中国科学院大学

University of Chinese Academy of Sciences

硕士学位论文

基于 DPU 的服务网格架构设计与性能优化

作者姓名：李明

指导教师：鄢贵海 研究员

中国科学院计算与技术研究所

学位类别：工学硕士

学科专业：网络空间安全

培养单位：中国科学院计算与技术研究所

2023 年 6 月

The Architecture Design and Optimization of Service Mesh Based on DPU(Data Processing Unit)

A thesis submitted to

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Master of Science in Engineering

in Cyberspace Security

by

LI Ming

Supervisor: Professor YAN Guihai

Institute of Computing Technology, Chinese Academy of Sciences

June, 2023

中国科学院大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。承诺除文中已经注明引用的内容外，本论文不包含任何其他个人或集体享有著作权的研究成果，未在以往任何学位申请中全部或部分提交。对本论文所涉及的研究工作做出贡献的其他个人或集体，均已在文中以明确方式标明或致谢。本人完全意识到本声明的法律结果由本人承担。

作者签名：

日　　期：

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院大学有关收集、保存和使用学位论文的规定，即中国科学院大学有权按照学术研究公开原则和保护知识产权的原则，保留并向国家指定或中国科学院指定机构送交学位论文的电子版和印刷版文件，且电子版与印刷版内容应完全相同，允许该论文被检索、查阅和借阅，公布本学位论文的全部或部分内容，可以采用扫描、影印、缩印等复制手段以及其他法律许可的方式保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　导师签名：

日　　期：　　日　　期：

摘 要

服务网格 (Service Mesh) 是支撑云原生的关键技术之一, 其以专用中间层的方式实现了服务治理的相关功能。该中间层通常以代理 (Proxy) 的方式被部署在服务集群内，代理与服务通过网络进行通信。然而，在云原生场景中，代理部署方式存在两个重要问题。一是代理无法监管部署在网卡直连加速器上的服务，如通过在网卡端卸载 NVMe-oF 提供的存储服务，通过 GPUDirect 提供的 GPU 加速服务。二是代理的 I/O 密集或计算密集型操作占用了大量 CPU 资源导致系统性能下降。针对上述问题，本文分析了传统服务网格软件存在的性能瓶颈，并提出了一种基于 DPU 的服务网格架构设计。本文的贡献和工作如下：

1. 分析了服务网格存在的性能瓶颈

通过实验结果及建模分析，本文发现了四类导致当前主流服务网格性能瓶颈的因素：(1) 缺乏对部署在加速器上的服务进行感知和管理的支持；(2) 运行代理引入的多种 I/O 和计算密集的操作占用大量 CPU 资源；(3) 服务间的通信路径冗长使得通信延迟较高。(4) 各类远程过程调用（RPC）、函数调用、上下文切换和资源竞争等操作导致通信稳定性较差。实验结果表明，在主机侧 CPU 上实现服务网格难以满足云原生应用的进一步发展需求。

2. 提出了一种以数据为中心的服务网格架构设计

本文提出了一种基于 DPU 的服务网格架构 FlatProxy，该设计将服务网格卸载到网络入口处以保证对所有进出节点的流量进行监测和管理。在设计过程中，本文对架构进行了四个方面的优化：(1) 提出了一种软硬协同的设计方法以保证服务网格功能完备的同时实现数据处理加速。该方法在 DPU 的嵌入式 CPU 上实现了复杂的控制功能，利用专用硬件加速多种 I/O 密集和计算密集的数据处理。(2) 提出了一种分类处理架构以实现更高吞吐的数据处理。该架构使用流水线的方式处理低层网络协议及相关功能，使用众核架构处理高层网络协议及相关功能。(3) 提出了一种多层互联的硬件多线程模型以实现服务网格多样的数据处理功能和动态的数据流映射。该模型由一组独立的协议处理模块 PPM 组成, PPM 通过实时加载功能指令完成行为动态可变，不同 PPM 通过 NoC 互连完成动态的数据流映射。同时，位于不同网络层的 PPM 仅与相邻网络层的 PPM 通信以减少互联复杂度。(4) 进行了服务和 FlatProxy 通信的优化以提高系统整体性能。该优化利用 SR-IOV 以及套接字直连 (Socket Direct) 技术减少了主机侧网络协议栈的处理开销。

