bridge unused=vfio-pci

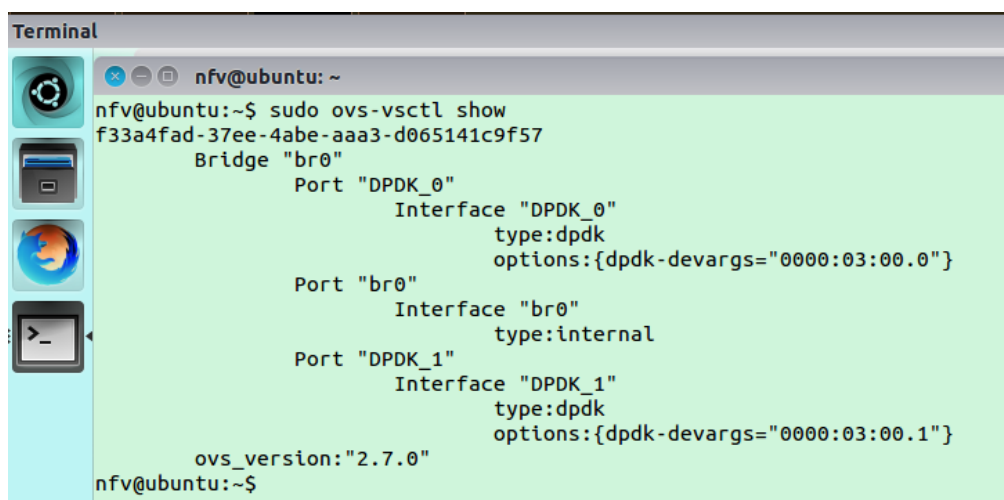
Other = network devices
=====
<none>
```

图 3-10 绑定网卡至 DPDK

第五步，配置大页，并将其页数设置为 1024 个，设置每个大页的内存为 2M 大小，所以总的大页将使用 2G 内存，满足当前宿主机条件，之后完成大页挂载到 hugetlbfs 上；

第六步，启动 OpenvSwitch，设置 OVS 数据库参数来开启对 DPDK 功能的支持，并启动 OVS-vswitchd 进程；

第七步，创建 OpenvSwitch 网桥，连接远端 SDN 控制器，并将 DPDK 端口加入到网桥之中，结果如图 3-11 所示。至此，DPDK 与 OpenvSwitch 的整合完成。



```
Terminal
nfv@ubuntu: ~
nfv@ubuntu:~$ sudo ovs-vsctl show
f33a4fad-37ee-4abe-aaa3-d065141c9f57
Bridge "br0"
  Port "DPDK_0"
    Interface "DPDK_0"
      type:dtpk
      options:{dtpk-devargs="0000:03:00.0"}
  Port "br0"
    Interface "br0"
      type:internal
  Port "DPDK_1"
    Interface "DPDK_1"
      type:dtpk
      options:{dtpk-devargs="0000:03:00.1"}
ovs_version:"2.7.0"
nfv@ubuntu:~$
```

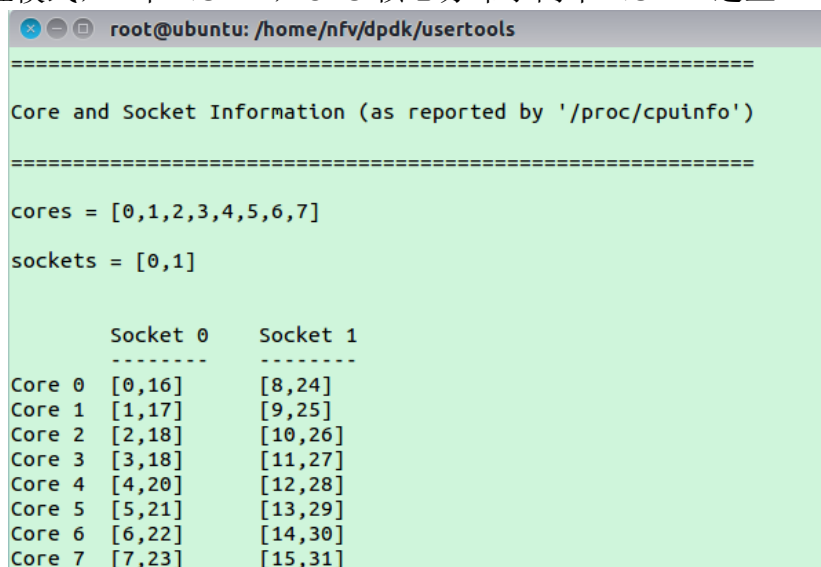
图 3-11 创建网桥并加入 DPDK 端口

之后可以根据具体的需要，为相应的 OpenvSwitch 网桥创建流表，来控制网络流量的转发途径。

在完成 DPDK 和 OpenvSwitch 的整合后，尽管 DPDK 会对 OpenvSwitch 处理数据包的过程进行加速，但由于仍未充分利用物理宿主机的硬件资源，其性能仍然

具有一定的提升空间。我们根据二者的使用特性，针对其部署及设置过程提出优化方案，主要为 CPU 核规划以及核绑定，硬件配置优化、系统程序优化、Filter 配置优化以及 OpenvSwitch 与 CPU 核优化。

第一步，根据物理硬件资源进行 CPU 核资源规划以及绑定 CPU，专核专用，从而提高宿主机性能。首先在部署完 DPDK 之后查看当前 CPU、NUMA 等硬件资源，如图 3-12 所示，可以看到当前测试所用物理宿主机有 2 个物理 CPU，CPU 已开启超线程模式，2 个 NUMA，CPU 核心分布于两个 NUMA 之上。



```

root@ubuntu: /home/nfv/dpdk/usertools
=====
Core and Socket Information (as reported by '/proc/cpuinfo')
=====

cores = [0,1,2,3,4,5,6,7]
sockets = [0,1]

          Socket 0      Socket 1
          -----      -
Core 0  [0,16]        [8,24]
Core 1  [1,17]        [9,25]
Core 2  [2,18]        [10,26]
Core 3  [3,18]        [11,27]
Core 4  [4,20]        [12,28]
Core 5  [5,21]        [13,29]
Core 6  [6,22]        [14,30]
Core 7  [7,23]        [15,31]

```

图 3-12 查看 CPU 资源

接着根据当前的 CPU 资源，对 CPU 核进行进程绑定，使我们的测试进程独立运行在 CPU 核心上，为预留资源以供后续部署虚拟机时使用，我们将 OpenvSwitch 的主进程绑定在 NUMA0 和 NUMA1 的 4 个核心之上，OpenvSwitch-PMD 线程绑定在 NUMA0 和 NUMA1 的 4 个核心之上，这样既使得数据平面进程高效运行，也为后续的测试留下了充足的物理资源，具体绑定如图 3-13 所示。

	Socket 0	Socket 1	
	-----	-----	
Core 1	[0,16]	[8,24]	
Core 1	[1,17]	[9,25]	
Core 2	[2,18]	[10,26]	->OpenvSwitch
Core 3	[3,18]	[11,27]	->OVS-PMD
Core 4	[4,20]	[12,28]	
Core 5	[5,21]	[13,29]	
Core 6	[6,22]	[14,30]	
Core 7	[7,23]	[15,31]	

图 3-13 CPU 资源绑定

第二步，硬件配置优化是在物理宿主机 BIOS 设置中。首先启动 VT-d，开放物理硬件管辖权给虚拟机，使得虚拟机可以独占一些物理资源；其次关闭 CPU C3 和 C6 state，以使物理宿主机工作在高性能状态；最后调整 CPU 功率策略为 performance，使 CPU 不进行空闲睡眠，以提高 CPU 性能表现。

第三步，系统程序优化，为避免系统自带优化程序对内存以及 CPU 的不必要占用，我们选择关闭 irqbalance 服务，理论上 irqbalance 服务可以优化系统中断分配，但是效果较差，并且会带来内存以及 CPU 利用率的上升，对测试的 DPDK-OVS 数据转发平面性能造成不必要的影响。

第四步对 Filter 配置进行优化，保证我们所设定的 NUMA 被过滤出并可以生效，具体改进配置如图 3-14 所示。

```
[filter scheduler]
enabled_filters=RetryFilter,AvailabilityZoneFilter,ComputeFilter,ComputeCapabilitiesFilter,ImagePropertiesFilter,ServerGroupAntiAffinityFilter,ServerGroupAffinityFilter,PciPassthroughFilter,AggregateInstanceExtraSpecsFilter,AggregateCoreFilter,NUMATopologyFilter
```

图 3-14 Filter 配置优化

第五步针对 OpenvSwitch 进行核绑定规划。根据第一步针对 CPU 核资源规划，我们确定 OpenvSwitch 的主进程绑定在核 2，18，10，26 之上，OpenvSwitch-PMD 进程绑定在核 3，19，11，27 之上。配置内存及其他数据平面收发网络包参数，并手动配置 OVS-vswitchd 的 CPU 亲和性，优化过程如图 3-15 所示。

```
#python -c "print '%x'(((1<<2)|(1<<18)|(1<<10)|(1<<26)))"
4040404
#ovs-vsctl --no-wait set Open_vSwitch.other_config:dppk-lcore-mask=0x4040404
#python -c "print '%x'(((1<<3)|(1<<19)|(1<<11)|(1<<27)))"
8080808
#ovs-vsctl --no-wait set Open_vSwitch.other_config:dppk-lcore-mask=0x8080808
#ovs-vsctl set Open_vSwitch.other_config:dppk-init="ture"
#ovs-vsctl set Open_vSwitch.other_config:dppk-alloc-mem="4096"
#ovs-vsctl set Open_vSwitch.other_config:dppk-socket-mem="4096,4096"
#ovs-vsctl set Open_vSwitch.other_config:dppk-rxqs="32"
#ps aux|grep ovs-vswitchd
root    21228  799  2.1 20557928 256484?  S<Lsl Feb27 14361:04
ovs-vswitchdunix:/var/run/openvswitch/db.sock -vconsole:emer -vsyslog:err -vfile:info --nlo
ckall --no-chdir --log-file=/var/log/openvswitch/ovs-vsitchd.log
--pidfile=/var/run/openvswitch/ovs-vswitchd.pid --detach
root    25480  0.0  0.0 112652  972 pts/1    S+   14:53  0:00  grep  --color=auto ovs-vswitc
hd
#taskset -cp 2,18,10,26 21228
pid 21228's current affinity list:2,10,18,26
pid 21228's new affinity list:2,10,18,26
```

图 3-15 OpenvSwitch 配置优化过程

DPDK 软件和 OpenvSwitch 软件作为两款不同的软件，需要协同部署，相互搭配。尽管如此，安装过程中仍不可避免的会因为软件原因造成相关的程序错误，所以需要 OpenvSwitch 进行源码的部署，以便后续的改动。

3.3 实现 NetMap 数据面

3.3.1 NetMap 架构

NetMap 是一个高速的网络 I/O 框架，它使用了零拷贝的思想来实现网络包在到达物理网卡后的加速目标，能后在当前使用广泛的千兆网卡上达到以线速率的标准接收和发送网络数据包。

零拷贝技术的实现是通过降低网络通信中的数据拷贝或者线程分享实现的，它使得宿主机和路由器与网络适配器进行通信时，有效的减少了 CPU 在内存间进行数据拷贝的次数，这可以使网络通信过程中不必要的拷贝过程不再发生，大大提高了通信效率和节省了内存空间。

当宿主机部署了 NetMap 框架后，可以使物理网卡进入 NetMap 模式。传统宿主机中网卡环会与操作系统中的网络协议栈解耦和，并拷贝一份网卡环交给 NetMap 虚拟交换机进程使用。NetMap 在拥有 NetMap 环后，还会新建一对环并负责对其维护，这对环将负责与宿主机网络协议栈进行网络通信。NetMap 环和新建的一对通信环均在内存的共享空间实现，可以用来对网络数据包进行缓存。用户空间中的进程可以通过调用 NetMap 向外提供的 API 来获取 NetMap 环中的数据包内容。

通过这种 NetMap 的实现，用户空间中的进程可以实现依靠 NetMap API 来对缓存中的数据包直接访问，从而绕开操作系统中的协议栈，实现了网络包从内核态到用户态的零拷贝。与此同时，NetMap 还会对内核中的相关内存区进行屏蔽，这样可以保护相关的内存区域不被用户态的进程误操作，造成系统的崩溃。

在这个基础上，NetMap 还提供了以下几种方式来提高网络吞吐的性能：NetMap 交换机可以根据用户的需要，对内存进行固定大小的分配，之后用来存储相关的数据包，这样使用提前静态分配的方法，可以有效避免当数据包到达网卡时，需要动态的对内存进行分配，导致性能额外的开销；NetMap 可以使用一些轻量级的元数据组来表示硬件的相关特性，这样可以屏蔽底层硬件对于上层操作系统的接口，这些元数据可以表征底层硬件的相关特性，如可以支持一次系统调用可以转发更大量的网络数据包，这样有效地降低了多余的系统调用所带来的额外的性能开销。

NetMap 是依靠三个关键组件来完成网络包的零拷贝快速转发的，三个组件分别是 `packet_buffer` 组件、`netmap_ring` 组件和 `netmap_if` 组件。这三个组件可以由用户态的进程观察到，并且这三个组件在内存态中会被操作系统内核放置到同一块内存区域，以便所有的用户进程完成共享，方便在不同进程中实现网络数据包的零

拷贝转发。NetMap 组件的结构图如图 3-16 所示。

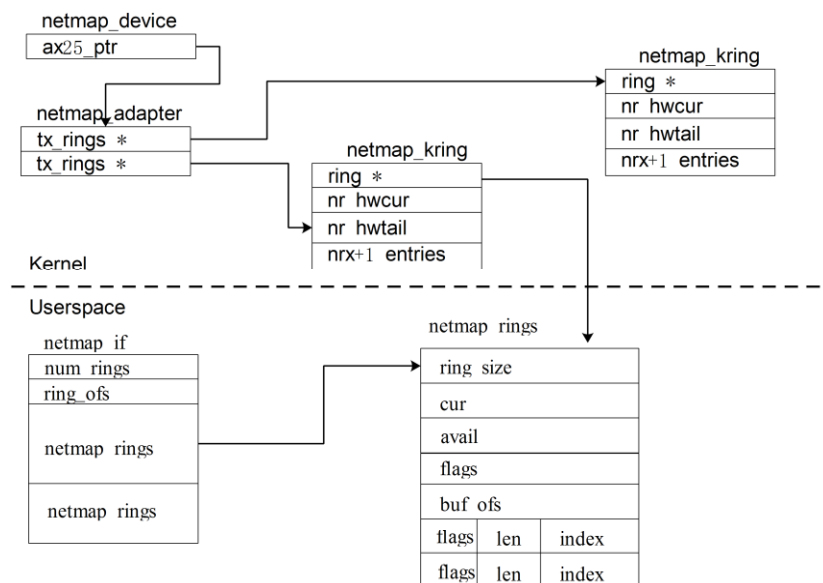


图 3-16 NetMap 组件结构图

其中，`packet_buffer` 的大小通常是已经设置好的，一般为 2KB，这个组件由物理网卡和用户空间中的进程共享。每一个 `packet_buffer` 都包含一个索引，这个索引可以被用户空间中的进程或者是操作系统内核用来变换为虚拟地址，也可以被物理网卡用作物理地址。当网卡被设置到进入 NetMap 模式后，`packet_buffer` 会进行提前的静态分配，在这个过程中，每一个 `packet_buffer` 都会被映射到一个 NetMap 环和一个相应的物理网卡环。

`netmap_ring` 是处于物理网卡之中的一个独立的数据备份区域。它可以包含以下几种内容：`ring_size`、`cur`、`avail`、`buf_ofs` 和 `sots[]` 等。其中，`ring_size` 表示 `netmap_ring` 数据结构中可以包含的 slot 总量，`cur` 表示 `netmap_ring` 中当前进程进行读写的标识位，`avail` 表示当前条件下可以使用的缓存的总量，`buf_ofs` 表示当前 `netmap_ring` 结构中缓存区域数据开始位置的偏移量，`sots[]` 表示缓存中数据包的长度，数据包的标签以及数据缓存区中的一些索引。

而在 `netmap_if` 结构中，它向外提供了一些相关接口的信息，这些信息仅仅由用户空间进程享有只读权限，例如 `ring` 的数目等等信息。

此外，NetMap 在内核空间中针对每一个物理网卡（即每一个 `net_device`）创建一个 `netmap_adapter` 结构，该结构主要被用来负责在这块物理网卡上当前环境中的所有可用网络数据包接收队列以及网络数据包发送队列。`netmap_kring` 是内核空间中用来描述一个缓冲队列的结构，其实际指向的是 `netmap_ring` 结构的一个数据结构，只有在内核空间中才能由操作系统内核来进行遍历。

3.3.2 NetMap 初始化配置以及结构改进

在部署 NetMap 虚拟交换机时，内存池初始化步骤如下：

第一步，进行 `netmap_adapter` 容量的申请，比如使用的是 `e1000` 网卡，首先调用 `e1000_probe()` 函数进行查询，之后调用 `netmap_attach()` 函数，申请 `netmap_adapter` 内存并且设置 `ifp` 函数 `WNA(ifp)=na`；将信息保存至网络设备 `netdevice` 的 `ax25ptr` 指针中；

第二步，在 `linux_netmap_open` 接口中申请 `netmap_priv_d` 数据结构 `priv`，然后进行存储，定位在 `file->private_data` 域中；

第三步，通过进入函数 `ioctl(NIOCGINFO)` 中，对相关参数进行调用依次连接到 `linux_netmap_ioctl` 以至 `netmap_ioctl` 之中的 `CaseNIOCGINFO` 结构中，进而将上一步申请的结构作为参数传递到函数 `netmap_get_memory_locked(priv)` 中。此时，`priv` 中的 `np_ifp` 字段为 `NULL`，因此 `NA` 为 `NULL`，所以得到 `nmd=nm_mem`；

第四步，接下来进入函数 `netmap_mem_global_finalize`，在该函数中先调用 `netmap_memory_config` 方法对参数进行相关初始化，比如页对齐等，然后进入函数 `netmap_mem_finalize_all`，调用函数方法 `netmap_finalize_obj_allocator` 对内存完成规划，在虚拟地址和物理地址之间完成映射，在进入配置文件进行全局配置，成果图如图 3-17 所示；

```
netmap_buf_size = nmd->pools[NETMAP_BUF_POOL]._objsize;
netmap_total_buffers = nmd->pools[NETMAP_BUF_POOL].objtotal;

netmap_buffer_lut = nmd->pools[NETMAP_BUF_POOL].lut;
netmap_buffer_base = nmd->pools[NETMAP_BUF_POOL].lut[0].vaddr;
```

图 3-17 Netmap 全局配置

第五步，接下来在函数 `ioctl(NIOCREGIF)` 中通过 `get_ifp` 方法捕获到网络设备 `net_device` 的 `ifp`，然后调用 `netmap_do_regif` 中，将 `priv->np_ifp` 的值设置为 `ifp` 当前值；将当前的网络设备 `net_device` 保存至 `file->private->np_ifp` 结构中，再通过 `ifp` 进行指针操作捕获到 `netmap_adapter` 数据结构的具体信息；

最后一步，通过调用函数 `netmap_if_new` 使用方法 `netmap_mem_if_new` 对保存在 `netmap_adapter` 之中的 `netmap_ring` 进行分配，调用 `netmap_new_bufs` 方法给每个 `netmap_ring` 的每个 `slot` 字段向 `netmap` 缓存区申请内存。

在实际 NetMap 虚拟交换机的部署中，我们通过对源码的解读，看到 `netmap_ring` 这个数据结构中是依靠 `cur` 来提供缓存位置的信息，通过 `avail` 来提供当前条件下可以使用的缓存的总量，仅仅依靠这两个变量位置，也使得用户空间进程和操作系统内核在对 `netmap_ring` 进行读取数据和写入数据时，要分别对这两个

变量进行修改，同时还要保持其他线程不会修改这两个字段。这就导致了在 NFV 场景下，当网络数据包需要进行快速转发的时候，会面临一定的性能瓶颈，由于用户进程频繁地改变字段，使得在内核态和用户态之间不断地进行切换，产生了额外的性能损耗。我们针对这一情况，对 netmap_ring 数据结构进行了改进。

在 netmap_ring 这一数据结构中，增加了 NetMap_tail 和 NetMap_head 两个变量两个来实现环形队列，这样用户进程在处理数据时可以避免调用系统调用来修改相关变量，Netmap_tail 变量和 Netmap_head 变量分别交给用户进程和内核来进行识别、修改。伪代码添加和删除数据如图 3-18 所示。