本文的设计与目前主流的服务网格代理 Envoy 相比，在不占用主机 (Host) 端 CPU 资源的情况下，将延迟降低了 80% 以上，吞吐量提高了 3 倍，请求响应能力提高 6 倍。

关键词：微服务，服务网格，DPU，软硬协同

Abstract

The service mesh is a crucial technology supporting cloud-native architecture, which enables service governance through a dedicated intermediate layer. Typically, this intermediate layer is deployed as a proxy within the service's Pod, communicating with the service through the network. However, there are two significant challenges in proxy deployment methods in cloud-native scenarios. Firstly, proxy cannot supervise services deployed on network-attached accelerators, such as hardware-accelerated NVMe over Fabric storage services or GPU acceleration services provided through GPUDirect. Secondly, the intensive I/O or computational operations of the proxy consume significant CPU resources, leading to system performance degradation. To solve the above challenges, we do two things as following.

1. Analyzing the performance bottlenecks in the service mesh

The performance bottlenetcks include: (1) the inability to perceive and manage services deployed on accelerators, (2) multiple I/O and computationally intensive operations consuming a significant amount of CPU resources, (3) lengthy communication paths resulting in high communication latency, (4) various operations such as remote procedure calls (RPCs), function calls, context switching, and resource competition leading to poor communication stability. The experimental results indicate that deploying service mesh on the host is difficult to meet the further development of cloud native applications.

2. Proposing a data-centric architecture of service mesh

This study proposes a DPU-based service mesh architecture called FlatProxy. The proposed architecture offloads the service mesh to the network entrance to monitor and manage all incoming and outgoing traffic. In designing FlatProxy, we optimized the architecture in four ways: (1) This study proposes a software-hardware collaborative design method to ensure the full functionality of the service mesh while achieving data processing acceleration. This method implements complex control functions on the embedded CPU of DPU and uses dedicated hardware to speed up I/O intensive and computational data processing. (2) This study proposes a classification processing architecture to achieve higher throughput data processing. This architecture uses a pipeline approach to handle low-level network protocols and a multi-core architecture to handle high-level network protocols. (3) This study proposes a multi-layer interconnected hardware multi-threading model to achieve diverse data processing functions and dynamic data flow mapping in the service mesh. This model comprises independent protocol processing modules (PPMs) that dynamically change their behavior through real-time loading of functional instructions. Different PPMs are interconnected through NoC to

achieve dynamic data flow mapping, and PPMs located in different network layers only communicate with PPMs in adjacent network layers to reduce interconnection complexity. (4) The optimization of service and FlatProxy communication is carried out to improve overall system performance. This optimization utilizes SR-IOV and socket direct technology to reduce the participation of the host-side network protocol stack.

Compared with the current mainstream service mesh proxy Envoy, FlatProxy reduces latency by over 80%, increases throughput by more than three times, and improves request response ability by more than six times while hardly consuming host CPU resources.

Key Words: Microservice, Service Mesh, DPU, Software and Hardware Co-design

目 录

第 1 章 引言 ····· 1
1.1 研究背景及意义 ····· 1
1.2 主要研究内容 ····· 2
1.3 主要创新点 ····· 4
1.4 本文组织结构 ····· 5
第 2 章 相关研究工作 ····· 7
2.1 云原生 ····· 7
2.1.1 云原生应用 ····· 7
2.1.2 云原生网络 ····· 8
2.1.3 基础设施 ····· 9
2.2 服务网格 ····· 9
2.3 边车代理 (Sidecar Proxy) ····· 10
2.3.1 功能模型 ····· 10
2.3.2 通信模型 ····· 11
2.3.3 主要特点 ····· 12
2.3.4 相关工作 ····· 12
2.4 DPU(Data Processing Unit) ····· 13
2.4.1 DPU 通用模型 ····· 13
2.4.2 DPU 主要功能介绍 ····· 14
2.4.3 DPU 主要特征 ····· 14
2.4.4 DPU 硬件相关研究 ····· 14
2.5 本章小结 ····· 15
第 3 章 服务网格性能分析 ····· 17
3.1 服务网格对服务通信的影响 ····· 17
3.1.1 通信延迟 ····· 17
3.1.2 数据吞吐 ····· 19
3.1.3 资源占用 ····· 21
3.1.4 主机侧实现服务网格的不足 ····· 21
3.2 卸载的可行性分析 ····· 22
3.2.1 服务网格 on-path 架构 ····· 23
3.2.2 架构的实现与评估 ····· 24
3.3 本章小结 ····· 25