```
/*向无锁队列中添加数据*/
Put(Type data)
{
    if(Full())
        return(-1);

    *NetMap_tail = data;
    if(++NetMap_tail >= pQueueEnd)
        NetMap_tail = pQueueBase;
    return(0);
}
```

(a)

```
/*对无锁队列中删除数据*/
Type *Get()
{
    Type return_val = Null;
    if(Empty() == False)
    {
        return_val = *NetMap_head;
        if(++NetMap_head >= pQueueEnd)
            NetMap_head = pQueueBase;
    }
    return(return_val);
}
```

(b)

图 3-18 无锁队列伪代码操作。(a)添加数据操作；(b)删除数据操作

修改后的 netmap_ring 数据结构在系统中显示如图 3-19 所示。

我们在保留原有 ring_size、cur、avail、flags 等字段的基础上，对 netmap_rings 数据结构增加了新的字段 NetMap_tail 和 NetMap_head 两个变量，通过在添加数据时对 NetMap_tail 字段继续访问和删除数据时对 NetMap_head 进行访问，实现无锁队列，从而提高其数据结构在高并发下的访问速度。

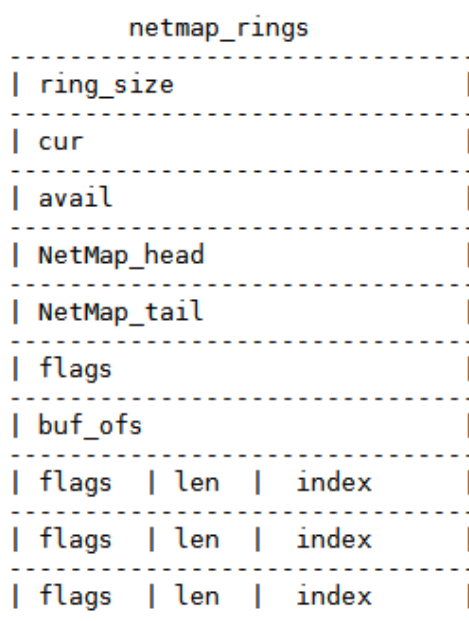


图 3-19 新 netmap_ring 结构图

配置完成后运行一个 NetMap 测试用例，成功运行结果如图 3-20 所示。

```

nfv@ubuntu: ~/netmap/LINUX/build-apps
nfv@ubuntu:~$ cd netmap/LINUX/build-apps
nfv@ubuntu:~$ ./pkt-gen -i eth0 -f tx
843.803242 main [1930] interface is eth0
843.806747 main [2050] running on 1 cpus (have 2)
843.808433 extract_ip_range [367] range is 10.0.0.1:0 to 10.0.0.1:0
843.808493 extract_ip_range [367] range is 10.1.0.1:0 to 10.1.0.1:0
843.813622 main [2148] mapped 334980KB at 0xa3080000
Sending on netmap:eth0: 1 queues, 1 threads and 1 cpus.
10.0.0.1 -> 10.1.0.1 (00:00:00:00:00:00 -> ff:ff:ff:ff:ff:ff)
843.813828 main [2233] Sending 512 packets every 0.000000000 s
843.813851 main [2235] Wait 2 secs for phy reset
845.814327 main [2237] Ready...
845.816526 sender_body [1211] start, fd 3 main_fd 3
846.816691 main_thread [1720] 69.345 Kpps (69.424 Kpkts 33.324 Mbps in 1001145 usec) 14.98 avg_batch
0 min_space
847.268327 sender_body [1293] drop copy
847.817915 main_thread [1720] 69.176 Kpps (69.261 Kpkts 33.245 Mbps in 1001229 usec) 14.98 avg_batch
99999 min_space
848.819100 main_thread [1720] 65.934 Kpps (66.012 Kpkts 31.686 Mbps in 1001183 usec) 14.99 avg_batch
99999 min_space

```

图 3-20 NetMap 部署测试用例

3.4 实现 Click 数据面

3.4.1 Click 体系结构

Click 是一个可以供使用者灵活配置的基于模块的软件路由器，它对外提供了一些可供调用的组件，按照一定的方式将这些组件组合可以形成不同的功能化模块。组件是 Click 最基础的对外服务的功能单元，它本身定义简单，只能执行如复制、分类、匹配、调度和排队等基础功能，更高级的网络功能需要将多个功能组件进行整合，才能实现更高级的功能服务。使用者在对基础组件经行组合时，需要按

照有向图的方式对基础组件进行连接，这个连接的顺序就是高级网络功能服务中数据流的流向。在实际的完成中，每一个基础组件都是由 C 语言编写的对象，通过指针配置和虚函数来对其进行连接操作，实现网络数据包在不同基础组件之间的流动。

在整个 Click 路由模块架构中，有很多类型的基础组件，而其中最重要的是 Click 行为组件。Click 行为组件是组成的高级网络功能中数据包具体动作操作的主要负责人，它对数据包执行诸如转发、内容修改和丢弃等行为动作。在具体的实现中，Click 的基础行为组件可以由 element 类对象、输入输出接口以及配置连接串三个部分组成。

其中，element 类对象都是由 C 语言实现的，在实现代码中配置了该 element 类对象中的相关字段和方法，比如对象的接口处理，组件的初始化方法以及针对不同数据的不同处理方式等等。输入输出接口是 Click 架构在设计时就需要配置好的提供给外部使用者的开放接口，在具体使用中，需要根据不同的需求来配置接口的数目，从一个 element 类对象的输出端口到另一个 element 类对象的输入端口被称作一个链接，每一个链接就是一种数据包通过的路径。而配置连接串则是一系列参数字段的集合，这个集合由用户自身设置，负责在 Click 启动初始化时向 element 类对象完成从用户到组件内部的参数传递，传递以列表的形式完成。

在传统的路由器模式中，如果需要针对数据包进行转发，路由器需要对整个数据包进行移动操作，当在网络功能虚拟化场景下，如果面对包长较大的数据包，传统的路由器会造成大量的性能损耗。Click 软件路由器针对这一情况重新定义了一种适用于自己内部的数据结构，首先当 Click 接收到数据包后会马上对这些数据包依据一定的标准进行分类，提取相同类别的包头并生成一种注释结构。这个注释结构的内容中只含有包头的相关信息和一个指向数据包中具体内容的行为指针，这样只需要传递这个容量大大减少的注释结构既可以完成数据包在 element 模块之间的传递，大大提高了数据包的转发性能。

此外，由于网络数据量的越来越大，如果系统内核频繁陷入到 CPU 中断中，那么这种中断方式将会对性能造成极大的额外开销，Click 针对这一情况进行了相关优化，采用了轮询的方式进行数据包到达处理。在这个过程中，Click 需要避开原有物理网卡中的原生驱动程序，调用自身的驱动程序，来完成有效的轮询。

3.4.2 Click 的部署以及调试

Click 当前已经提供了超过 200 个基础 element 组件，而要实现一个标准的路由器需要将十六个基础组件完成整合^[39]，连接方式如图 3-21 所示^[40]。

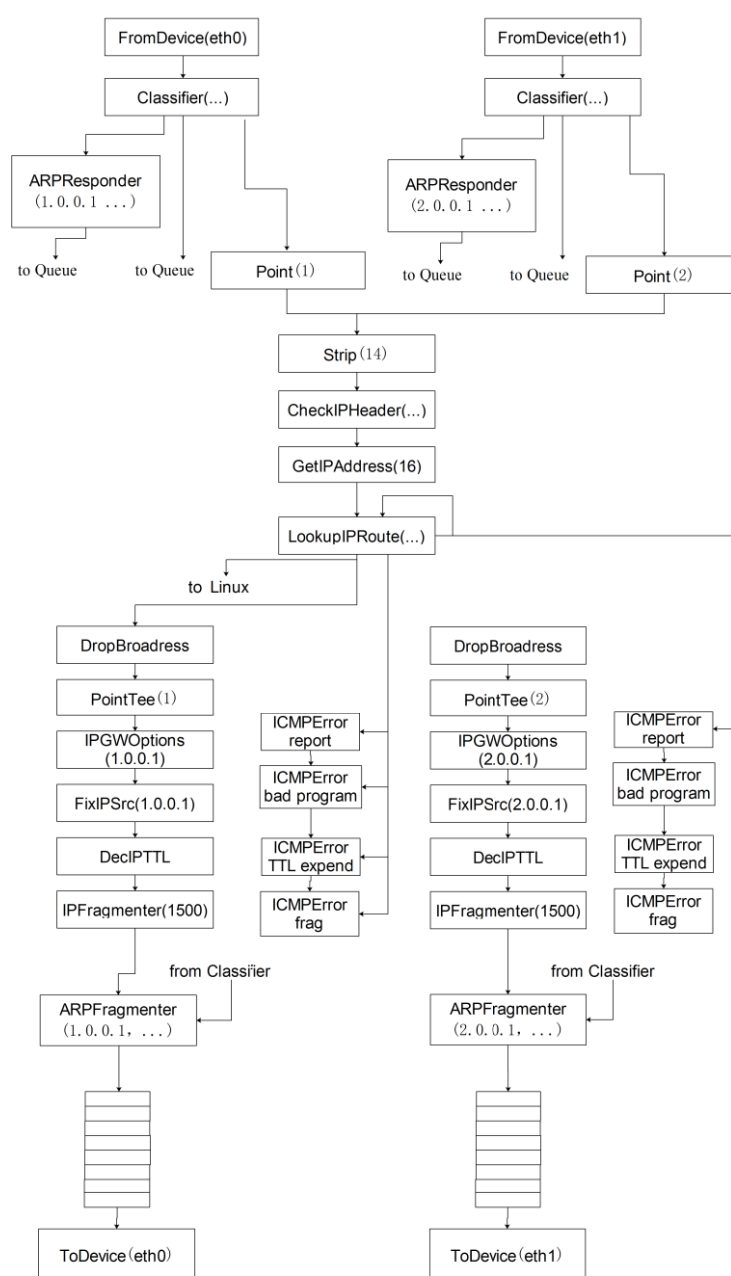


图 3-21 Click 标准路由器配置

CheckIPHeader 组件：用于对 IP 数据包的包头进行检查，如果是正确的数据报将其传递到下一个 element 组件；

Classifier 组件：用于对数据报进行分类处理并根据注释信息的不同将各类数据报从不同的输出接口完成输出；

LookUpIPRouter 组件：用于遍历静态路由表，并根据相关路由信息规则对数据包进行转发；

DropBroadcasts 组件：如果该组件判断某数据包的目的地址为广播地址，则该

模块负责将这个数据包进行丢弃方式处理；

ARPQuerier 组件：用于通过查看附加在数据包目标地址上的备注信息，通过 ARP 协议寻找到相应的以太网地址；

ICMPError 组件：根据 ICMP 协议对有异常情况的数据包进行相关的处理动作。

3.5 性能测试

3.5.1 测试环境介绍

为准确对比 DPDK-OVS、NetMap 和 Click 三种数据面的性能优劣，我们搭建了一个小规模测试平台，本文中所有的测试均在该测试平台上完成。

测试平台由三台 DELL R630 服务器和一台 Pica8 SDN 交换机组成，每台服务器配备 2 颗至强 E52600 处理器，CPU 频率为 1.7GHz，内存类型为 DDR4，内存大小为 8G，两块 Intel 千兆网卡，三台服务器均运行 64 位 Ubuntu 14.04 操作系统。连接方式如图 3-22 所示，两台服务器分别作为 NFV 流量的发送端和接收端，一台服务器部署 NFV 数据平面。Pica8 SDN 交换机作为外部的交换机连接三台服务器，流表中流表项由远程的主机控制下达。发送端服务器局域网内 IP 地址配置为 192.168.116.129，接收端服务器局域网内 IP 地址配置为 192.168.116.131。

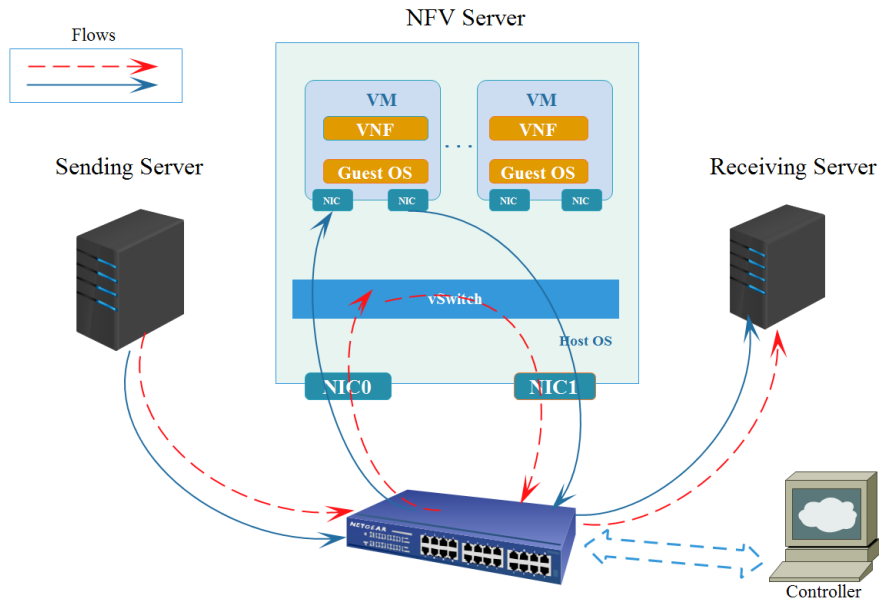


图 3-22 测试平台

为使 NFV 服务器中的两块物理网卡连通，我们将在 NFV 服务器上部署待测试的数据平面。在所有的性能测试中，我们将在 NFV 服务器上依次独立部署 DPDK-

OVS、NetMap Vale 和 Click 三种数据平面进行测试，所有的实验均独立运行 10 次取其平均值。

在测试中，我们考虑了两种场景：一种是在 NFV 服务器上仅仅部署数据平面，此时并未在宿主服务器上部署虚拟机，来对比三种数据平面的纯转发性能。网络流量如图 3-22 中虚线所示，通过 NFV 服务器的数据平面转发后即被转发至外部 SDN 交换机再转发至接收端服务器；另一种是在分别部署三种数据平面的基础上，分别部署虚拟 I/O 和虚拟机。由于软件配置问题，DPDK-OVS 上虚拟 I/O 可以选用 vhost-user 和 virtio-net 进行对比，NetMap 和 Click 只能选用 virtio-net。让网络流量通过虚拟网卡进入虚拟机中，再转发至数据平面最后转发至接收端服务器，网络流量如图 3-22 中实线所示。

在网络功能虚拟化场景下，发送端服务器上采用 tcpreplay 软件来发送网络流量^[40]，通过将真实环境中捕获的网络包保存在 pcap 文件中，重现保存的网络包，并可以指定其通过某一端口发送出去。tcpreplay 软件中的 ‘-m’ 和 ‘-p’ 选项可以控制网络流量的发送速率，通过这种方式我们可以控制 NFV 服务器上每秒处理的包的数量。而在接受端服务器上，我们采用 tcpdump 软件来捕获物理网卡上接受到的包。

为比较三种数据平面具体在网络功能虚拟化环境下的性能表现，我们将针对三种数据平面的网络吞吐量、网络延时、以及对不同大小的数据包的转发能力进行测试以及比较分析。

3.5.2 测试结果分析

为研究网络功能虚拟化平台中数据面性能对 NFV 性能的影响，我们在所搭建的测试平台上对 DPDK-OVS、NetMap 和 Click 三种数据平面进行了对比测试。

首先我们测试了在纯数据平面即不部署虚拟机的条件下，三种数据平面各自的网络吞吐量性能表现。在测试环境中，网络包从发送端服务器上发送，经过 SDN 交换机进入 NFV 服务器，通过数据面的转发，再经由 SDN 交换机到达接收端服务器。

我们使用 iperf 网络性能测试工具来对吞吐量进行测试。其测试效果如图 3-23 所示，我们在发送端服务器服务器以 500Mbps/s 的速率发送数据包，可以通过 iperf 工具观察到此时带宽为 436Mbit/s 到 449Mbit/s 之间浮动。

借助以上工具，我们首先对传统 OpenvSwitch 的性能和进行了与 DPDK 整合后的 DPDK-OVS 转发平面的性能进行了对比。首先比较了两种转发平面的网络吞吐性能，我们分别部署了这两种数据平面，在发送端服务器发送网络包的速率从

100Mbps/s 逐步增加到 1000Mbps/s, 对数据平面的转发性能做出对比测试。

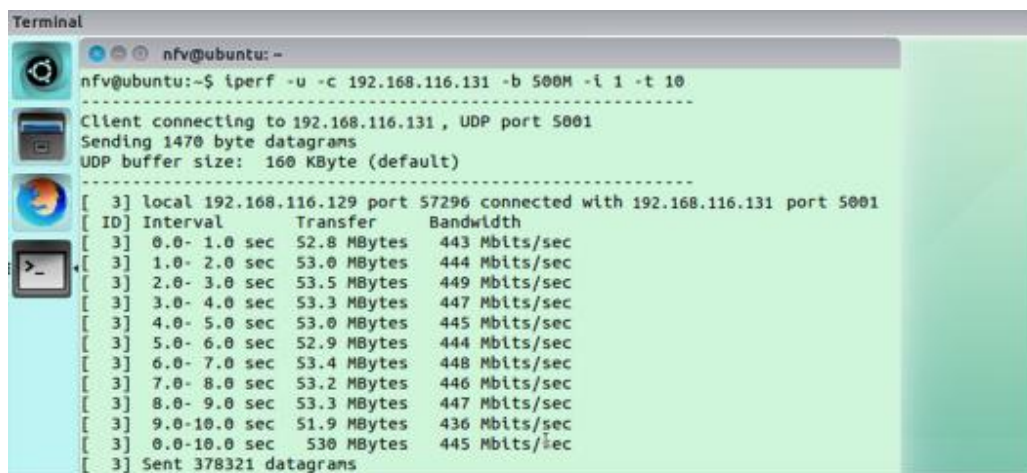


图 3-23 测试带宽

首先针对单独 OpenvSwitch 转发平面和结合 DPDK 后的 OpenvSwitch 以及进行优化后的 DPDK-OVS 进行转发性能测试, 测试结果如图 3-24 所示。可以看到在传统 OpenvSwitch 的工作模式下, 带宽利用率很低, 网络包转发能力远远不如 DPDK-OVS, 这主要是受制于数据包在内核空间的转发会严重影响到数据平面的转发, 使用 DPDK 工具针对这一影响瓶颈进行加速, 可以接近网卡的线速转发, 大大提高了原有的数据转发平面的转发性能。在经过我们优化方案优化后, DPDK-OVS 的性能有了进一步提升, 经过绑定 CPU 核优化以及硬件配置优化后, DPDK-OVS 的吞吐量上升了 15%左右, 优化效果显著。

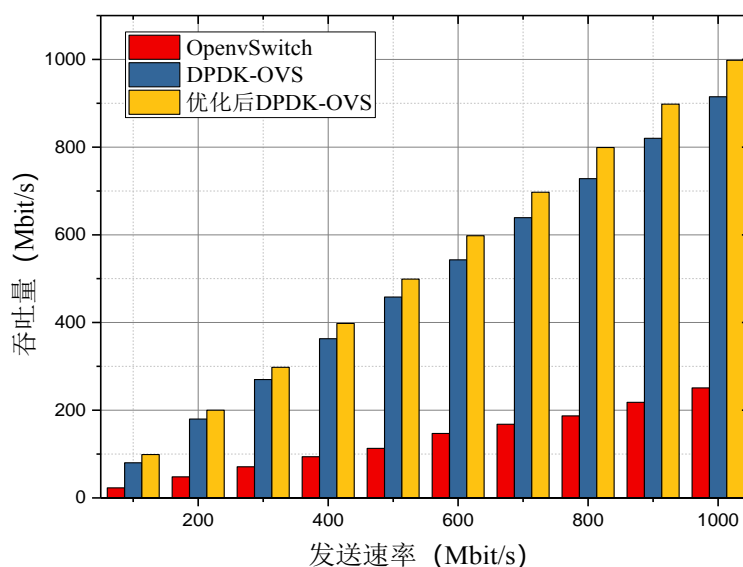


图 3-24 OpenvSwitch 优化前后吞吐性能

之后我们对优化前后的 NetMap 数据平面的吞吐性能进行了测试，其测试结果如图 3-25 所示。由于我们改进了 netmap_ring 数据结构，减少了 NetMap 在进行数据包转发过程中内核态和用户态切换的次数，其吞吐性能得到了 10%左右的提升，优化效果显著。

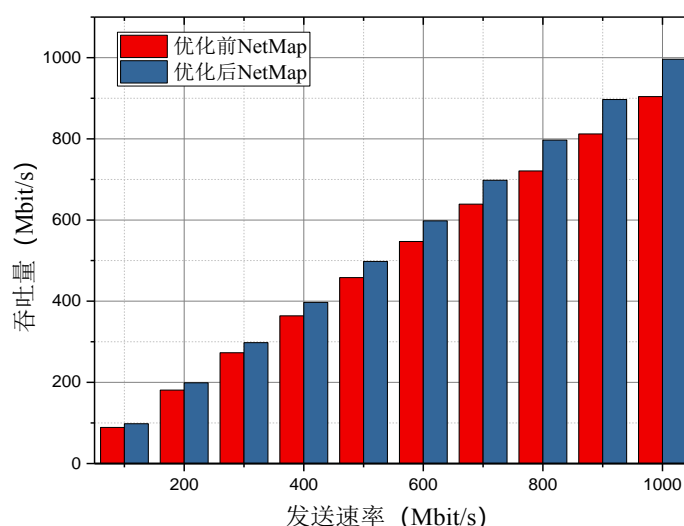


图 3-25 NetMap 优化前后吞吐性能

在针对优化后的 DPDK-OVS、NetMap 与 Click 三种数据转发平面的对比中，我们也按照上述方法，将发送端服务器发送网络包的速率从 100Mbits/s 逐步增加到 1000Mbits/s，测试结果如图 3-26 所示，可以看到在三种数据平面的吞吐量基本与带宽一致，说明基本可以达到线速转发，且三种数据平面之中，Click 略差于优化后的 DPDK-OVS 和 NetMap，三种数据平面均不会造成性能瓶颈障碍。

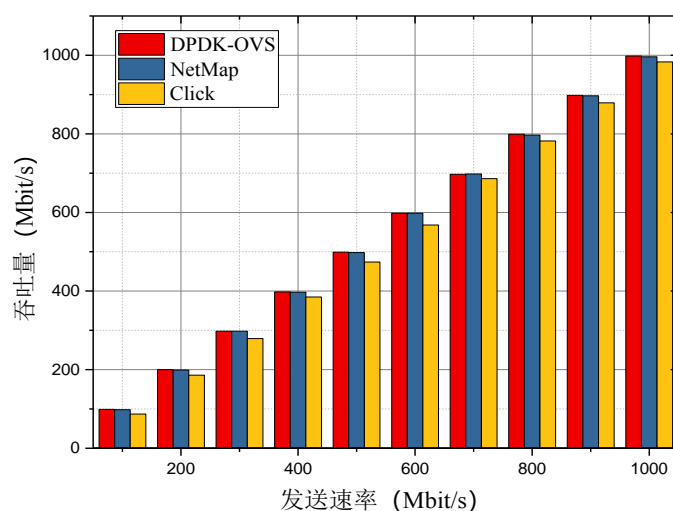


图 3-26 纯数据平面吞吐性能

使用 ping 命令来测量网络时延。如图 3-27 所示，初始状态下，DPDK-OVS、NetMap 和 Click 时延较大，而经过优化后，DPDK-OVS 和 NetMap 的网络时延均大幅下降，优化效果显著，说明网络非常稳定，没有发生网络拥塞。

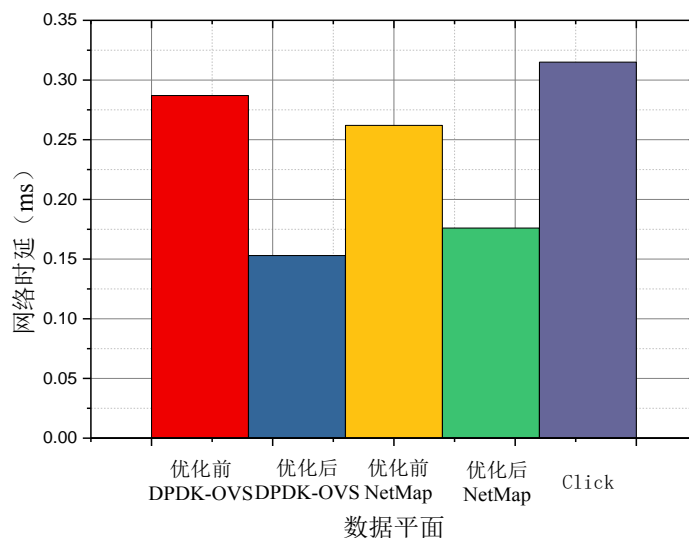


图 3-27 网络时延对比

在比较三种数据平面对不同大小的数据包的转发能力上，我们通过在发送端服务器上分别发送大小不同的数据包到网络中，经由 NFV 服务器处理后转发至接收端服务器。发送端服务器不断加大发送端速度，直到整个网络出现拥塞，在接收端服务器和 NFV 服务器上查看网卡收到的包的个数以及经数据平面处理过的包的个数。再利用整个测试中包处理的时间以及处理过的数据包的总数即可得到每个数据平面每秒可以处理的数据包的个数（packet per second, PPS）。

我们首先比较了 OpenvSwitch 和 DPDK-OVS 以及优化后 DPDK-OVS 的转发性能，测试结果如图 3-28 所示。

可以看到在经过了 DPDK 工具的加速后，OpenvSwitch 的转发能力得到了大幅加持，最大可以达到原先情况下五到六倍的最大转发能力，而经过 CPU 核规划优化后，DPDK-OVS 的包处理能力再次得到约 15% 左右提升。

之后比较了 NetMap 数据交换平面中，初始状态下和我们进行数据结构优化后的包转发能力，测试如图 3-29 所示。

可以看到在未经过数据结构优化时，用户空间进程和操作系统内核在对 netmap_ring 进行读取数据和写入数据时，要分别对其中的数据结构变量字段进行修改，同时还要保持其他线程不会修改这两个字段，当数据量非常大的时候，会造成很大的额外性能开销。在我们针对 netmap_ring 的数据结构做出优化后，可以看

到针对数据包的转发能力，优化后 NetMap 转发数据包的 pps 要比未进行优化时提高了 10%左右，其中当处理小包时优化效果要优于处理大包的优化效果。

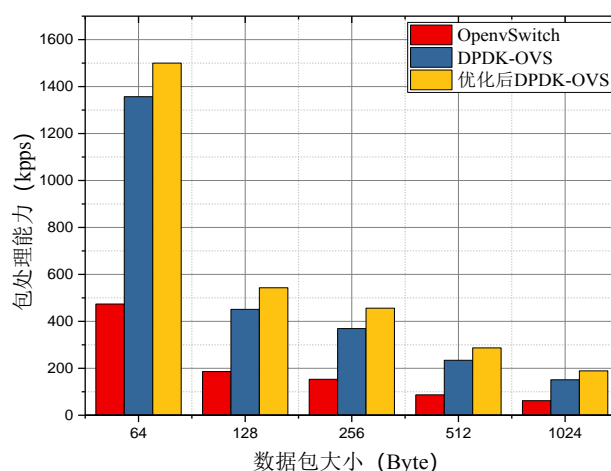


图 3-28 OpenvSwitch 整合优化前后最大转发能力对比

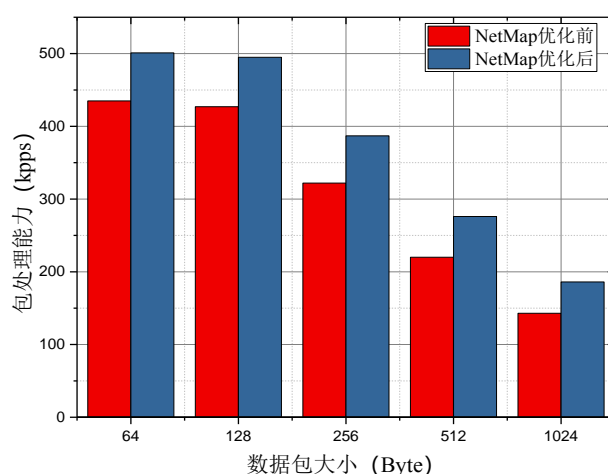


图 3-29 NetMap 优化前后最大转发能力对比

而针对服务器上所部署的三种数据转发平面，进行包最大处理能力的测试结果如图 3-30 所示，可以看到当数据包规格较小的时候，DPDK-OVS 的处理能力强于 NetMap 和 Click，这时后二者处理数据包时容易造成网络拥塞，当数据包较小时，应多选用 DPDK-OVS 数据面来部署。当数据包在 512 字节以上时，可以看到三种数据平面的处理能力基本相同，这说明三者的转发性能接近了网卡的极限速度，此时，包处理能力的性能瓶颈更应该在于物理网卡的 I/O 能力上而不是中央处理器的包处理能力上。

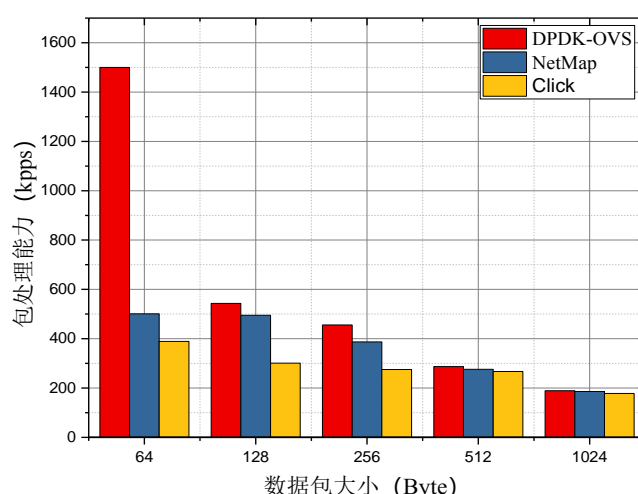


图 3-30 纯数据平面最大处理能力

当在 NFV 服务器上部署虚拟 I/O 和虚拟机后，再对三种数据平面的最大处理能力进行对比，测试结果如图 3-31 所示。

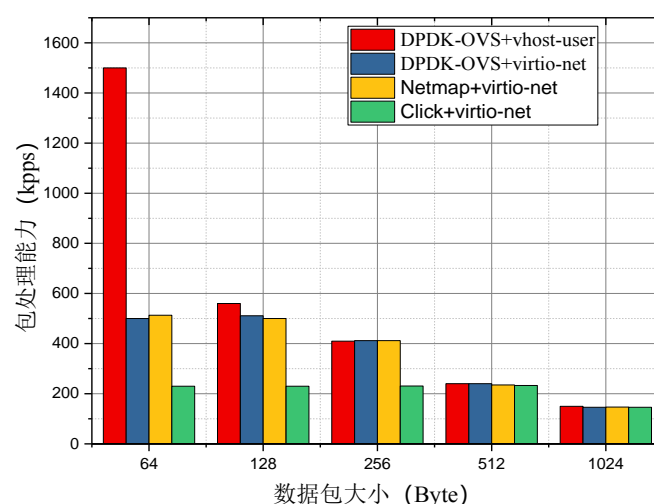


图 3-31 增加虚拟 I/O 后数据平面最大处理能力

可以看到，DPDK-OVS 在分别部署 vhost-user 虚拟 I/O 和 virtio-net 虚拟 I/O 时，在处理 64 字节小包时性能差异显著，且 DPDK-OVS+vhost-user 效果远好于 DPDK-OVS+virtio-net，说明此时性能瓶颈不在于数据平面，而在于虚拟 I/O 的性能上。

对比图 3-30 和 3-31 可以看到 NetMap 性能表现差异不大，说明部署虚拟 I/O 对其性能表现影响较小，而 Click 在部署了虚拟 I/O 后，当面对 64 字节、128 字节和 256 字节的包时，Click 的处理能力差于未部署虚拟 I/O 时，也差于 DPDK-OVS 和 NetMap，说明部署虚拟 I/O 增加了 Click 的性能开销，对其转发性能影响较大。

当处理 512 字节以上的大包时，三种数据平面的处理能力基本一致，说明此时虚拟 I/O 未影响到数据平面的处理能力，此时的性能瓶颈是物理网卡的 I/O 能力，而不在于数据平面或者虚拟 I/O 上。

从吞吐量上看，三种数据平面均接近系统限速转发，不会造成性能瓶颈。三种数据平面对数据包的转发上时延较小，可以保证较为稳定的网络传输。在不同大小的数据包的转发能力上，DPDK-OVS 的性能要优于 NetMap 和 Click，尤其是当考虑到配合上层虚拟机，增加虚拟 I/O 后，使用 vhost-uesr 配合 DPDK-OVS，性能提升更加显著。

但具体到工程平面的部署上时，DPDK-OVS 所存在的问题要大于 NetMap 和 Click。DPDK-OVS 数据面组合由于来自两个设计初衷不同的组件，其相关整合会影响到各自一些功能的实现，各自开源组件尚存在一些 bug 尚未修复，由于 DPDK 使用了大页技术，当宿主机的内存使用率过高时，会导致连接到 OpenvSwitch 上的虚拟机收发数据包延时大幅上升。同时系统编译器的不同也会对两个组件的整合产生一定的影响。所以在具体的使用中需要结合具体需求和我们的测试结果综合考虑，如何选择合适的的数据转发平面。此外，OpenvSwitch 由于设计原因，当面临大面积组网的时候，其转发性能表现也会产生较大波动，使用情况不如 NetMap 和 Click 稳定。

综合以上所有的测试结果考虑，DPDK-OVS 适用于使用虚拟机时的场景，尤其是使用了 vhost-user 虚拟 I/O，可以提供非常优秀的转发性能。NetMap 适用于仅宿主机中，可以应用在 Docker 容器的网络连接中，可以提供稳定的性能表现。Click 的性能表现差于前两个数据转发平面，但是由于前两个数据转发平面都需要一定的内存资源，故而在内存资源紧张时可以优先选用 Click 作为数据转发平面。

3.6 本章小结

在本章中，我们针对网络功能虚拟化实际部署中的需求，对不同数据转发平面进行了性能上的对比，选取了 DPDK-OVS、NetMap 和 Click 三种业内应用广泛的数据平面来进行研究，并对 DPDK-OVS 和 NetMap 进行了优化改进。我们搭建了由三台服务器组成的测试环境，分别对三者进行部署和性能测试，根据对三种数据平面在纯数据面使用中和部署虚拟机以及虚拟 I/O 的情况下的网络吞吐量、网络传输时延以及对不同大小的数据包的转发能力进行测试以及分析，结合具体部署过程中三中数据平面存在的问题，得出在实际使用中的结论。

第四章 面向网络功能虚拟化的虚拟方式对比

虚拟化技术作为当今最主流的几个 IT 技术之一，在近些年来得到了快速的发展。在传统的服务部署中，如果仅仅使用一台服务器进行服务的安装和运行，由于潮汐效应，可能会导致物理资源在某一时刻十分紧张而下一时刻又需求锐减，这将导致资源的极大浪费。虚拟化技术可以在原有宿主机的基础上，将计算、存储、网络等资源虚拟化成多个执行单元分别对外提供服务，极大地提高了资源利用率，这在网络功能虚拟化的实际部署中更加重要。KVM 虚拟化技术和 Docker 容器技术作为当下发展最快的两种虚拟化技术，其在各自的技术架构决定了其性能和使用场景的差异。在网络功能虚拟化场景下如何选择合适的虚拟化方式，需要对二者的实现方式和性能表现进行分析。

4.1 虚拟机技术与容器技术的对比

虚拟化将物理服务器所能提供的实体物理资源抽象成独立的虚拟资源，从而为不同需求的用户提供服务，同时保证了服务的隔离性和安全性。KVM (Kernel-based Virtual Machine) 作为完全虚拟化技术的代表和 Docker 容器技术作为操作系统级虚拟化的代表，分别从不同层面实现了虚拟化。其架构模型对比如图 4-1 所示。

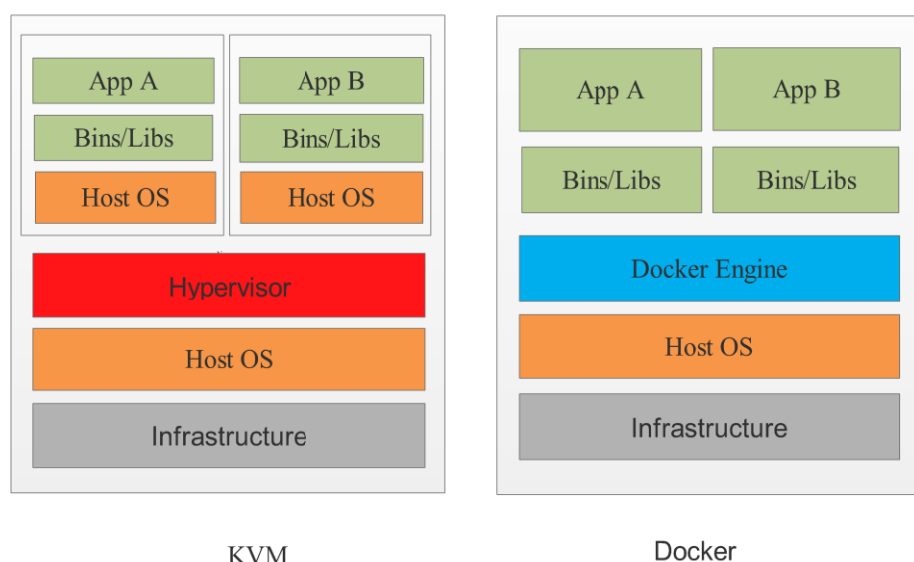


图 4-1 KVM 架构与 Docker 架构

完全虚拟化技术又叫硬件辅助虚拟化技术，它在上层客户虚拟机和底层宿主

机硬件之间加了一个由软件来实现的管理层 Hypervisor，即虚拟机监控器。宿主机服务器中的底层物理硬件需要 Hypervisor 来向上抽象服务，对虚拟机提供抽象的计算、存储和网络资源，而运行在虚拟机之中的客户机操作系统在运行中下达的指令，需要经由 Hypervisor 来捕获并处理成底层操作系统实际执行的 CPU 指令。在这个底层操作系统和客户机操作系统之间的交互中，由于 Hypervisor 的加入会导致一定程度上性能的损耗，所以单个虚拟机操作系统在相同条件下其运行性能并不如裸机。

操作系统级虚拟化相对完全虚拟化技术而言，是一种更加轻量级的虚拟化技术，其中以 Docker 容器技术作为代表。与传统的全虚拟化方式相比，Docker 容器技术并不需要 Hypervisor 在宿主机操作系统和客户机操作系统之间帮助交互，不需要采用全虚拟化中模拟 CPU 操作的指令集，容器可以和宿主机共同使用一个内核。容器技术是从 Linux 系统内核入手，利用 Linux 中的 Namespace 技术和 Cgroups 技术等操作系统原生，为使用者提供了相对于完全虚拟化而言更加轻量级的虚拟化技术。容器技术大大减小了部署时的额外开销，可以直接和宿主机的底层操作系统共享内核，在性能上更加贴近于宿主机本身，这样一来即可以更加充分地调用底层物理系统资源。

在网络功能虚拟化部署的应用上，KVM 虚拟机和 Docker 容器均可以作为 VNF 部署的承载体，在各自的环境中独立运行其网络功能，而由于其两种虚拟化的思路不同，其可以提供的性能表现也是不同的。

4.2 基于容器的网络功能虚拟化

4.2.1 Docker 容器架构

Docker 容器技术的实现依赖于 Linux Container 的一些核心技术，通过 Cgroups 和 Namespace 等技术实现资源和环境的隔离。其组件架构图如图 4-2 所示。

Docker 容器主要由五个重要组件构成：Docker 客户端、Docker 守护进程、Docker 底层驱动程序、Docker 镜像文件和镜像仓库。

Docker 客户端用于向 Docker 守护进程发送容器操作请求。

Docker 守护进程作为 Docker 容器功能中的核心部分，可以接受来自上层 Docker 客户端的请求，然后交由 Docker 实际的引擎 Engine 来进行处理，根据请求内容的不同，可以以此开放不同类别的处理器进行处理。

底层驱动程序按照实际功能类型区分可以分为执行驱动、网络驱动和镜像驱动。其将上层传递来的格式化命令转译为统一的操作系统级函数调用，从而由底层处理器完成对容器的创建和管理。

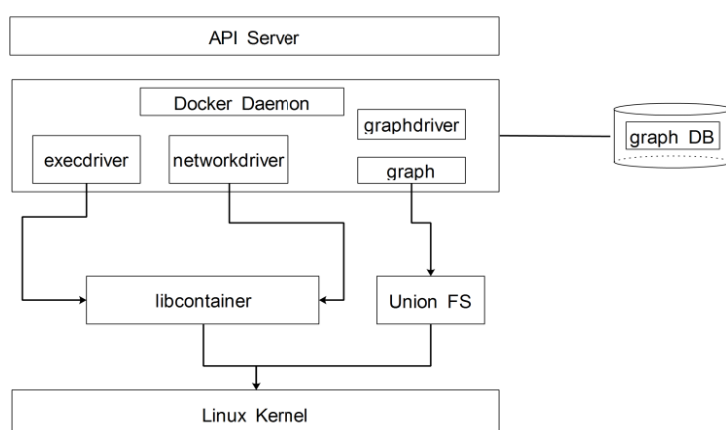


图 4-2 Docker 组件架构图

Docker 镜像是一个只读文件，它是 Docker 如何运行的主要参考，每一个镜像都在不同的时间由各种配置层相互叠加而成。如果有新的需求到达，需要配置和修改相应的文件，则在原有的 Docker 镜像文件上进行操作配置层的叠加即可生成新的镜像文件。

Docker 仓库是用来保存镜像文件的代码仓库，可以分为公有仓库 Docker Hub 和各类私有仓库。

4.2.2 部署 Docker 容器及虚拟网络功能

在虚拟网络功能的部署上，我们选择 Nginx 这一业内常用的负载均衡软件作为 VNF 来部署在 Docker 容器中。

第一步，在根目录下需要创建 Nginx 目录，用来存放相关配置文件；

第二步，进入创建好的 Nginx 子目录，手动编写新的 Dockerfile 文件，其配置文件效果如图 4-3 所示；