第 4 章 FlatProxy: 基于 DPU 服务网格架构设计 ····· 27
4.1 FlatProxy 设计存在的难点 ····· 28
4.1.1 集中式服务网格高带宽的需求 ····· 28
4.1.2 服务网格多种数据处理功能的需求 ····· 28
4.1.3 多租户性能和安全隔离的需求 ····· 28
4.2 FlatProxy 的系统架构设计 ····· 29
4.2.1 软硬协同设计 ····· 30
4.2.2 分类处理架构 ····· 31
4.2.3 硬件多线程模型 ····· 32
4.2.4 多租户通信路径优化 ····· 35
4.3 基于 YUSUR K2-pro 原型验证平台的实现 ····· 36
4.3.1 系统总览 ····· 36
4.3.2 虚拟网络的实现 ····· 38
4.3.3 服务网格的实现 ····· 39
4.4 本章小结 ····· 41
第 5 章 实验设计与性能评估 ····· 43
5.1 实验环境及配置 ····· 43
5.2 微测试结果及分析 ····· 44
5.2.1 虚拟网络的测试 ····· 44
5.2.2 服务网格的测试 ····· 45
5.2.3 两类通信方式的测试 ····· 49
5.3 实验总结 ····· 51
第 6 章 工作总结与展望 ····· 53
参考文献 ····· 55
致谢 ····· 59
作者简历及攻读学位期间发表的学术论文与研究成果 ····· 61

图目录

图 1-1 异构平台服务网格代理问题 …… 2
图 1-2 异构平台以数据为中心的服务部署架构 …… 2
图 1-3 论文组织架构 …… 5
图 2-1 加速器调用方式。(a) 服务部署在 CPU 上 (b) 服务部署在加速器上 (c) 以数据为中心的服务部署 …… 8
图 2-2 虚拟网络模型。(a) 节点内通信 (b) 节点间通信 …… 8
图 2-3 服务网格模型。(a) 服务网格全景 (b) 部署服务网格的服务通信 … 10
图 2-4 代理数据处理流程 …… 11
图 2-5 代理与服务通信架构 …… 12
图 2-6 DPU 的模型。(a) 参考模型 (b)DPU 实例模型 …… 13
图 3-1 服务网格对服务通信延迟性能的影响。(a) 延迟随包大小变化 (b) 请求响应延迟 (c) 包传输延时抖动 …… 18
图 3-2 服务网格对服务通信吞吐性能的影响。(a) 吞吐随突发流量大小变化 (b) 吞吐随连接数的变化 (c) 吞吐随 CPU 核数的变化 (d) 响应速率随请求速率的变化 (e) 响应速率随连接数的变化 (f) 响应速率随核数的变化 (g) 包转发的性能 (最大包转发数和丢包率) …… 20
图 3-3 10Gbps 带宽条件下 CPU 使用率比较 …… 21
图 3-4 引入服务网格的服务通信路径 …… 22
图 3-5 边车代理 on-path 架构。(a) 数据通路 (b)L7 层功能的处理逻辑 (c)L3/L4 层处理逻辑 …… 23
图 3-6 硬件 on-path 架构的代理性能。(a)L3/L4 的延迟 (b)L3/L4 的吞吐 (c) 主机侧软件代理的请求响应分布 (d)DPU 侧硬件 on-path 架构代理的请求响应分布 …… 25
图 3-7 基于 DPU 的服务网格架构 …… 25
图 4-1 代理的部署方式对比。(a) 基于 CPU 的代理部署 (b) 基于 DPU 的代理部署 …… 27
图 4-2 FlatProxy 的顶层设计。(a)FlatProxy 的模型 (b)PPM 的模型 …… 29
图 4-3 软硬协同设计 …… 31
图 4-4 分类处理架构 …… 32
图 4-5 多层互连硬件多线程模型 …… 33
图 4-6 服务与 FlatProxy 通信优化。(a) 松耦合通信 (b) 紧耦合通信 …… 35
图 4-7 基于 K2-Pro 原型验证平台的服务网格系统架构 …… 37
图 4-8 基于 DPU 的虚拟网络系统架构 …… 39