```

nfv@ubuntu: ~/nginx
FROM Ubuntu:14.04

MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"

ENV NGINX_VERSION 1.10.1-1-jessie

RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys 573BFD6B3D8FBC641079A6ABAF5BD827BD9BF62 \
    && echo "deb http://nginx.org/packages/debian/ jessie nginx" >> /etc/apt/sources.list \
    && apt-get update \
    && apt-get install --no-install-recommends --no-install-suggests -y \
        ca-certificates \
        nginx=${NGINX_VERSION} \
        nginx-module-xslt \
        nginx-module-geoip \
        nginx-module-image-filter \
        nginx-module-perl \
        nginx-module-njs \
        gettext-base \
    && rm -rf /var/lib/apt/lists/*

# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log \
    && ln -sf /dev/stderr /var/log/nginx/error.log

EXPOSE 80 443

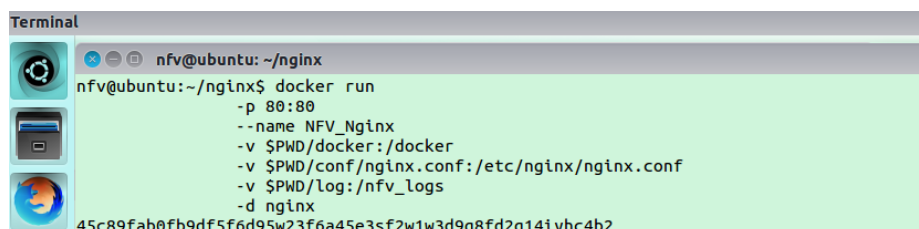
CMD ["nginx", "-g", "daemon off;"]

```

图 4-3 创建新的 Nginx Dockerfile

第三步，通过已经创建好的 Dockerfile 来创建一个镜像文件；

第四步，利用已有的包含 Nginx 功能的镜像文件创建容器，其创建方式如图 4-4 所示。

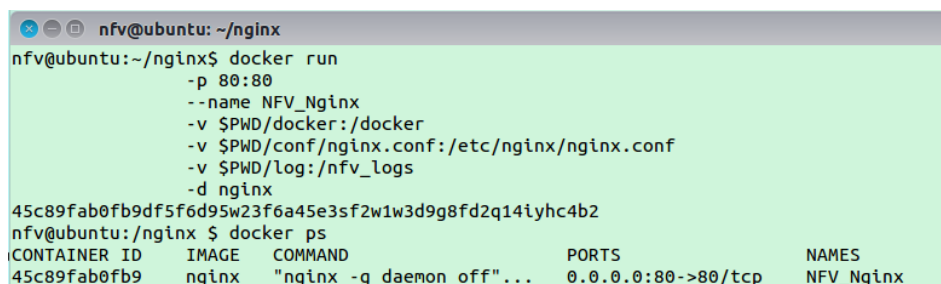


```
Terminal
nfv@ubuntu: ~/nginx
nfv@ubuntu:~/nginx$ docker run
-p 80:80
--name NFV_Nginx
-v $PWD/docker:/docker
-v $PWD/conf/nginx.conf:/etc/nginx/nginx.conf
-v $PWD/log:/nfv_logs
-d nginx
45c89fab0fb9df5f6d95w23f6a45e3sf2w1w3d9g8fd2q14iyhc4b2
```

图 4-4 启动 Docker

其中，-p 80:80，在物理宿主机的 80 端口和新创建的 Docker 容器的 80 端口之间构建映射，--name NFV_Nginx：将新创建的容器命名为 NFV_Nginx，-v \$PWD/docker:/docker：将主机中当前目录下的 docker 目录挂载到新创建的 NFV_Nginx 容器下的 /docker 文件目录下，-v \$PWD/conf/nginx.conf:/etc/nginx/nginx.conf：将主机中当前目录下的 nginx.conf 挂载到容器的/etc/nginx/nginx.conf，-v \$PWD/log:/nfv_logs：将主机中当前目录下的 log 挂载到容器的/nfv_logs 目录下。

至此，一个运行着 Nginx 负载均衡器的 Docker 容器即已创建完成。创建完成后运行启动命令成功，运行相关命令查看可以看到运行着 Nginx 网络功能的容器进程已经开始启动，启动结果如图 4-5 所示。



```
nfv@ubuntu: ~/nginx
nfv@ubuntu:~/nginx$ docker run
-p 80:80
--name NFV_Nginx
-v $PWD/docker:/docker
-v $PWD/conf/nginx.conf:/etc/nginx/nginx.conf
-v $PWD/log:/nfv_logs
-d nginx
45c89fab0fb9df5f6d95w23f6a45e3sf2w1w3d9g8fd2q14iyhc4b2
nfv@ubuntu:~/nginx $ docker ps
CONTAINER ID    IMAGE    COMMAND                  PORTS          NAMES
45c89fab0fb9    nginx    "nginx -g daemon off"...  0.0.0.0:80->80/tcp  NFV_Nginx
```

图 4-5 查看 Docker 启动效果

4.2.3 优化容器网络连接

当在一台物理服务器上部署多个 Docker 容器构建服务功能链时，需要合理规划容器间的网络连接，才能使 Docker 容器的性能得到最大化的利用。

Docker 容器默认的网络通信一般分为五种模式，分别为 bridge 模式、host 模式、none 模式、其他容器模式以及用户自定义模式。其中 bridge 模式是为容器新建立一个独立的 Namespace，该容器具有独立的网卡以及独立的网络协议栈，通

过与宿主机端口绑定来实现与宿主机通信，通过系统下自带的 Linux bridge 实现容器间的通信；host 模式是直接使用物理宿主机的 Namespace，这种模式下宿主机的 IP 地址即为容器的 IP 地址，容器可以将网卡挂载到宿主机的物理网卡上；none 模式可以为容器创建一个新的独立 Namespace，但是并不做任何配置，容器只有 IO 进程，无法与宿主机或者其他容器进行网络通信；其他容器模式可以是多个容器共享一个 Namespace，这些具有共享 Namespace 的 Docker 容器之间不具有网络隔离性，但他们与宿主机之间网络仍然具有隔离性；用户定义网络模式可以使用任何 Docker 支持的第三方网络驱动来对容器的网络进行定制。

部署于 Docker 容器中的虚拟网络功能有可能是通过单独的应用对外提供服务的，也可能是与部署于其他容器中的虚拟网络功能连接，构建服务功能链来共同对外提供服务。当 Docker 部署在运行着 Linux 操作系统的通用服务器上，其默认的网络连接方式是通过 Linux bridge。传统创建容器网络的时候，当一台 NFV 服务器需要部署多个 Docker 容器来构造服务功能链时，最简单的网络架构连接方式如图 4-6 所示，可以使用一个网桥将所有的容器连接在一起，容器之间通过这个网桥进行通信。

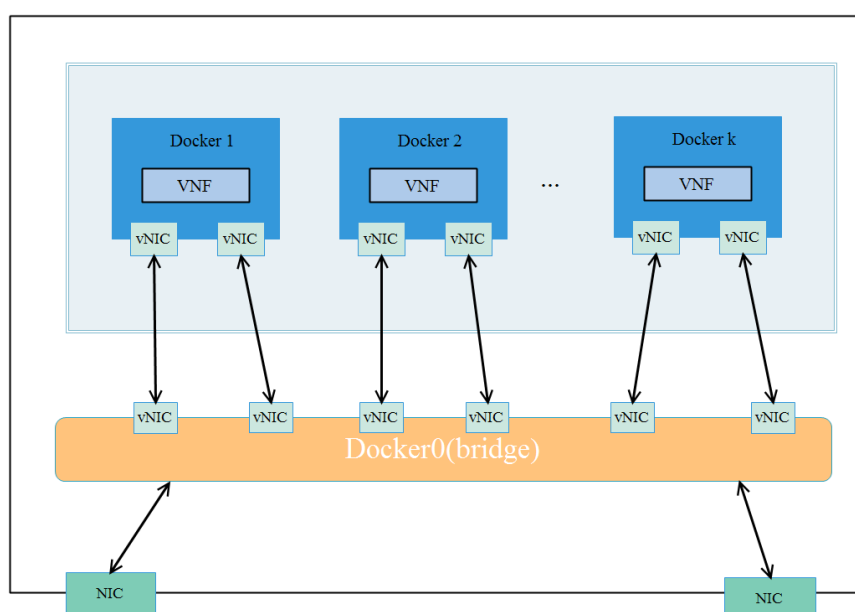


图 4-6 使用一个 Linux bridge 来构建网络

由第三章可知，当需要容器或者虚拟机运行在网络功能虚拟化场景下时，需要满足 NFV 数据包高速的需求。而 Linux bridge 在接收到来自物理网卡的数据包后，需要向 Docker 容器进行数据包的转发时，会涉及到大量的用户态内核态上下文切换和数据包的拷贝，这造成了极大的性能损耗。根据第三章的对比结果，我们可以看到 DPDK-OVS 和 NetMap 两种软件交换机效果较好，而 DPDK-OVS 构建网络

时，使用虚拟 I/O 接口针对虚拟机而言更加合适，所以我们选用 NetMap VALE 软件交换机来实现数据平面，构建 Docker 容器网络。

同时，在每个 Docker 容器之间，如果仅仅是采用一个 Linux bridge 来进行数据包的转发，那么如果根据上层需要，部署在宿主机内的容器数目需要进行增加，就需要对进入到用户空间，对这个 Linux bridge 进行端口的增加和对 MAC 地址-端口映射表的修改。当映射表内容过多时，会造成查询访问效率下降，从而导致转发性能下降。为此，我们针对这一情况提出了改进方案。为了提高 Docker 容器间数据包的转发性能，可以在服务器允许的情况下，在每两个 Docker 容器之间创建一个 Linux bridge，这个 Linux bridge 只负责二者之间的数据包转发。当需要新建容器时，在与其向邻的 Docker 容器之间再新建一个 Linux bridge 并设定相关转发规则即可。

因此，我们在构建宿主机内部的 Docker 容器网络时，采用了如图 4-7 所示的方法。

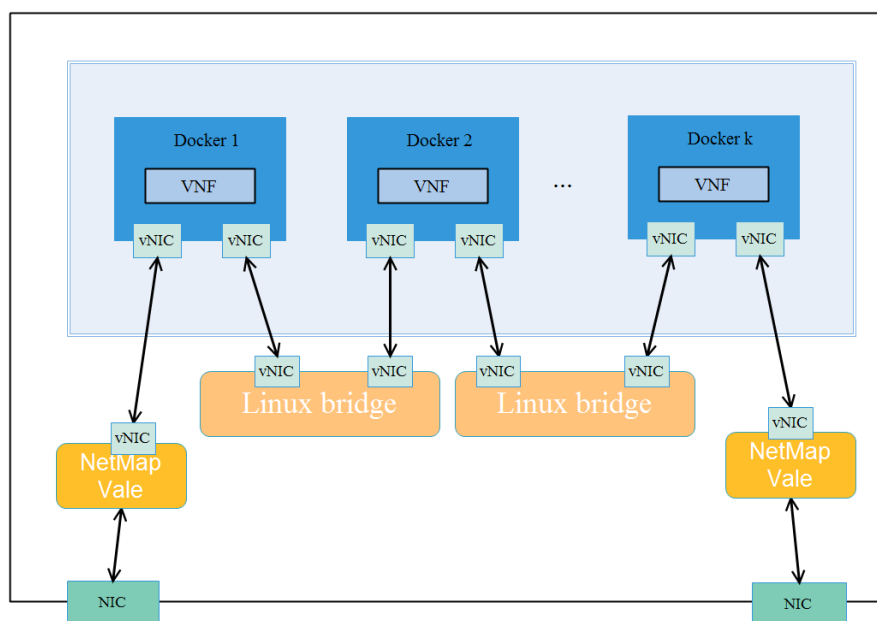


图 4-7 改进后的容器网络

优化后的容器网络设计可以有效地提高了具有通信关系的容器两两之间的数据包拷贝效率问题，解决了因为服务功能链过长带来的查询端口效率低下带来的性能问题。但该方案仍然具有瓶颈，原因在于容器间的通信关系并非固定，由于容器具有简单易部署的优点，其新建容器和删除容器动作十分频繁，当单服务器内网络规模较大时，这给网桥管理带来了极大的不便。

为有效改善这以瓶颈，我们提出使用共享内存的字符设备来实现容器间网络

数据包转发，这样既解决了多网桥的管理复杂问题，也避免了在网桥中数据包的频繁拷贝问题。

我们在每一条服务功能链上使用一个共享内存的字符设备来作为一个缓冲池，当服务功能链上的第一个虚拟网络功能从 NetMap 数据面接收到网络数据包后，将这个数据包写入整个功能链共用的缓冲池中，后续的容器直接使用接收到的缓冲池地址即可对查询到数据包并对其进行操作。容器间通信采用进程间通信方法，使用环境队列传递消息，通过传递包缓冲区的偏移地址来进行数据包的传递。优化后的容器网络设计如图 4-8 所示。

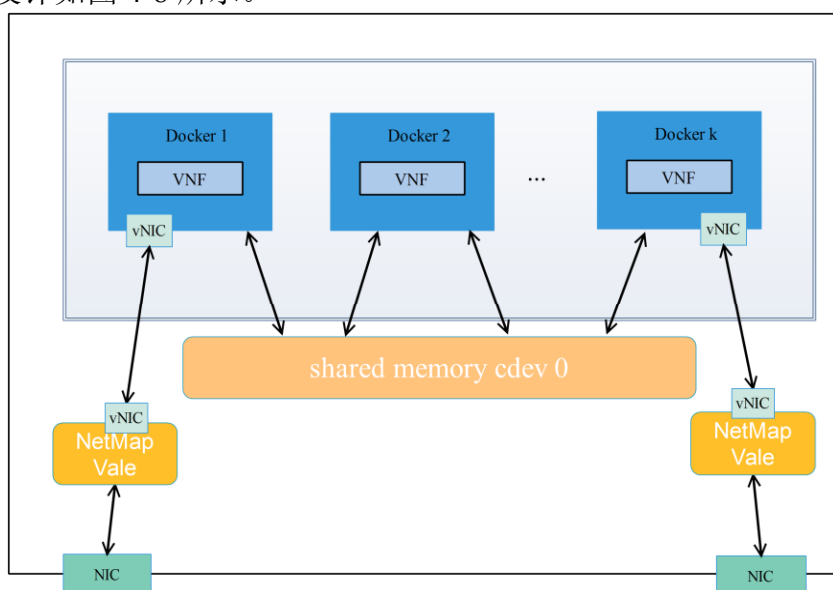


图 4-8 使用共享内存的容器网络

实现基于共享内存的容器间通信，首先要实现一个字符设备，再申请一块内存区域，并将其视作一个字符设备以供数据包保存。在实现过程中，关键的步骤是实现 `mmap()` 函数，使字符设备中的内存地址可以和容器进程中的虚拟地址互相映射。其实现关键步骤如图 4-9 所示。

```

1 static int chardev_mmap()(struct file *filp, struct vm_area_struct *vma)
2 {
3     unsigned long size = vma -> vm_end - vma -> vm_start;
4
5     /*获取字符设备内存的页帧号*/
6     unsigned long pfn = __pa(dev_memaddr)>>PAGE_SHIFT;
7     ...
8     if(remap_pfn_range(vma, vma -> vm_start,
9                        pfn, //设备物理地址节帧号
10                       size, //映射内存大小, 字节为单位
11                       vma -> vm_page_prot)){
12         return -EAGAIN;
13     }
14     vma -> vm_ops = &vm_ops;
15     vma -> vm_flags |= (VM_IO | (VM_DONTEXPAND | VM_DONETDUMP));
16     ...
17     return 0;
18 }

```

图 4-9 字符设备实现 `mmap` 函数主要过程

至此，基于 Docker 容器的网络功能虚拟化平台即已搭建完毕。

4.3 基于 KVM 的网络功能虚拟化

4.3.1 KVM 部署架构

KVM 虚拟化技术常常用来部署在 X86 通用服务器上，被应用在 Linux 操作系统环境下将物理的宿主机拥有的实体物理资源进行抽象，向应用层用户提供多个虚拟的隔离环境来满足需求。在 KVM 的实际使用中，单个客户虚拟机表现为运行在宿主机中的常规 Linux 进程，统一由 Linux 操作系统内核中的进程调度程序来负责管理和维护。而在客户虚拟机中的每个虚拟 CPU 也被虚拟化为一个常规的 Linux 进程，通过 QEMU 工具来与宿主机操作系统中的指令进行转换，完成操作指令的下达。其架构如图 4-10 所示。

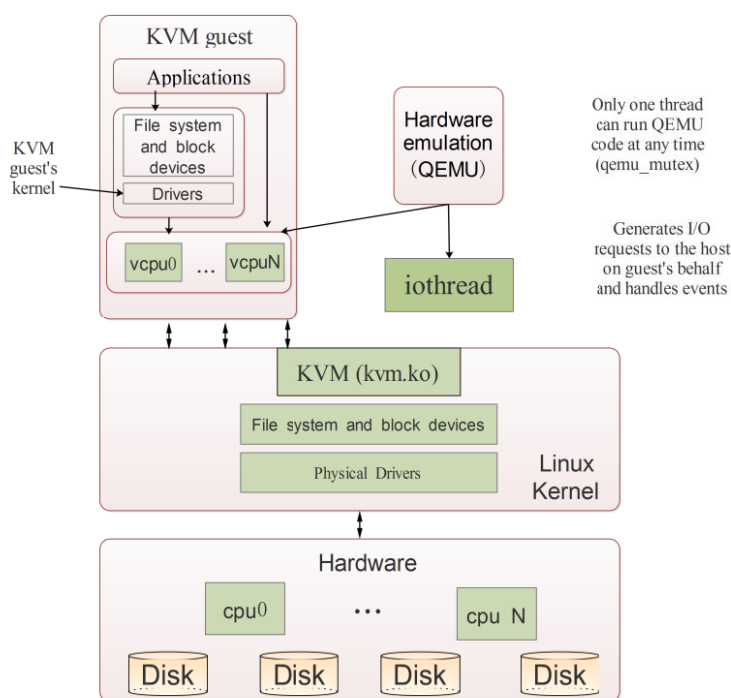


图 4-10 KVM 组件架构图

在 KVM 体系架构中，KVM guest 是客户机系统，可以部署独立与宿主机操作系统的操作系统，而底层的相关硬件资源如 CPU、内存、虚拟网卡以及其他设备驱动的资源，也会通过相应的技术手段完成从物理到虚拟的抽象，以提供给顶层的虚拟机来使用。虚拟机中部署的操作系统以及应用在底层宿主机操作系统中表现为 KVM 进程中的一个线程，完成指令间的转换和资源的隔离。

具体的物理资源向虚拟资源的抽象过程是由 KVM 内核模块来完成的，他负责借助相应物理设备的支持，将计算、存储、网络等资源向上抽象，交由客户机中的

模块完成使用。同时 KVM 内核模块负责拦截来自客户虚拟机的 I/O 请求，交给 QEMU 模块完成指令转换。

QEMU 在开发设计之初并不是 KVM 原生的模块，它自身也可以实现一些虚拟化的功能。KVM 借助 QEMU 模块的帮助完成从客户虚拟机中的请求到宿主机中相关操作的调用，QEMU 在这个过程中将来自虚拟机的 I/O 请求解析，并提供设备的模拟化功能。故而从另一个角度来看，也可以认为 QEMU 使用了 KVM 的虚拟化组件来对自身的虚拟化功能进行性能加速处理。

4.3.2 创建 KVM 虚拟机

为部署虚拟网络功能，首先需要使用 KVM 虚拟化方式来创建虚拟机。其中创建虚拟机有以下几种方法：

第一种是使用 virt-manager 工具。如果是图形化的 Linux 操作系统，可以选择使用 virt-manager 软件工具来进行 KVM 虚拟机的创建，使用 virt-manager 工具可以在图形化的界面下对需要创建的虚拟机进行规格上的设定，可以指定虚拟机的操作系统，在物理资源上可以指定其 CPU、内存、硬盘等资源大小，也可以通过其设置网卡连接方式，合理配置虚拟机的网络。

第二种是使用 QEMU-IMG 和 QEMU-KVM 命令行的方式进行安装，这种方式首先要创建一个 qcow2 格式的镜像文件，再通过命令行启动一个虚拟机，在启动的时候可以不同的命令行参数规定其物理资源如 CPU、内存、硬盘等资源的规格。将其挂在到 cdrom 上，安装需要的操作系统，最后在镜像文件的基础上启动虚拟机即可。

第三种是使用 OpenStack 工具。OpenStack 作为一款 IaaS 平台，其可以方便快捷地创建虚拟机，并且其内部实现原理也是使用 KVM 的内核，所以在 OpenStack 工具内部，可以使用相关的 libvirt API，通过编程的方式创建虚拟机，也可以通过设置相应的模板，实现按照客户的编排来启动所需要的特定规格的虚拟机。

在创建好虚拟机后，通过 ssh 工具访问到虚拟机，即可完成虚拟网络功能的软件安装和配置，也可以制作包含相关网络功能软件的镜像文件，构建镜像仓库，以便后续使用。

如图 4-11 所示，我们在宿主机服务器上使用 virt-manager 工具来对 KVM 虚拟机进行创建，可以看到使用该图形化的工具可以方便地对享有的虚拟机进行资源的监视和管理。

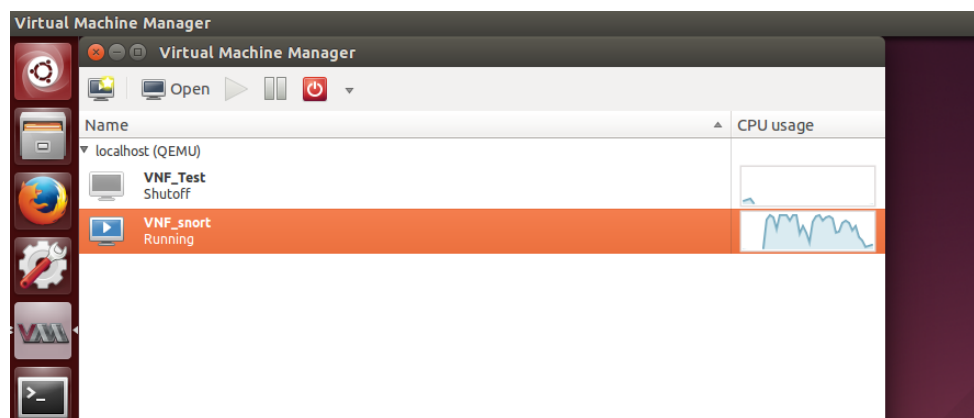


图 4-11 virt-manager 管理 KVM 虚拟机

4.3.3 构建虚拟机网络

KVM 虚拟化默认有三种网络连接方式：仅主机模式、地址转换模式和 Bridge 模式。

其中仅主机模式是将位于宿主机中所有的虚拟机组成一个内网，这个内网无法与外界连通，适用于安全性需求高的场景；NAT 方式是在 KVM 虚拟化技术使用中，模块默认的网络连接方式，可以完成宿主机和虚拟机实现互访，虚拟机也可以访问互联网，但是不支持外界直接对虚拟机进行访问；Bridge 方式即使用 Linux bridge 构建虚拟网桥，使得虚拟机具有独立的 IP，虚拟机可以成为宿主机中具有独立网络的设备，虚拟机和宿主机将各自的网络设备挂在到网桥之上，设定网桥转发规则，虚拟机和宿主机网络互通，但是虚拟机不能够直接访问外部网络。

根据第三章的测试，在宿主机中部署数据平面，如果使用默认的 Linux Bridge，将无法满足 NFV 中数据包高速转发的需求，在三种数据平面中，我们可以选用 DPDK-OVS 这一数据面，可以将 DPDK 部署在宿主机和虚拟机中，避免数据包进入内核态，然后使用 OpenvSwitch 作为虚拟交换机，OpenvSwitch 相对于传统的 Linux bridge 来说性能有一些提高，并可以提供更加细粒度的流量控制。我们将宿主机的物理网卡和虚拟机的虚拟网卡连接到 OpenvSwitch 的相应端口上，在由远程的 SDN controller 来下发流表项，即可构建虚拟机网络。所构建的 KVM 虚拟机网络如图 4-12 所示。

在本节中，我们在一个宿主机中使用了一个 OpenvSwitch 作为虚拟交换机，而未采用 4.2.3 节中构建容器网络时，每个宿主机中部署多个交换机的方式，是因为在 DPDK 的加入下，OpenvSwitch 的性能本身已经得到了很大的提升，相比 4.2.3 节中仅采用一个虚拟交换机时带来的性能损耗，OpenvSwitch 增加端口所造成的性能损耗要小得多，并且如果宿主机中如果 OpenvSwitch 的数量过多，也会导致 SDN

controller 的管理维护困难。

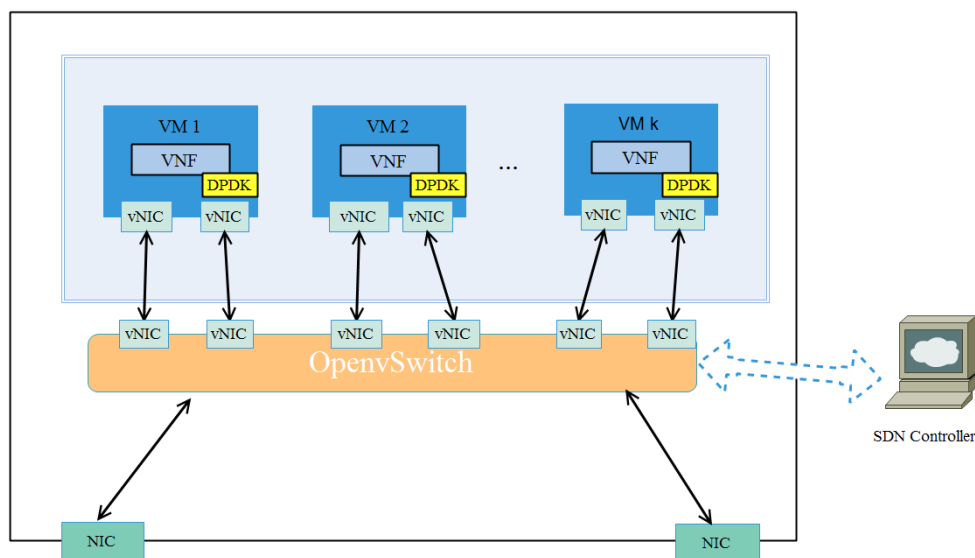


图 4-12 KVM 虚拟机网络

4.4 性能测试

4.4.1 测试环境

为比较 Docker 容器技术和 KVM 虚拟化技术的性能差异，我们搭建了实验环境来对二者进行性能上的测试。

测试环境由三台 DELL 服务器组成，服务器配备 2 颗至强 E52600 处理器，每颗处理器有 8 个核心，CPU 主频率为 1.7GHz，内存类型为 DDR4，内存大小为 8G，并配备两块 Intel 千兆网卡，操作系统为 CentOS，一台 Pica8 SDN 交换机。测试的内容为 Docker 容器和 KVM 虚拟机的 CPU 计算能力，内存访问以及读写能力，网络吞吐能力和重启时间长度。

我们使用 SysBench 测试工具来测试 CPU 的运算能力。使用 MBW 内存性能测试工具来比较 Docker 容器和 KVM 虚拟机的内存性能差异。网络吞吐能力采用与第三章相同的 iperf 测试工具进行测试。重启时间长度根据系统标记时间来衡量。所有的实验均独立运行 10 次取其平均值。

4.4.2 测试结果

我们首先对优化后的容器网络连接进行了性能测试，我们采取了第三章图 3-22 的测试环境，在 NFV 服务器上使用 Docker 容器进行不同长度的网络服务功能链的部署，并从发送端服务器发送不同大小的数据包到接收端服务器，对比在不同服务功能链长度的条件下，优化容器网络连接方式前后的性能差异。

由于单服务器内服务功能链长度一般在 10 以内，我们测试了功能链长度为 3 和长度为 6 条件下，优化容器网络连接前后的 NFV 服务器的包转发能力，其结果如图 4-13 和 4-14 所示。

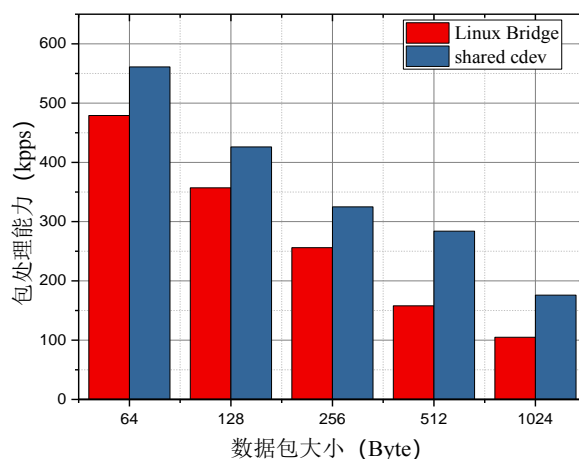


图 4-13 三个 VNF 构成功能链时性能

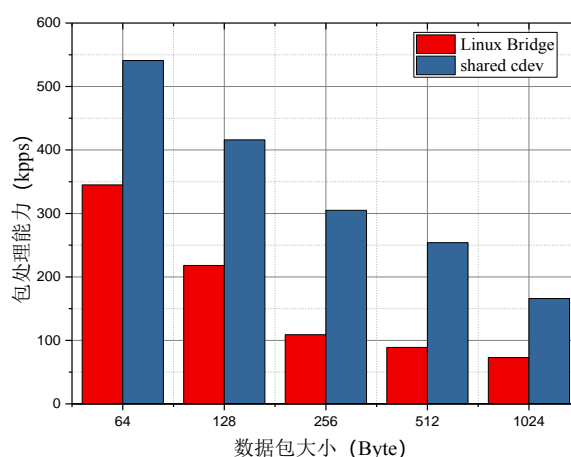


图 4-14 六个 VNF 构成功能链时性能

可以看到在由 3 个 VNF 组成的服务功能链中，使用单个 Linux Bridge 进行转发数据包的性能差于使用共享内存设备的性能，其中当数据包较小时，优化后性能提升较小，大约在 5% 左右，当数据包较大时，由于优化后的容器网络可以减少数据包的频繁拷贝问题，优化效果显著，性能提升可以达到 45% 左右。

在由 6 个 VNF 构成的服务功能链环境下，可以看到使用共享内存的优化效果更加显著，性能提升最高达到了 179%，这是由于服务功能链的加长，导致了数据包拷贝次数增多，而使用共享内存的字符设备进行数据包转发，其数据包拷贝次数仍然是两次，不受服务功能链长度变化影响，该方案优化效果显著。

针对 KVM 虚拟机和 Docker 容器的性能测试，我们首先对二者的 CPU 计算能力进行了测试，通过 SysBench 测试工具分别测试宿主机、Docker 容器和 KVM 虚拟机，测试三者生成素数所需要的时间，生成素数的个数从 5000 依次递增到 25000，其测试结果如图 4-15 所示。

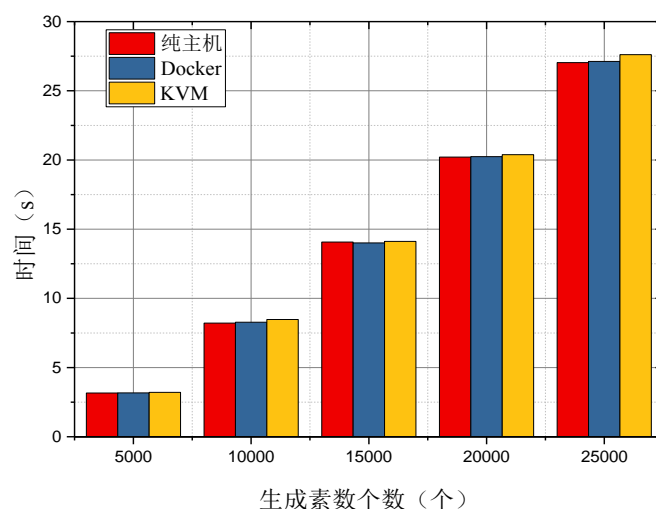


图 4-15 CPU 性能对比

可以看到，未安装 Docker 容器和 KVM 虚拟机的纯主机与仅运行 Docker 容器或 KVM 虚拟机的主机，在各自的计算环境下其 CPU 计算能力相差无几。Docker 容器相当于一个进程，直接运行在宿主机上，所以其运行计算性能测试时，只是测试线程在物理主机的 CPU 上执行，并不存在性能的损耗。

而运行 KVM 虚拟机的宿主机尽管多运行着一层 Hypervisor，但其计算能力相比于纯主机而言，所产生的性能损耗并不大，究其原因是因为当虚拟机进行计算运算时，Hypervisor 只是一个 Virtual Machine Monitor，其并不参与运行指令的转换。由于当前的通用服务器的 CPU 都支持虚拟化，KVM 虚拟机中的 GuestOS 在进行运算时只需要从宿主机的 VMX Non-root 模式自动切换到 VMX root 模式，其运行的计算代码仍然是直接运行在底层的物理 CPU 上的，即 CPU 并未被 KVM 内核虚拟化为虚拟 CPU，这使得 KVM 虚拟机只是在客户机到宿主机的指令转换时有一定时延，而当虚拟机开始计算时，其 CPU 计算能力并未受到损失。