图 4-9 基于 DPU 的 TCP 代理系统架构 ····· 40
图 5-1 软件 VXLAN 与硬件加速 VXLAN 的比较。(a) 延迟随包大小变化 (b) 请求响应延迟 (c) 吞吐随突发流量大小的变化 (d) 请求响应速率 ·· 44
图 5-2 不同优化策略的通信延迟比较。(a) 延迟随包大小变化 (b) 请求响应平均延迟 (c) 请求响应 99% 延迟 (d) 包传输延时抖动 ····· 45
图 5-3 不同优化策略服务通信吞吐性能的比较。(a) 吞吐随突发流量大小变化 (b) 吞吐随连接数的变化 (c) 吞吐随 CPU 核数的变化 (d) 响应速率随请求速率的变化 (e) 响应速率随连接数的变化 (f) 响应速率随核数的变化 (g) 包转发的丢包情况（Envoy 的测试与 sockmap 测试结果重合） ····· 47
图 5-4 CPU 资源使用率随突发流量大小的变化情况 ····· 48
图 5-5 两种服务与 FlatProxy 通信方式的性能比较。(a) 不同比例包的请求延时 (b) 响应延时随连接的变化 (c) 请求响应速率随连接数的变化 ··· 50

表目录

表 2-1 虚拟网络实现方式特征比较 ····· 9
表 2-2 服务网格功能列表。(a) 传输层的功能 (b) 应用层的功能 ····· 11
表 3-1 使用 envoy 实现 TCP 代理延时分析 ····· 18
表 3-2 使用 envoy 实现 HTTP 代理延时分析 ····· 19
表 3-3 硬件平台特性比较 ····· 23
表 5-1 FlatProxy 和 envoy 实现 TCP 代理延时分析比较 ····· 46
表 5-2 FlatProxy 和 envoy 实现 HTTP 代理延时分析比较 ····· 48

符号列表

缩写

CPU	Central Processing Unit
NIC	Network Interface Card
GPU	Graphic Processing Unit
TPU	Tensor Processing Unit
FPGA	Field Programmable Gate Array
NVMe	Non-Volatile Memory Express
I/O	Input/Output
DPU	Data Processing Unit
NoC	Network on Chip
PPM	Protocol Processing Module
SRIOV	Single Root I/O Virtualization
IOMMU	Input/Output Memory Management Unit
ASIC	Application Specific Integrated Circuit)
AI	Artificial Intelligence
PCIe	Peripheral Component Interconnect Express
ARM	Advanced RISC Machine
RISC-V	Ruduced Instruction Set Computer V
IP	Internet Protocol Address
eBPF	extened Berkeley Packet Filter
MAC	Media Access Control Address
NAT	Network Address Translation
TCP	Transmission Control Protocol
VPP	Vector Packet Processing
DSA	Domain Specific Architecture
VLAN	Virtual Local Area Network
VXLAN	Visual eXtensible Local Area Network
TC	Traffic Control
NP	Network Processor
HTTP	HyperText Transfer Protocol
UDP	User Datagram Protocol

NUMA	Non-Uniform Memory Access
VM	Virtual Machine
URL	Uniform Resource Locator
RTC	Run to Complete
RMT	Reconfigurable Match Table
DMA	Direct Memory Access
gRPC	Google Remote Procedure Call
GSO	Generic Segmentation Offload
GRO	Generic Receive Offload
DPDK	Date planedevelopment kit
VF	Virtual Function
CNI	Container Network Interface
VDPA	Virtio Data Path Acceleration
TOE	TCP Offload Engine
OVS	Open Virtual Switch
QoS	Quality of Service

第 1 章　引言

1.1　研究背景及意义

为应对日益复杂的业务需求，越来越多的企业正在采用云计算和云原生技术构建应用程序[1]。云原生应用使得企业可以以更加敏捷的方式快速开发、部署和管理应用程序。然而，随着服务规模越来越大，各服务的配合协调问题变得越来越突出，例如，在分布式环境下，服务请求和响应的可靠和可用性难以保证。解决上述问题将可以极大地提高云原生应用的可维护性和稳定性。因此，本文聚焦于云原生环境下微服务服务通信管理方面的问题。