为测试多用户环境下 KVM 虚拟机与 Docker 容器计算性能的对比，我们在实验环境中的物理宿主机分别部署了五个 KVM 虚拟机和五个 Docker 容器，每个虚拟机内存为 1G，硬盘为 10GB，运行 64 位 Ubuntu16.04.4 操作系统。五台 KVM 虚拟机和五个 Docker 容器同时进行生成素数测试，从而测试其计算能力，测试结果如图 4-16 所示。

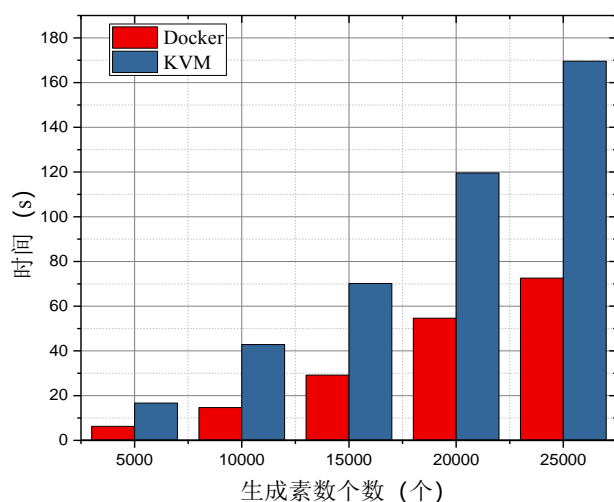


图 4-16 多个客户机时 CPU 性能对比

可以看到当部署多个 Docker 容器或多个 KVM 虚拟机时，KVM 虚拟机生成同样个数的素数需要更多的时间，其计算能力相比 Docker 出现了大幅下滑。这是因为多个 KVM 在 Hypervisor 之间进行切换时，需要占用一定的宿主机 CPU 资源，从而导致了其计算能力的下降。而 Docker 容器只需要直接竞争 CPU 资源，完成计算即可。

我们使用了 MBW 内存性能测试工具来比较 Docker 容器和 KVM 虚拟机的内存性能差异。其测试结果如图 4-17 所示。

可以看到，在内存性能上，Docker 容器基本与纯主机一致。相对而言，KVM 虚拟机的内存性能有一部分损耗。这是因为在 KVM 内存虚拟化中，涉及到虚拟机和宿主机的内存映射问题。

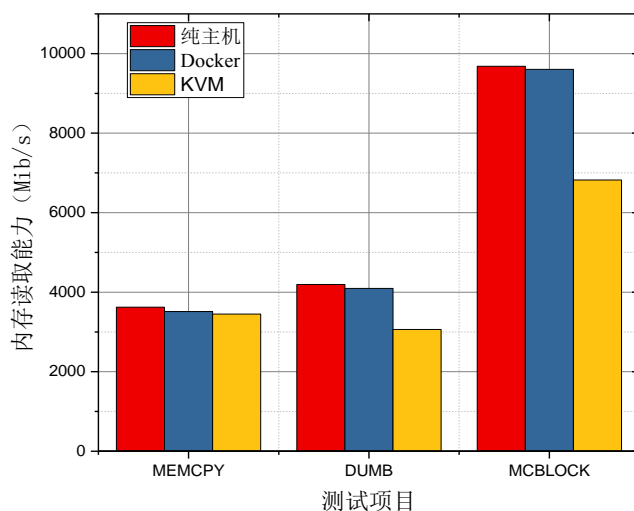


图 4-17 内存性能对比

在 KVM 虚拟机中涉及到四种地址：客户虚拟机虚拟地址，客户虚拟机物理地址，物理宿主机虚拟地址，以及物理宿主机物理地址。KVM 内核通过利用系统调用，将运行在宿主机中的 QEMU 进程中的宿主机虚拟地址映射到客户机物理地址，而在虚拟机中，通过内部操作系统中的地址映射表来维护客户虚拟机虚拟地址到客户虚拟机物理地址之间的映射关系。与此同时，由于我们会在宿主机中部署多台虚拟机，在涉及到宿主机虚拟地址到宿主机物理地址映射时，由于不同虚拟机和宿主机之间的权限可能不同，也会造成多次查询开销，导致性能损耗。

相比于 Docker 只需要完成在宿主机中的虚拟地址到物理地址的映射，KVM 虚拟机由于多进行了两次地址映射，从而影响了内存的性能表现。

在网络吞吐性能上，我们采用了和第三章相同的 iperf 工具来进行测试，通过在发送端不断加大发送网络报文的速度，在接收端对比通过分别部署两种虚拟化方式的网络流量的速度，进而得出两种虚拟化方式中各自服务器的吞吐量。测试结果如图 4-18 所示。

从测试结果可以看到，Docker 容器与 KVM 虚拟机在网络吞吐量上能力几乎均达到了发送端的最大速度。KVM 在早期使用 QEMU 模拟网络功能，性能仅能达到物理宿主机的三分之一左右，在新增了 virtio 技术和 vhost 技术后，虚拟机所发出的网络 I/O 请求将不会再在虚拟机和宿主机之间进行切换，而是直接由 QEMU 的驱动程序进行拦截并和宿主机中的真实网络设备驱动进行通信，来完成虚拟机的 I/O 操作，这使得 KVM 虚拟机的网络吞吐能力大幅提升。

而 Docker 容器相当于一个运行在宿主机中的进程，其容器的隔离性使用 Linux Namespace 功能来隔离容器的运行环境，使用 cgroups 功能来限制容器使用的资源，这使得 Docker 容器的网络吞吐性能也接近物理宿主机的吞吐性能。

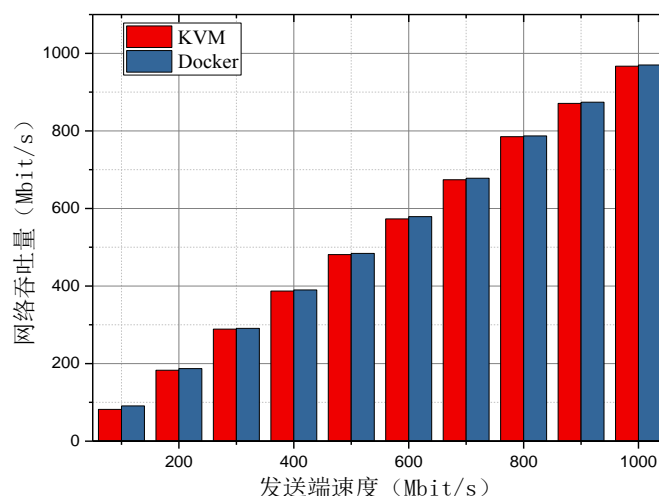


图 4-18 网络吞吐性能对比

最后我们对 Docker 容器和 KVM 虚拟机重启时间长度进行了测试。我们在实验环境中的物理宿主机分别部署了五个 KVM 虚拟机和五个 Docker 容器，每个虚拟机内存为 1G，硬盘为 10GB，运行 64 位 Ubuntu16.04.4 操作系统。测试结果如图 4-19 所示。

可以看到，Docker 容器重启时间长度与 KVM 虚拟机重启时间长度差别巨大，且随着同时重启数目的增多，这种差别越来越大。这是因为相对 Docker 只是宿主机系统中的一个进程而言，KVM 虚拟机是运行在 Hypervisor 之上的一个完整的环境，在这个过程中还需要重启虚拟机中的操作系统。

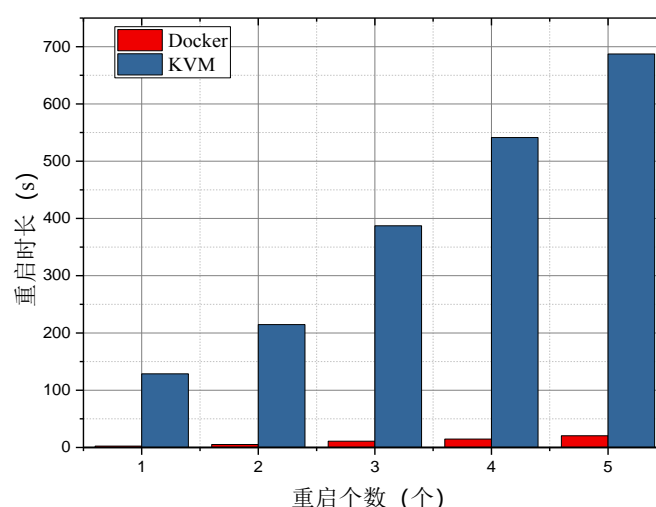


图 4-19 Docker 容器与 KVM 虚拟机重启时间对比

KVM 虚拟机重启首先会涉及到重启开始时虚拟机资源的清理工作，将系统资源回收和再释放，并不断检查，以防再启动时造成资源前后不一致。之后将会加载完整的镜像文件，加载磁盘文件，读取文件信息，加载内核资源，启动内核模块，启动 login 程序，最后进入到登陆状态。整个过程基本与宿主机重启过程类似。

由以上比较可以看出，在使用性能上，由于 Docker 属于操作系统级虚拟化，只是相当于一个进程运行在宿主机服务器上，其拥有的 CPU 计算能力、内存读取能力、网络吞吐能力以及重启的启动时间上均贴适于宿主机具有的能力。而 KVM 虚拟化方式相对而言属于一种较为重的虚拟化方式，其采用全虚拟化的方式向用户提供了一个完整的运行环境，在一些性能上会造成损耗。当进行 VNF 部署时，IO 密集型需求和计算密集型需求部署于 Docker 容器中更容易获得最佳性能。但也要看到，KVM 虚拟机在使用过程中也具有 Docker 容器所不具备的一些优势。

KVM 虚拟机的资源隔离性要优于 Docker 容器，这是因为 Docker 使用 cgroup 技术实现资源隔离的，其只能限制资源消耗的最大值，可能会造成与其他程序共享资

源的情况。KVM 虚拟机的安全性优于 Docker 容器，Docker 容器的用户拥有该宿主主机上所有容器的使用权限，这可能会造成一些安全隐患，而 KVM 虚拟机拥有独立的操作系统，可以使不同用户拥有不同的操作权限。同时，KVM 虚拟化技术还支持热迁移，可以使虚拟机在不停机的情况下单点迁移，而这是目前的 Docker 技术所不具有的。

综上，KVM 虚拟化技术和 Docker 容器技术在性能和隔离性安全性上各有其优势，应该根据网络功能虚拟化场景的不同，按照需求来决定所采用的虚拟化的手段。

4.5 本章小结

在本章中，我们针对网络功能虚拟化实际部署中的需求，对 Docker 容器技术和 KVM 虚拟化技术进行了对比，搭建了实验环境，在 Docker 容器中和 KVM 虚拟机中部署了具体的虚拟网络功能，并对服务器中构建基于容器的服务功能链网络实现做出改进。最后搭建实验环境，对 Docker 容器技术和 KVM 虚拟化技术的性能差异进行了对比，并对之后的使用场景提出建议。

第五章 平台搭建与测试

通过第三章和第四章，我们对 NFV 部署平台中的数据转发平面和虚拟化平面进行了部署、对比和性能测试，本章将在以上测试结果的基础上，搭建综合测试平台，实现部署虚拟网络功能，进行功能测试和性能测试。

5.1 实验平台

5.1.1 环境配置

在第三章数据平面的部署、性能测试中，我们可以看到 DPDK-OVS 在性能上比 NetMap 和 Click 稍好，其在小包的转发处理上性能更加优异。同时，在虚拟 I/O 的选择上，DPDK-OVS 与 vhost-user 的组合性能表现比 NetMap 和 Click 搭配 virtio-net 虚拟 I/O 的性能表现更好。因此，我们在数据平面上选择在 DPDK-OVS 数据面的基础上搭建 KVM 虚拟机，来进行测试。

而针对 Docker 容器，如第四章所示，可以选用 NetMap Vale 来作为数据平面，构建容器网络，实现数据包的转发。

我们使用了四台 DELL R630 服务器、一台 Pica8 SDN 交换机和一台笔记本电脑搭建实验环境。每台服务器配备 2 颗至强 E52600 处理器，CPU 频率为 1.7GHz，内存类型为 DDR4，内存大小为 8G，两块 Intel 千兆网卡，每台服务器均运行 64 位 Ubuntu 14.04 操作系统。

测试平台由三台 DELL R630 服务器和一台 Pica8 SDN 交换机组成，每台服务器配备 2 颗至强 E52600 处理器，CPU 频率为 1.7GHz，内存类型为 DDR4，内存大小为 8G，两块 Intel 千兆网卡，三台服务器均运行 64 位 Ubuntu 14.04 操作系统。Pica8 SDN 交换机作为外部的交换机连接三台服务器，流表中流表项由远程的主机控制下达。

5.1.2 环境搭建

在实际的测试环境搭建中，我们采用的系统模型如图 5-1 所示。

其中各个服务器功能如下：

(1) NFV 服务器：我们在两台服务器上分别创建 KVM 虚拟机和 Docker 容器，并部署相关的虚拟网络功能，完成相应的功能测试和性能测试。

(2) 发送端/接收端服务器：我们在两台服务器上分别运行 tcpreplay 软件发送网络流量和运行 tcpdump 软件来捕获流量，通过控制流量速率来进行性能测试。

(3) 一台 Pica 8 SDN 交换机和一台笔记本电脑：通过笔记本电脑上的 SDN 控制器 Floodlight 对交换机进行流表项的配置。

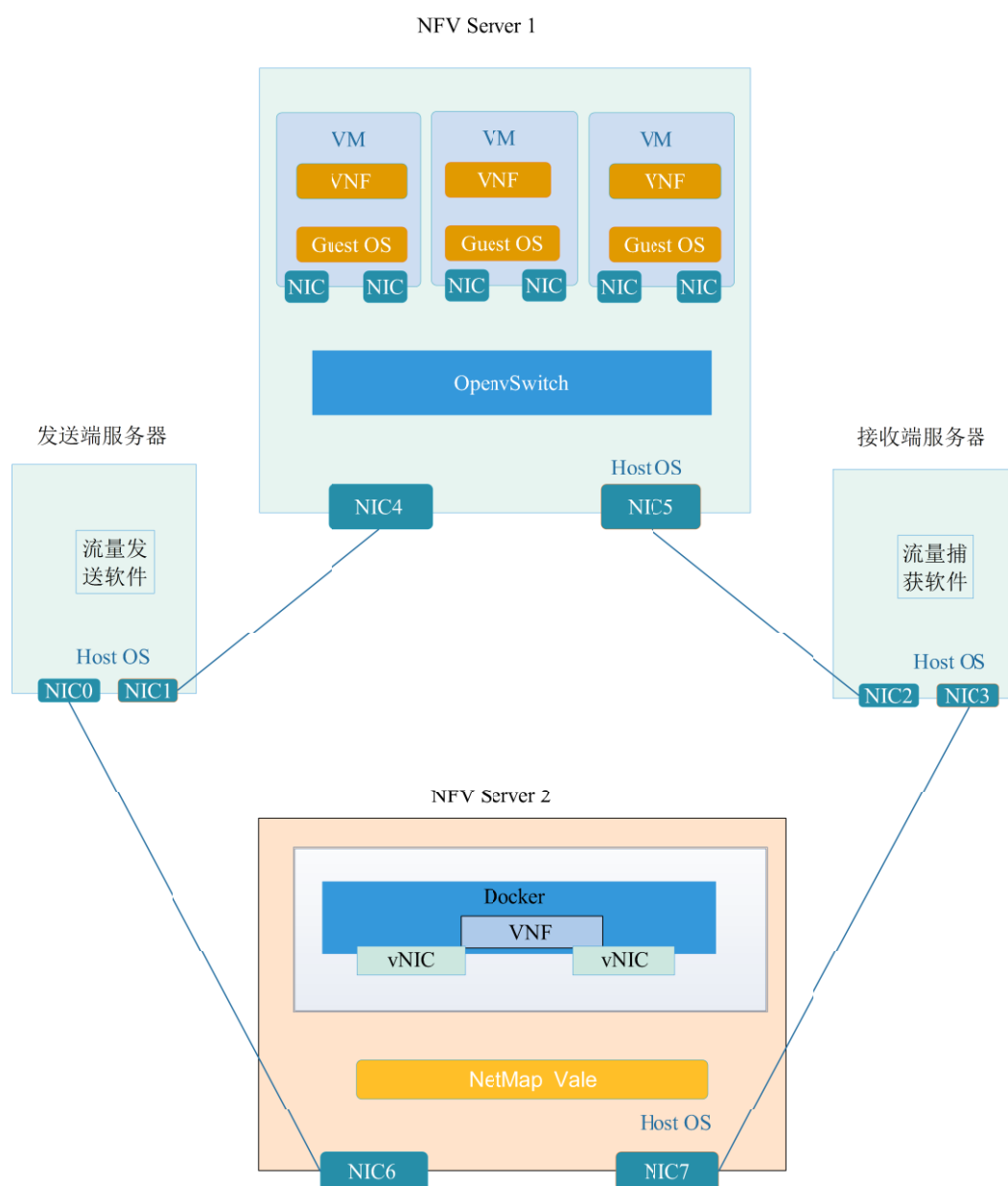


图 5-1 系统连接图

其中，每个物理网卡的配置如表 5-1 所示。

为对 KVM 虚拟机进行编排配置和动态迁移等功能测试，我们在 NFV 服务器 1 上使用了 OpenStack 云平台管理工具，版本为 OpenStack Mitaka。

OpenStack 是一个开源的云计算管理平台项目，其可以帮助用户更加方便的创建虚拟机，实现虚拟机的编排、管理、扩容缩容等功能，它是使用 KVM 作为最底层的 Hypervisor 来实现虚拟化的。OpenStack 一般由控制节点、网络节点和计算节

点三个节点组成，其核心服务有认证服务 Keystone、镜像服务 Glance、计算服务 Nova、网络服务 Neutron 和块存储服务 Cinder 等核心服务。其中控制节点主要负责对其余节点的指令控制，以及对虚拟机的创建、删除、迁移、资源分配等，计算节点主要负责提供计算能力，负责虚拟机的运行，网络节点主要负责整个集群中内网和外部网络的沟通。

表 5-1 物理网卡配置表

物理网卡名	网卡 IP 地址
NIC 0	192.168.10.20
NIC 1	192.168.10.10
NIC 2	192.168.10.30
NIC 3	192.168.10.40
NIC 4	192.168.10.50
NIC 5	192.168.10.60
NIC 6	192.168.10.70
NIC 7	192.168.10.80

如图 5-2 所示，我们创建了三台 KVM 虚拟机来搭建 OpenStack 集群，分别配置为 OpenStack 的控制节点、计算节点和网络节点。

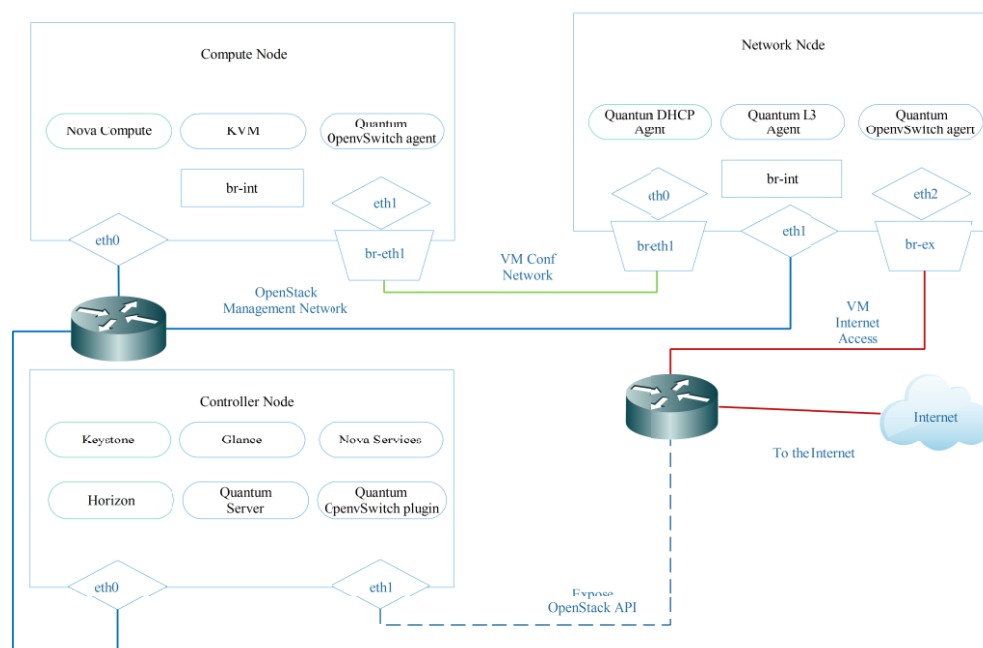


图 5-2 OpenStack 节点网络连接图

三台虚拟机的配置如表 5-2 所示，每台虚拟机均安装 ubuntu14.04 操作系统。

表 5-2 集群配置信息

Node	CPU	内存	存储
Controller	1 核	1G	40G
Compute	2 核	4G	40G
Network	1 核	2G	40G

每个虚拟机节点配置的 OpenStack 组件如图 5-3 所示。

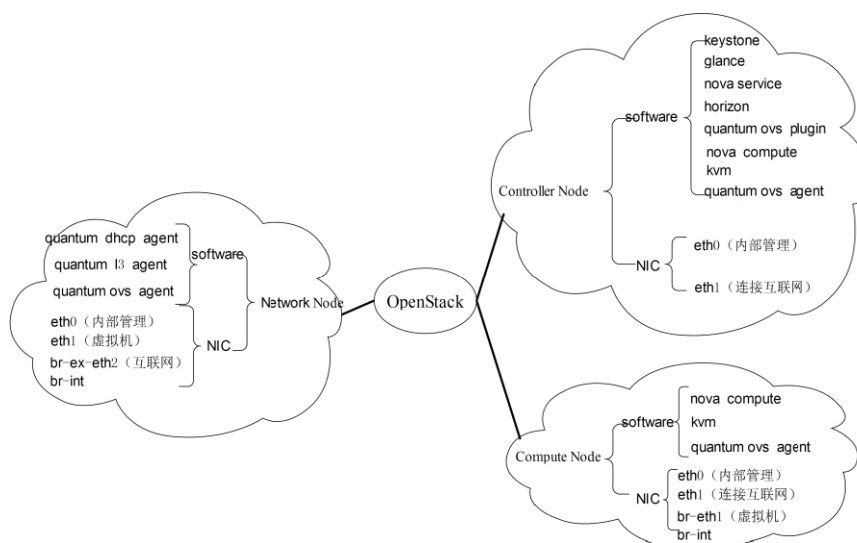


图 5-3 节点服务配置

如图 5-4 所示，配置完成后可以通过登陆控制节点的 Horizon 服务，使用网页完成对 KVM 虚拟机的编排和创建。

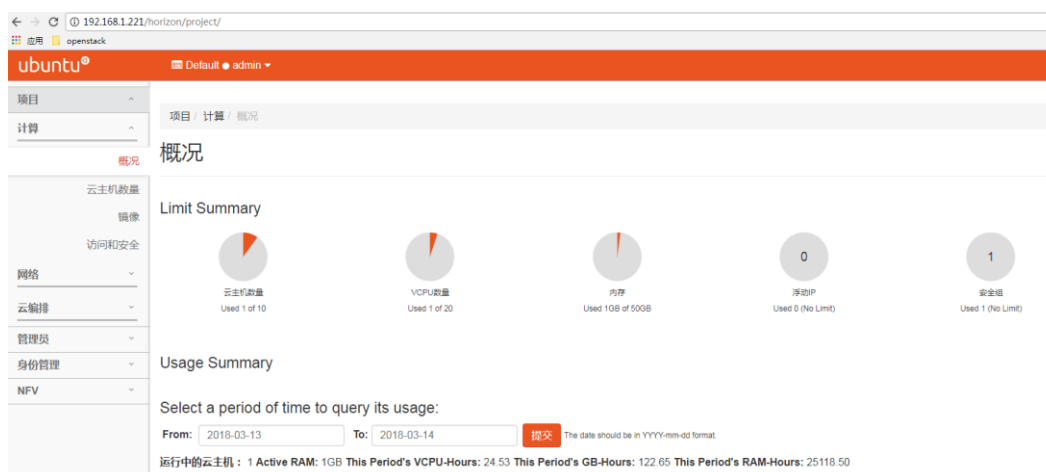


图 5-4 OpenStack 服务

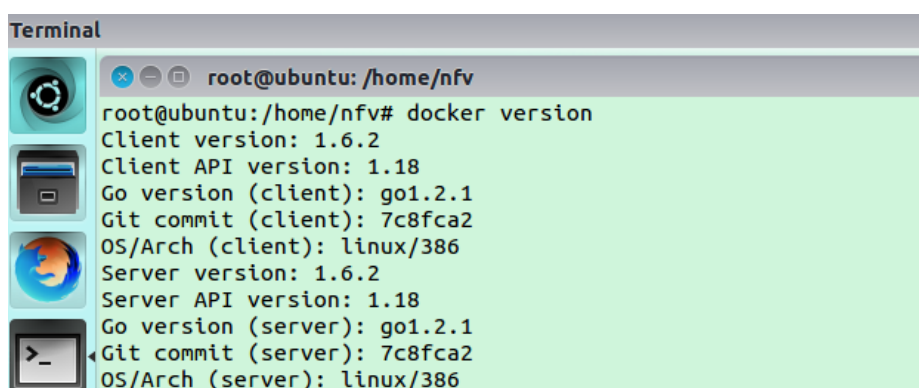
为对 Docker 容器进行网络功能虚拟化功能测试和性能测试，我们在 NFV 服务器 2 上搭建了 Docker 使用环境。Docker 版本为 1.6.2。

安装过程中需要配置相关环境，并获取最新的 Docker 安装包，执行安装。安装过程如图 5-5 所示。