微服务架构是应用的一种架构组织方式，其以分布式通信的方式联合各分立的功能组件（称为服务）实现完整的应用需求[2–5]，可有效简化应用开发的复杂度。然而，当服务规模逐渐增大后，服务间的配合协作问题将变得越来越突出。为实现大规模服务的稳定运行目前主要有以下三种措施：一是通过将服务注册到控制中心然后由控制中心将服务信息下发到各个节点实现全局的服务发现。二是实现负载均衡、熔断限流、流量统计、连接跟踪等流控和观测技术以保证服务通信质量。三是实现身份验证，数据加密，安全隔离等措施以保证多租户场景下安全性要求。上述的解决方法组成了服务治理的核心内容，是大规模服务稳定运行的关键[6–8]。

然而，传统依托服务治理框架和调用库的服务开发模式使得服务变得复杂，加大了服务开发难度。同时，紧耦合的服务开发方式受制于框架本身的能力，使得服务规模难以进一步扩大。为解决上述问题，业界采用了一种松耦合服务治理方式即服务网格，将业务服务与服务治理功能完全分隔开。服务网格通过在每个服务的 Pod 内部署一个代理软件接管所有出入服务的流量，实现对服务间通信的管理，观测和安全检查等服务治理功能[1,9–11]。服务网格的优势在于代理与业务服务的网络通信使服务治理对服务透明，能够极大的简化服务的开发部署难度，增强系统整体的可扩展性。然而，当服务被部署在加速器或存储设备上通过网卡与外界进行数据通信时，传统部署在主机侧 CPU 上的代理无法感知该通信过程，从而难以监管此类服务。

图1-1示意了一种典型的异构平台的代理结构，主要由四部分组成：网卡、与网卡直连的存储、与网卡直连的加速器和主机侧 CPU。在该示意图中有三种服务，在主机侧 CPU 运行的服务 A，在存储侧的服务 B，以及在加速器上运行的服务 C。基于这三类服务，有三种数据流分别是网络到存储的数据流 x、网络到加速器的数据流 y、网络到主机侧 CPU 的数据流 z，其中数据流 x，y 并不在代理 Proxy 的管理范围以内。随着越来越多服务被部署在与网卡直连的加速器、存储高性能硬件平台[12–14] 上，这些脱离了服务网格统一管理的数据通信将增加系统管理的复杂度。同时，由于数据量的急速增加，服务网格 I/O 密集和计算密集型操作将占据大量 CPU 资源，并导致系统性能下降[15]。本文实验结果表明，在原

有的微服务体系中引入服务网格将导致延迟增长 2 倍左右，吞吐下降高达 40%，CPU 资源占用增长 2 倍以上。

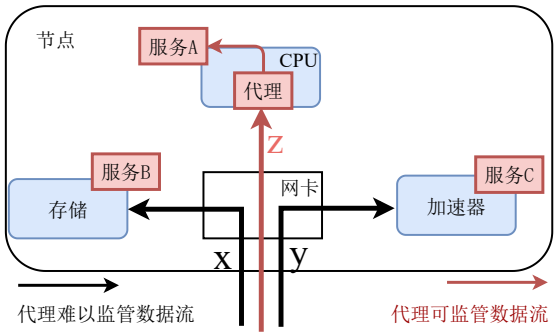


图 1-1 异构平台服务网格代理问题

Figure 1-1 The dilemma of proxy on heterogeneous platform

1.2 主要研究内容

在数据中心服务部署中，为了提高服务密度和性能[16]，服务通常会部署在 CPU、FPGA，GPU 等多种异构的硬件平台上，而不是统一部署在主机侧的 CPU 中，如图 1-2 所示。然而，传统部署在主机侧的服务网格缺乏对部署在加速器上的服务进行监管的能力。因此，为适应新型服务部署方式，本文进行了新的服务网格架构的探索。

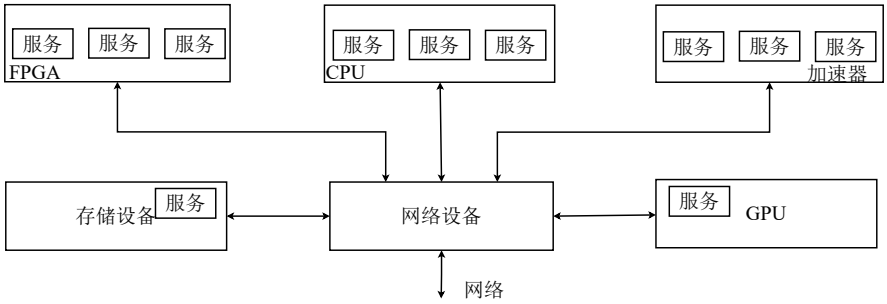


图 1-2 异构平台以数据为中心的服务部署架构

Figure 1-2 The architecture of data-centric services deployment on heterogeneous hardware