```
1 //1.配置Docker安装源
2 sudo apt-get update
3 sudo apt-get install apt-transport-httpsudo apt-get
4 sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 36A1D7869245C89
5 50F96657SAFDFA6R856FDFADF5
6 sudo sh -c "echo deb https://get.docker.com/ubuntu docker main > /etc/apt/sources.
7 list.d/docker.list"
8 //2.安装Docker
9 sudo apt-get uodate
10 sudo apt-get install lxc-docker
11 //3.启动Docker并配置Docker开机启动
12 sudo apt-get install sysv-rc-conf
13 sudo sysc-rc-conf docker on
14 sudo service docker restart
```

图 5-5 安装 Docker 实现过程

配置环境并执行安装程序，成功后运行 Docker 容器并查看版本，结果如图 5-6 所示。

A terminal window titled 'Terminal' with a dark background. The prompt is 'root@ubuntu: /home/nfv'. The command 'docker version' has been executed, and the output is displayed in a light green background. The output shows client and server versions (1.6.2), API versions (1.18), Go versions (go1.2.1), Git commits (7c8fca2), and OS/Arch (linux/386).

```
Terminal
root@ubuntu: /home/nfv
root@ubuntu: /home/nfv# docker version
Client version: 1.6.2
Client API version: 1.18
Go version (client): go1.2.1
Git commit (client): 7c8fca2
OS/Arch (client): linux/386
Server version: 1.6.2
Server API version: 1.18
Go version (server): go1.2.1
Git commit (server): 7c8fca2
OS/Arch (server): linux/386
```

图 5-6 查看 Docker 版本

5.2 实现虚拟网络功能

5.2.1 虚拟机部署虚拟网络功能

在 OpenStack 上，新建虚拟机需要准备相关的镜像文件，而在虚拟机中部署虚拟网络功能，可以先将虚拟机启动，再采用 SSH 登陆工具进入该虚拟机，下载相关软件，完成功能部署。也可以采用制作镜像文件的方法，直接将所需要的虚拟网络功能软件制作在镜像文件中，设置好配置指令，再直接启动虚拟机。为方便未来大规模部署，我们采用后一种方式，来完成虚拟网络功能在 KVM 虚拟机中的部署，所实现的虚拟网络功能为入侵检测功能，使用 Snort 软件来实现。

第一步需要下载 Ubuntu 镜像，我们选用的基础镜像为 ubuntu-14.04.5-server-

amd64.iso，选用镜像制作工具 OZ，修改默认镜像格式为 QCOW2 格式。

第二步创建配置 tdl 脚本文件，其部分关键步骤如图 5-7 所示。

```

1 <!-- 替换源文件 -->
2 <file name='/etc/apt/sources.list' type='url'>file:///sources.list14</file>
3 <!-- 导入snort安装文件 -->
4 <file name='/tmp/daq-2.0.6.tar.gz' type='url'>
5 http://192.168.1.109/file/daq-2.0.6.tar.gz
6 </file>
7 <file name='/tmp/snort-2.9.9.0.tar.gz' type='url'>
8 http://192.168.1.109/file/snort-2.9.9.0.tar.gz
9 </file>
10 <file name='/tmp/snortrules-snapshot-2976.tar.gz' type='url'>
11 http://192.168.1.109/file/snortrules-snapshot-2976.tar.gz
12 </file>
13 <!-- 安装snort-->
14 <command name='snort'>
15 apt-get install -y zlib1g-dev liblzma-dev openssl libssl-dev
16 apt-get install -y bison flex
17 apt-get install -y libpcap-dev libpcap3-dev libdumbnet-dev
18 apt-get install -y build-essential
19 tar xvfz /tmp/daq-2.0.6.tar.gz -C /tmp
20 cd /tmp/daq-2.0.6
21 ./configure
22 make
23 sudo make install
24 tar xvfz /tmp/snort-2.9.9.0.tar.gz -C /tmp
25 cd /tmp/snort-2.9.9.0
26 ./configure --enable-sourcefire
27 make
28 sudo make install
29 </command>

```

图 5-7 配置镜像制作文件

第三步修改镜像格式为 QCOW2，并压缩镜像文件。至此，一个包含了 snort 入侵检测软件的虚拟机镜像文件制作完成。结果如图 5-8 所示。

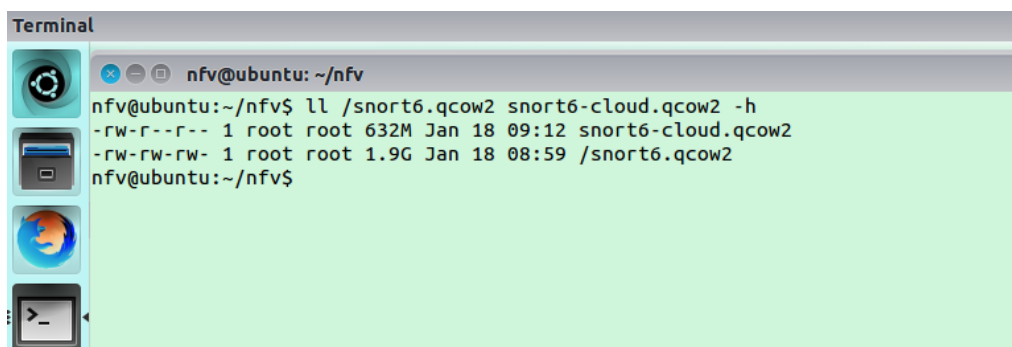


图 5-8 制作的镜像文件

第四步，上传至 OpenStack 镜像库。效果如图 5-9 所示。



图 5-9 OpenStack 镜像库

5.2.2 容器部署虚拟网络功能

为了与 KVM 虚拟机进行对比测试，我们在 Docker 容器中部署了虚拟网络功能，仍然选用 Snort 软件实现入侵检测功能。在实现中，需要构建 Dockerfile，来实现指令的下达。其中，我们创建的部署 Snort 软件的 Dockerfile 部分过程和关键步骤如图 5-10 所示。

```

1 //1.定义基础镜像和维护人
2 FROM ubuntu:14.04
3 MAINTAINER Shawn Guo "shawn_guo@163.com"
4
5 //2.配置环境变量
6 # Specify container username e.g. training, demo
7 ENV VIRTUSER snort
8 # Specify program
9 ENV PROG snort
10 # Specify source extension
11 ENV EXT tar.gz
12 # Specify Snort version to download and install
13 ENV VERS 2.9.8.0
14 # Specific libpcap to download and install
15 ENV LVERS libpcap-1.7.4
16
17 //3.下载Snort
18 # Compile and install Snort
19 USER $VIRTUSER
20 WORKDIR /home/$VIRTUSER
21 #RUN wget --no-check-certificate https://www.snort.org/downloads/snort/$PROG-$VERS.tar.gz
22 COPY snort-2.9.8.0.tar.gz /home/$VIRTUSER
23 RUN tar -zxvf $PROG-$VERS.$EXT
24 WORKDIR /home/$VIRTUSER/$PROG-$VERS

```

图 5-10 创建配置 Snort 软件的 Dockerfile

之后运行 Docker 容器。至此，在 Docker 容器中的虚拟网络功能入侵检测系统即部署完成。

5.3 虚拟网络功能测试

为确保虚拟网络功能在两台 NFV 服务器上部署成功，我们在实验平台上进行了对虚拟网络功能 Snort 入侵检测系统进行了功能测试。

在测试平台上，我们从发送端服务器向接收端服务器发送网络流量，两条网络流量分别经过运行着入侵检测功能的 KVM 虚拟机和 Docker 容器，在两个 VNF

处，根据发送端服务器和接收端服务器的 IP 地址，设置相应的入侵检测规则，来验证 VNF 的功能是否正常运行。

我们指定从发送端服务器的网卡 1 向接收端服务器的网卡 2 发送网络流量，其流量途经部署了 KVM 虚拟机的 NFV 服务器 1；指定从发送端服务器的网卡 0 向接收端服务器的网卡 3 发送网络流量，其流量途经部署了 Docker 容器的 NFV 服务器 2。

首先我们对网络连接进行了测试，在发送端服务器向接收端服务器发送网络数据包，结果如图 5-11 所示，说明网络连接正常，数据流量可以送达接收端服务器。

```
nfv@ubuntu:~$ ping -I 192.168.10.10 192.168.10.30
PING 192.168.10.30 (192.168.10.30) from 192.168.10.10 : 56(84) bytes of data.
64 bytes from 192.168.10.30: icmp_seq=1 ttl=64 time=0.649 ms
64 bytes from 192.168.10.30: icmp_seq=2 ttl=64 time=0.488 ms
64 bytes from 192.168.10.30: icmp_seq=3 ttl=64 time=0.512 ms
64 bytes from 192.168.10.30: icmp_seq=4 ttl=64 time=0.634 ms
64 bytes from 192.168.10.30: icmp_seq=5 ttl=64 time=0.546 ms
64 bytes from 192.168.10.30: icmp_seq=6 ttl=64 time=0.539 ms
```

(a)

```
nfv@ubuntu:~$ ping -I 192.168.10.20 192.168.10.40
PING 192.168.10.40 (192.168.10.40) from 192.168.10.20 : 56(84) bytes of data.
64 bytes from 192.168.10.40: icmp_seq=1 ttl=64 time=1.26 ms
64 bytes from 192.168.10.40: icmp_seq=2 ttl=64 time=0.419 ms
64 bytes from 192.168.10.40: icmp_seq=3 ttl=64 time=0.485 ms
64 bytes from 192.168.10.40: icmp_seq=4 ttl=64 time=0.655 ms
64 bytes from 192.168.10.40: icmp_seq=5 ttl=64 time=1.88 ms
64 bytes from 192.168.10.40: icmp_seq=6 ttl=64 time=0.602 ms
```

(b)

图 5-11 发送端服务器 ping 接收端服务器。(a)流量途经 NFV 服务器 1；(b)流量途经 NFV 服务器 2

在 KVM 虚拟机中设置针对网络流量 1 的告警规则，在 Docker 容器中设置针对网络流量 2 的告警规则，规则如图 5-12 所示。

```
alert icmp 192.168.10.10 any -> 192.168.10.30 any (msg: " KVM—flow1 ";sid:100001)
```

(a)

```
alert icmp 192.168.10.20 any -> 192.168.10.40 any (msg: " Docker-flow2 ";sid:100001)
```

(b)

图 5-12 VNF 中 Snort 告警规则。(a)KVM 虚拟机中 Snort 告警规则；(b)Docker 容器中 Snort 告警规则

之后在发送端服务器分别向接收端服务器发送网络流量，停止后检查两个 NFV 服务器上入侵检测系统的告警日志。

可以看到在 KVM 虚拟机和 Docker 容器中，告警日志均已检测到相应的网络流量，入侵检测功能运行正常，入侵检测结果如图 5-13 所示。

```
02/26-16:15:13.164956  [**] [1:100001:0] " KVM-flow1 " [**] [Priority: 0] {ICMP} 192.168.10.10
-> 192.168.10.30
02/26-16:15:14.164567  [**] [1:100001:0] " KVM-flow1 " [**] [Priority: 0] {ICMP} 192.168.10.10
-> 192.168.10.30
02/26-16:15:15.164590  [**] [1:100001:0] " KVM-flow1 " [**] [Priority: 0] {ICMP} 192.168.10.10
-> 192.168.10.30
02/26-16:15:16.166108  [**] [1:100001:0] " KVM-flow1 " [**] [Priority: 0] {ICMP} 192.168.10.10
-> 192.168.10.30
```

(a)

```
02/26-14:43:04.242200  [**] [1:100001:0] " Docker-flow2 " [**] [Priority: 0] {ICMP} 192.168.10.20
-> 192.168.10.40
02/26-14:43:04.242200  [**] [1:100001:0] " Docker-flow2 " [**] [Priority: 0] {ICMP} 192.168.10.20
-> 192.168.10.40
02/26-14:43:04.242200  [**] [1:100001:0] " Docker-flow2 " [**] [Priority: 0] {ICMP} 192.168.10.20
-> 192.168.10.40
02/26-14:43:04.242200  [**] [1:100001:0] " Docker-flow2 " [**] [Priority: 0] {ICMP} 192.168.10.20
-> 192.168.10.40
```

(b)

图 5-13 VNF 中 Snort 告警信息。(a)KVM 虚拟机中 Snort 告警信息；(b)Docker 容器中 Snort 告警信息

5.4 性能测试

5.4.1 网络性能测试

在所搭建的实验平台上，我们使用了两种方式实现网络功能虚拟化平台的部署，在 NFV 服务器 1 上使用了 DPDK-OVS 作为数据转发平面，使用 KVM 虚拟机作为虚拟化手段，在 KVM 虚拟机中部署虚拟网络功能；在 NFV 服务器 2 上使用了 NetMap VALE 虚拟交换机作为数据转发平面，使用 Docker 容器作为网络虚拟化的部署环境。

在第三章和第四章可以看到，DPDK-OVS 作为数据转发平面在处理小包时其性能表现要优于 NetMap VALE，而在其他性能表现上差距不大；KVM 虚拟机由于在宿主机上增加了 Hypervisor 虚拟机监控层，这导致了在我们的性能测试中 KVM 虚拟机在内存性能、启动时长等性能指标上弱于 Docker 容器，但由于 KVM 虚拟机向用户层提供了一整套虚拟环境，其相比于 Docker 容器也具有一些其他的优势，比如实时迁移、编排配置等。

本节我们将对部署了不同转发平面和虚拟化方式的两台服务器进行网络性能测试，对 KVM 虚拟机具有的实时迁移能力进行测试。

我们在如图 5-1 所示的实验环境中进行测试，在发送端服务器向接收端服务器发送网络流量，分别测试网络流量通过 NFV 服务器 1 和 NFV 服务器 2 时数据包转发性能，其测试结果如图 5-14 所示。

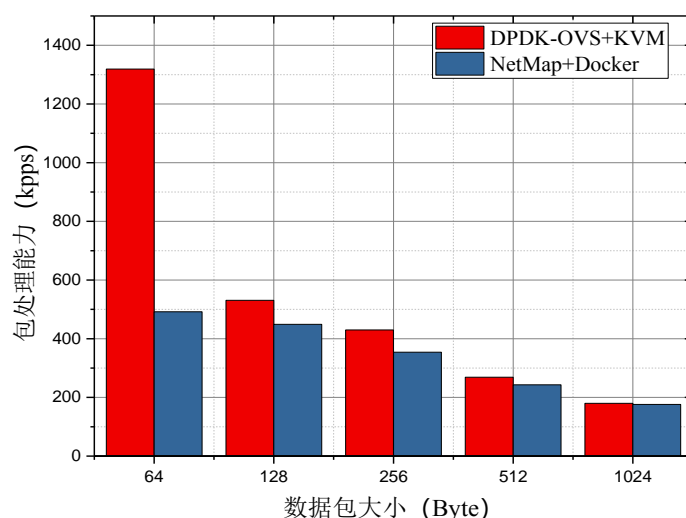


图 5-14 包转发能力测试结果

从测试结果可以看到，搭载 DPDK-OVS 和 KVM 虚拟机的 NFV 服务器在处理小包的时候，其性能要远优于搭载 NetMap 和 Docker 容器的 NFV 服务器，这与第三章比较 DPDK-OVS 和 NetMap 时的性能表现基本吻合，表明在处理小包时，选择 DPDK-OVS 更加适合，同时选择 vhost-user 作为虚拟 I/O，上层的虚拟化方式不会造成性能瓶颈。

在处理大包时，两个服务器均出现了处理能力下降的情况，且随着网络数据包的包长增加，两个服务器的处理能力逐渐接近，平台的选择可以依据其他的功能和性能指标。

而对比图 3-26 和图 5-14，当部署了 KVM 虚拟机和 Docker 容器后，两个平台的包转发能力也仅仅出现了小幅度的下降，这表明虚拟化方式的选择，对两个平台的包转发能力影响不大，造成两个平台处理包速率的性能差异主要来自于数据转发平面的包转发能力，与第四章测试两种虚拟化方式的网络性能结果基本一致，即当前虚拟化方式的选择并不会对网络性能造成瓶颈。

之后我们对 NFV 服务器 1 和 NFV 服务器 2 的带宽进行了测试，发送端服务器发送网络包的速率从 100Mbps/s 逐步增加到 1000Mbps/s，在接收端服务器查看收包速率，测试结果如图 5-15 所示。

可以看到部署了不同数据平面和采用了不同虚拟化方式的两台 NFV 服务器均可以达到线速转发数据包，表明整个 NFV 服务平台可以高性能地对外提供服务，数据包的转发不会造成瓶颈，优化方案取得了较为有效的结果。

由 NFV 服务器 1 和 NFV 服务器 2 的网络性能比较可以看出，如何选择虚拟化方式并不会对网络功能虚拟化场景下的流量转发造成瓶颈，选择 KVM 虚拟化方

式或者 Docker 容器虚拟化方式可以参考其他的性能指标。

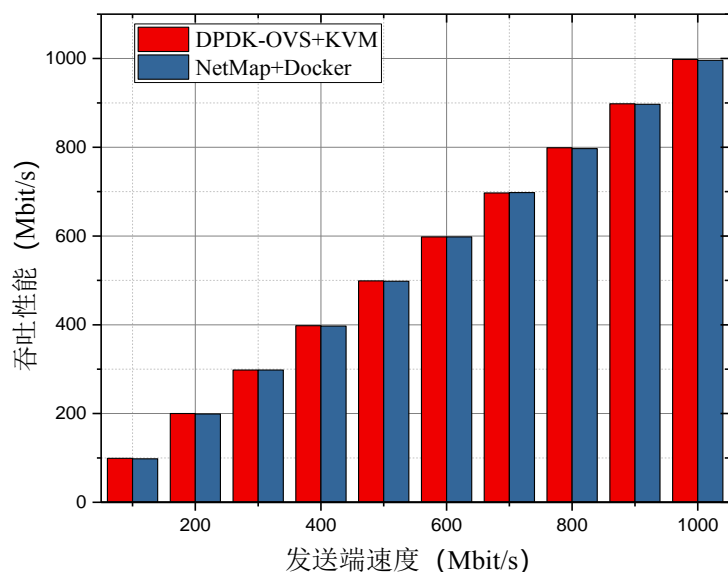


图 5-15 转发性能测试结果

最后我们对两台 NFV 服务器的网络时延进行了测试，我们在每台 NFV 服务器上都部署了不同长度的服务功能链，来对比在相同条件下两种 NFV 平台的网络时延，测试结果如图 5-16 所示。

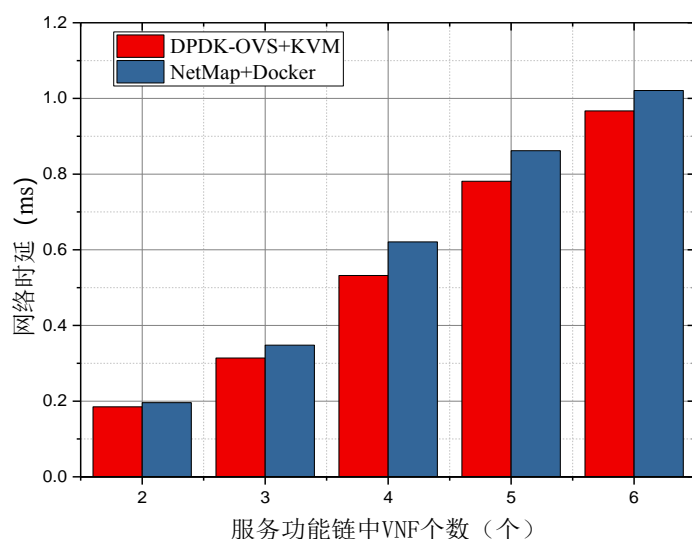


图 5-16 网络时延测试结果

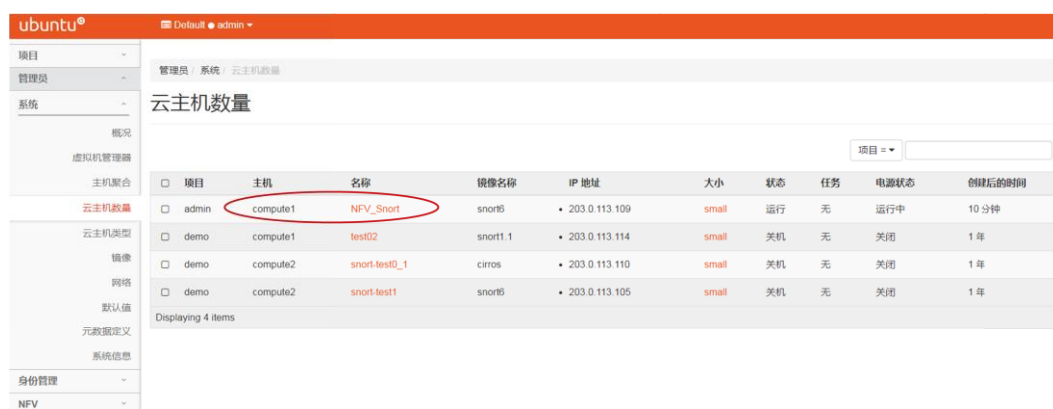
可以看到，我们部署了不同长度的服务功能链，随着其中 VNF 个数的增加网络延时不断增加，并且采用 DPDK-OVS 数据平面和 KVM 虚拟化方式的 NFV 服务器网络时延性能优于采用 NetMap 数据平面和 Docker 容器技术的 NFV 服务器，

这是优于采用的虚拟 IO 上，vhost-user 对于包的转发效率上要优于 virtio-net。

5.4.2 容灾性能测试

根据第四章的测试结果可以看到，Docker 容器在当前技术条件下，其重新启动时间要优于 KVM 虚拟机，这是因为 KVM 虚拟机向用户提供了一个完整的虚拟环境，重新启动需要进行一系列清除和配置操作。而也正是因为其提供了完整的工作环境，在 Hypervisor 的帮助下，KVM 虚拟机可以实现较好的容灾性能，可以做到不停机实时迁移，而当前条件下 Docker 容器并不能做到这一点。

我们首先依靠 OpenStack 管理工具在 NFV 服务器 1 上创建一个 VNF，它是一个包含了 Snort 入侵检测系统的 KVM 虚拟机，其运行在计算节点 Compute1 之上。结果如图 5-17 所示。



项目	主机	名称	镜像名称	IP 地址	大小	状态	任务	电源状态	创建后的时间
admin	compute1	NFV_Snort	snort6	203.0.113.109	small	运行	无	运行中	10 分钟
demo	compute1	test02	snort1.1	203.0.113.114	small	关机	无	关闭	1 年
demo	compute2	snort-test0_1	cirros	203.0.113.110	small	关机	无	关闭	1 年
demo	compute2	snort-test1	snort6	203.0.113.105	small	关机	无	关闭	1 年

图 5-17 创建 VNF

其网络详细信息如图 5-18 所示，ID 为 87b3ed71-27a9-4dd6-9804-e97d49996ce1。

在完成迁移动作之前，需要对 OpenStack 中部署块存储服务并配置相关的文件权限，实现各个计算节点之间可以完成共享存储。之后对当前 KVM 虚拟机进行迁移需要根据其 ID 在控制台下达 Nova 指令，将其从 compute1 节点迁移至 compute2 节点，完成实时迁移。控制指令如图 5-19 所示。如果没有出现错误，控制台不会返回信息，迁移验证可以到控制台进行查看。

实时运行情况如图 5-20 所示，在虚拟机迁移之后进行状态查看，可以看到该 VNF 已经从 compute1 节点转移到 compute2 节点运行，并且运行时间延长，说明并未进行重启，实时迁移成功。

以上测试表明，KVM 虚拟机可以实现整个环境的实时迁移，做到服务不间断，其容灾性能较好。而在当前技术条件下，Docker 容器还无法完成实时迁移。



图 5-18 VNF 信息

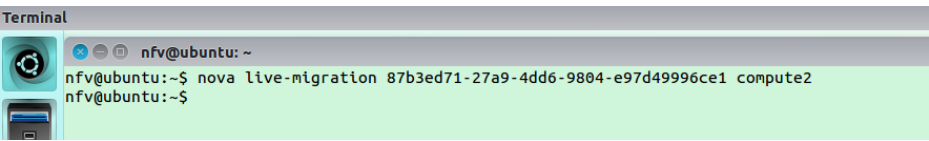


图 5-19 迁移控制指令

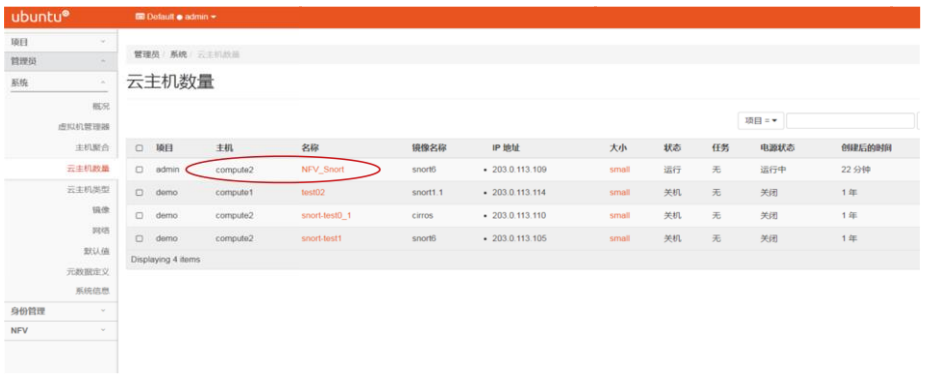


图 5-20 迁移后的 VNF

综上所述可知，两个平台均可以作为网络功能虚拟化的实际部署平台，可以根据实际的性能要求完成不同数据平面和虚拟化方式的选择。当应用于小包流量较多的场景，可以使用 DPDK-OVS 数据平面，但由于其工程上涉及组件较为复杂，当环境内存紧张时可以选择 NetMap 作为虚拟交换机，实现网络流量的快速转发。而在虚拟化方式的选择上，Docker 作为操作系统级的虚拟化方式，其避免了

Hypervisor 造成的性能损耗, 并且部署速度以及重启速度较快, 适用于需要快速部署的场景, 且当部署多个客户机时, 其计算性能不会出现大幅下降, 适用于 IO 密集型需求和计算密集型需求。而 KVM 虚拟机具有实时迁移的能力, 面对容灾需求高的场景, 选择 KVM 虚拟化方式较为合适。

5.5 本章小结

本章根据前两章数据平面和虚拟化方式的部署结果和性能测试结果, 搭建了大规模使用平台, 部署了基于 DPDK-OVS 数据平面的 KVM 虚拟化平台和基于 NetMap 数据平面的 Docker 容器平台, 并在两个平台上分别部署了入侵检测系统虚拟网络功能, 对两个平台进行了功能测试和性能测试, 完成了入侵检测功能, 进行了网络吞吐性能对比和容器性能对比, 并对两个平台所适用的环境进行总结。

第六章 总结与展望

6.1 本论文工作总结

网络功能虚拟化通过对传统的专有电信网络设备软硬件解耦和,并借助相关的虚拟化技术,将网络功能用虚拟化的技术实现并部署于标准化通用服务器之上,可以大大提高物理资源效率。ETSI 定义下的 NFV 标准架构主要由 NFV 基础设施模块、虚拟网络功能模块和 NFV 管理与编排模块三个部分组成。

而在实际应用中,具体的 NFV 使用还有很多的问题尚未解决,比如通用服务器在处理 NFV 中的高速率转发业务的时候,面临着严重的性能瓶颈,需要有针对性地分析在实际使用中如何提高 NFV 的使用效率。在单服务器中,数据包的整个转发流程会有多个过程会对网络功能的使用效率造成影响,而采用不同的虚拟化方式,也将对后续网络功能的部署、故障迁移产生不同的影响。

本文针对 NFV 具体部署在 X86 平台上的问题进行研究,分析了通用服务器在部署 NFV 虚拟网络功能的架构组织,对数据转发平面进行部署和性能分析,对虚拟化方式进行性能测试,最后搭建大规模实验平台,比较网络功能虚拟化平台不同实现方式之间的差异。主要工作如下:

1. 介绍了 NFV 的相关基础知识,对 NFV 系统中涉及到了三个重要模块 NFVI、VNF 以及 VNF MANO 进行了详细阐述。阐明了 NFV 技术落地中影响 NFV 运行效率中的几个关键部分,包括数据面,虚拟化方式等相关知识,介绍了三种常用的数据面和两种常用的虚拟化方式。

2. 针对 NFV 部署中的数据转发平面,研究了 DPDK-OVS、NetMap 和 Click 这三种主流的数据转发平面,对其部署以及过程进行了详细的分析,对 DPDK-OVS 进行了整合及优化部署,对 NetMap 数据平面的数据结构进行优化,并搭建实验平面对三种数据平面的网络吞吐量、网络延时、以及对不同大小的数据包的转发能力进行测试以及比较分析,验证了针对 DPDK-OVS 以及 NetMap 的优化效果。

3. 针对宿主机中虚拟化方式的不同,阐述了 KVM 虚拟化方式和 Docker 容器技术的各自技术特点,对两种虚拟化方式进行了实际部署,介绍了相关虚拟化方式如何部署虚拟网络功能,并对容器部署中的网络问题进行了优化,搭建实验平台对两种虚拟化方式的性能指标进行测试和分析,重点比较和分析了 CPU 计算能力,内存访问以及读写能力,网络吞吐能力和重启时间长度等性能指标。

4. 结合前两章内容,搭建了大规模网络功能虚拟化实验平台,将不同的数据平面与虚拟化方式进行结合,验证网络功能虚拟化平台对外提供虚拟网络功能的

能力，并对两种网络功能虚拟化平台的网络性能和容灾性能进行了测试。

测试结果表明，本文进行的性能测试有效地体现了不同数据平面和不同虚拟化方式之间的性能差异，实行的优化方案提升了原有方案的性能，并对网络功能虚拟化平台如何实现高性能部署提供了支持，平台提供虚拟网络功能的功能和性能符合预期要求，为后续网络功能虚拟化技术高性能地在实际环境中大规模快速部署提供了理论和实际支持。

6.2 下一步工作展望

本文研究和部署了三种数据平面在网络功能虚拟化平台中的转发性能，比较了两种虚拟化方式之间的性能差异，并根据数据转发平面和虚拟化方式的不同搭建了两网络功能虚拟化平台，未来的工作可以从以下几方面展开：

1. 在性能比较中，尤其是采用 KVM 虚拟化技术时，更多加入对虚拟 I/O 性能的研究，完善性能测试。
2. 在多服务器上搭建服务器功能链，连接多个虚拟网络功能，比较在搭建服务功能链时网络功能虚拟化平台的性能。
3. 针对 OpenStack 虚拟机管理工具进行更多的研究，将不同的优化技术整合进 OpenStack 框架中，针对 Docker 容器技术进行研究，尝试编排容器和开发容器的实时迁移的功能。

致谢

岁月如梭，研究生三年的时光不知不觉即将结束，回想起在电子科技大学求学的七年时光，对这七年引导我、帮助我的人，心中充满了感激之情。

首先要感谢我的导师章小宁教授，您深厚的理论素养、严谨的治学态度都让我受益匪浅，在这三年中您不仅在我的学术工作中给予了耐心的指导，在生活中更是扮演一位家长的角色，让我学习到了很多书本上学不到的知识。在此，谨向章老师表达我由衷的感谢。感谢林蓉平老师在项目中的指导与教诲，您扎实的理论基础使我在科研中能更好地取得进步。

其次，我想感谢科 B318 的每一位同窗好友，在教研室的科研生活中，使我学习到了更多的技术，让我感受到实验室是一个大家庭，感谢何磊师兄、周泉师兄、刁造翔师兄和蒋亚杰师姐一直以来的关心和支持，使我迅速融入到教研室这个集体中。

感谢同门的沈少辉同学，王自豪同学，李毅超同学，宋雪同学，李卓峰同学，怀念三年中一起科研的时光，怀念每一次一起出游的快乐，怀念实习时候一起早出晚归的日子，是你们让我的研究生生活更加快乐。同门之谊，终身难忘。

感谢我的家人，是你们一直以来的支持和陪伴，让我在遥远的成都依旧可以感受家庭的关心。感谢我的女朋友，是你三年中的理解与陪伴，让我的生活有了更多的阳光与温暖。

最后，感谢各位参加论文评审和答辩的各位专家老师，感谢在百忙之中抽出来宝贵时间给予我的指导。

参考文献

- [1] 孙石峰, 罗成. NFV 管理和编排面临的挑战[J]. 邮电设计技术, 2016(9):68-73
- [2] J Wolkerstorfer, E Oswald, M Lamberger. An ASIC Implementation of the AES SBoxes[C] The Cryptographer's Track at the Rsa Conference on Topics in Cryptology. Springer-Verlag, 2002:67-78
- [3] L Seiler, D Carmean, E Sprangle, et al. Larrabee: A Many-Core x86 Architecture for Visual Computing[J]. IEEE Micro, 2009, 29(1):10-21
- [4] 陈炜, 韩小勇, 尼凌飞. 移动核心网应用 NFV 的关键问题探讨及实践[J]. 中兴通讯技术, 2014(3):12-15
- [5] P Lipp, S Santesson, M Mbarka, et al. ETSI TS 102 853: Electronic signatures and infrastructures (ESI); signature validation procedures and policies v1.1.1[J]. àrees temàtiques de la upc::informàtica::seguretat informàtica, 2012(5):36-41
- [6] 刘旭, 李侠宇, 朱浩. 5G 中的 SDN/NFV 和云计算[J]. 电信网技术, 2015(5):1-5
- [7] 李晨, 谷欣, 林群阳,等. NFV 网络加速技术探讨及实践[J]. 电信技术, 2016(6):25-29
- [8] 孙茜, 田霖, 周一青,等. 基于 NFV 与 SDN 的未来接入网虚拟化关键技术[J]. 信息通信技术, 2016(1):57-62
- [9] Tom Nolle, Who Will Orchestratc Orchestrators? [OL]:<http://blog.cimicorp.com/>
- [10] 杨健. 网络功能虚拟化系统测试技术研究[D]. 北京邮电大学, 2014
- [11] ETSI.European Telecommunications Standards Institute Industry SpecificationGroups (ISG),NFV.<http://www.etsi.org/technologies-clusters/technologies/nfv>, 2015,Accessed;June 03,2015
- [12] 唐宏, 欧亮. 网络功能虚拟化中的网络转发性能优化技术研究[J]. 电信科学, 2014, 30(11):135-139
- [13] W Wu, M Crawford, M Bowden. The performance analysis of linux networking - Packet receiving[M]. Elsevier Science Publishers B. V. 2007
- [14] N Y Koh, C Pu, S Bhatia, et al. Efficient Packet Processing in User-Level OSes: A Study of UML[J]. 2006:63-70
- [15] B Wickizer, A Silas, T Clements, et al. An analysis of Linux scalability to many cores[J]. 2010
- [16] P Emmerich, F Wohlfart, et al. Performance characteristics of virtual switching[C] IEEE, International Conference on Cloud NETWORKING. IEEE, 2014:120-125

-
- [17] V Kazempour, A Kamali, A. AASH Fedorova: an asymmetry-aware scheduler for hypervisors[C] ACM Sigplan/sigops International Conference on Virtual Execution Environments. ACM, 2010:85-96
- [18] A Vahdat, K Yocum, K Walsh, et al. Scalability and accuracy in a large-scale network emulator[J]. Acm Sigops Operating Systems Review, 2002, 36(SI):271-284
- [19] M Kamruzzaman, S Swanson, D M Tullsen. Inter-core prefetching for multicore processors using migrating helper threads[C] ACM, 2011:393-404
- [20] B Pfaff, J Pettit, T Koponen, et al. The design and implementation of open vSwitch[J]. ;login:: the magazine of USENIX & SAGE, 2015, 40:pages. 12-16
- [21] L Rizzo. Netmap: a novel framework for fast packet I/O[C] Usenix Conference on Technical Conference. USENIX Association, 2012:9-9
- [22] G Pongracz, L Molnar, Z L Kis. Removing Roadblocks from SDN: OpenFlow Software Switch Performance on Intel DPDK[C] Second European Workshop on Software Defined Networks. IEEE Computer Society, 2013:62-67
- [23] DPDK:Data Plane Development Kit <http://www.dpdk.org/>
- [24] Open vSwitch:An Open Virtual Switch. <http://openvswitch.org/>
- [25] L Rizzo, M Landi. netmap:memory mapped access to network devices[J]. Acm Sigcomm Computer Communication Review, 2011, 41(4):422-423
- [26] 王小龙. 模块化分布式路由器数据平面研究与实现[D]. 北京邮电大学, 2010
- [27] E Kohler, R Morris, B Chen, et al. The click modular router[J]. Acm Transactions on Computer Systems, 2000, 18(3):263-297
- [28] S Shirinbab, L Lundberg, D Ilie. Performance Comparison of KVM, VMware and XenServer using a Large Telecommunication Application[J]. 2014
- [29] A Kivity, Y Kamay, D Laor. KVM: The kernel-based virtual machine for Linux[J]. Proc Linux Symposium, 2010
- [30] D Bartholomew. QEMU: a multihost, multitarget emulator[M]. Belltown Media, 2006
- [31] LXC.[2016-06-26].<http://en.wikipedia.org/wiki/LXC>
- [32] E W Biederman. Multiple Instances of the Global Linux Namespaces[J]. 2006
- [33] G Qin, G Roy, D Crooks, et al. Cluster Optimisation using Cgroups at a Tier-2[C] 2016:012010
- [34] R Rosen. Linux containers and the future cloud[J]
- [35] 于建威, 李知杰, 赵健飞. 基于 OpenStack 的 Docker 应用[J]. 软件导刊, 2015, 14(9):46-48

- [36] C Boettiger. An introduction to Docker for reproducible research[J]. Acm Sigops Operating Systems Review, 2015, 49(1):71-79
- [37] C Nedelcu. Nginx HTTP Server - Second Edition[J]. 2013
- [38] M Roesch. Snort - Lightweight Intrusion Detection for Networks[J]. Proc.usenix System Administration Conf, 1999:229--238
- [39] 涂青云. 应急通信中应急调度分发模块的研究与设计[D]. 北京邮电大学, 2011
- [40] G A Covington, G Gibb, J W Lockwood, et al. A Packet Generator on the NetFPGA Platform[C] IEEE Symposium on Field Programmable Custom Computing Machines. IEEE Computer Society, 2009:235-238

攻硕期间取得的成果

硕士期间获奖情况

2017 年获研究生一等奖学金

攻读硕士学位期间的研究成果

- [1] Zhang X, Zhao Y, Guo S, et al. Performance-aware Energy-efficient Virtual Machine Placement in Cloud data center[C] IEEE International Conference on Communications. IEEE, 2017.