由于在各种架构中，网卡都是数据出入服务的关键点。因此，做为网络入口的网卡是实现云原生服务治理的理想位置。在网口进行流量管理可以最大程度的避免数据的不必要传输，及早将数据进行分发的同时完成对流量的控制，观测和检查。基于上述思想，本文进行了以下两方面的工作。

1. 传统服务网格性能瓶颈分析

本文基于业界广泛使用的服务网格代理软件 Envoy 进行性能分析，并结合多种开源测试工具，从延时，吞吐，响应速度，抖动等多个维度对当前服务网格实现方式进行了评估。根据实验结果分析，服务网格的性能瓶颈主要体现在以下几个方面：

- 服务网格缺乏对部署在网络直连加速器上的服务进行监管的能力。网卡直连加速器的架构能够旁路（Bypass）CPU 为数据传输提供更高的性能，因此越来越多的服务被部署在此类架构的加速器上。然而，传统部署在 CPU 侧的服务网格代理通过网络监听的方式缺乏对旁路 CPU 通信的感知，从而导致无法监管被直接部署在加速器上的服务。
- 服务网格代理运行时的多种 I/O 和计算密集的操作占用大量 CPU 资源。服务网格代理做为流量的出入口涉及大量的 I/O 操作，同时网络处理中涉及的各类操作如加密、解密、序列化、反序列化等都是计算密集型操作。本文实验结果表明，在代理执行简单路由的情况下 CPU 资源使用超过 70%，是服务使用 CPU 资源的 2 倍多。
- 通信路径冗长使得通信延迟较高。代理与服务通过网络通信的方式导致一次单边服务通信包括六次网络堆栈处理、两次 vSwitch 转发、十多次数据复制以及各类系统调用。上述通信过程造成服务通信延迟过高，本文实验结果表明部署代理的服务通信延迟是不部署代理的 2 倍以上。
- 函数调用，上下文切换和资源调度等软件操作导致通信稳定性较差。部署代理的通信过程造成数据传输在内核空间和用户空间频繁切换，造成大量系统调用和软件开销。这些开销不仅降低了系统整体性能，也导致数据传输稳定性差。本文实验表明，部署代理后服务通信延时抖动是无代理情况的 2-3 倍。

基于本文多方面的性能分析可以看出在主机侧部署服务网格不仅无法满足服务新型部署架构的要求，而且占用大量 CPU 资源导致系统性能下降严重。上述不足导致服务网格难以满足各类高性能应用上云的需求，限制了云原生进一步的发展。

2. 基于 DPU 的服务网格架构设计

针对当前基于 CPU 的服务网格在架构和性能方面存在的不足。本文提出了一种以“数据”为中心的服务网格架构，即将服务网格卸载到网络入口处实现。该架构首先保证了对所有进出节点流量的监管，避免了当前服务网格难以感知部署在网络直连加速器上服务的缺点。其次，在“网卡”侧实现服务网格可以优化原本冗长的数据通路，避免容器、主机和网卡的多次数据拷贝。另外，“网卡”侧也更容易利用一些专用的硬件进行数据加速，从而满足越来越高的带宽需求。最后是代理卸载到网卡侧实现了与主机侧的物理隔离，可以有效减少主机侧的资源消耗。为了实现该架构，需要“网卡”具备智能路由以及强大的数据处理能力。因此，本文从互连能力，可编程能力，可控制能力以及数据处理加速能力四个方面对现有的相关硬件平台和技术进行了考察。类似 GPUDirect 这类网络直连的技术往往仅针对特定的应用场景难以适应云原生多类型流量的传输，也不能承载服务网格对流量的各类复杂操作[17]。智能网卡的相关研究重点还是集中在数据处理的加速功能，难以完成各类不同加速器与网络的直连[18]。DPU 作为智能网卡的升级和革新弥补了它的不足，提供了强大的控制能力和设备互连能力，是本文实现服务网格新架构探索的首选硬件平台。

由于 DPU 在服务网格中的应用还处于早期，相关研究较少，本文从如下三个关键问题入手开展研究，具体如下：

- 如何满足集中式服务网格的高带宽需求，内容介绍如下：当服务网格被卸载到 DPU 上，其不仅需要承担所有进出节点流量的监测和管理工作，还需要承担同节点不同服务之间的数据通信流量管控。这使得其在理论上需要处理的带宽峰值超过网口带宽，对其处理能力产生巨大挑战。
- 如何实现多样化数据处理功能的动态可扩展，内容介绍如下：服务网格不仅具备各类已有功能而且存在许多自定义和正在发展的功能。保证服务网格卸载后仍具有功能的可扩展性是其能否在市场获得广泛关注的重点之一。另外，服务网格在进行数据处理时，不同的功能可以根据数据类型进行动态组合。此类灵活性的需求与硬件加速存在一定的矛盾也是服务网格卸载必须考虑的难点之一。
- 如何保证多租户条件下性能和安全的隔离，内容介绍如下：多租户场景下，不同租户服务通信的性能和安全隔离是提高服务质量的必然要求。当服务网格卸载后，需要从软硬件两方面采取相应的措施满足不同租户通信的性能和安全性要求。

针对上述难点，本文综合考虑了服务网格在功能灵活性和性能两方面的需求，基于 DPU 的特点提出了多种优化方式，具体内容见下一小节。

1.3 主要创新点

为实现基于 DPU 的服务网格，本文进行了四方面的优化。

- **提出了一种软硬协同的设计方法** 根据服务网格各类功能特点的不同，本文通过软硬协同的方式分别进行了优化，包括基于 DPU 侧嵌入式处理器实现的完整代理，和对部分 I/O 密集和计算密集操作的硬件加速。
- **提出了一种分类处理的架构** 为兼顾管理进出节点的流量和节点内服务通信流量，本文针对网络协议层级的不同分别采用了流水线和众核的优化措施，在满足灵活性的同时提升了数据处理带宽。其中，低层协议（L2、L3 和 L4）格式固定，涉及的服务治理功能较为简单，可使用可编程流水线的架构在满足功能需求的同时尽可能提高数据处理的吞吐。应用层的协议格式多变，功能多样，需要硬件具备更多的灵活性。因此，本文采用众核的架构去处理该层的相关功能。另外，对于一些计算密集型的任务则需要专用的硬件进行加速处理，如加解密，压缩，解压缩，序列化和反序列化等。
- **提出了一种多层互连多线程模型** 为应对服务网格多样的功能，以及动态的数据流映射需求，本文提出了一种多层互连的硬件多线程模型。该模型由一组独立的协议处理模块 PPM 组成，PPM 按照实时加载的功能指令实现相应的数据处理，相互间通过 NoC 形成动态的数据通路。同时，位于不同网络层的 PPM 仅与相邻网络层的 PPM 通信以减少互连复杂度。
- **进行了服务和代理通信的优化** 针对多租户场景下的性能与安全问题，本文利用 SR-IOV[19] 以及套接字直连 (Socket Direct) 技术[20,21] 消除了服务和代理

通信的网络协议栈处理，从而降低宿主机对服务通信的性能干扰。同时，硬件的 SR-IOV 能力通过 IOMMU 能够保证不同用户之间的安全隔离，而 DPU 在不断迭代发展中针对安全所提供的各类框架将进一步提高系统整体的安全性。

1.4　本文组织结构

本文总共包括六个章节如图 1-3所示，后续章节及相关内容安排如下。

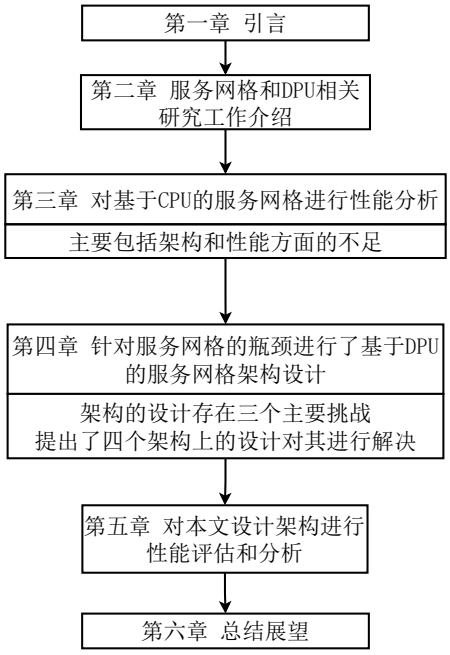


图 1-3 论文组织架构

Figure 1-3 Thesis structure

第二章介绍了服务网格和 DPU 的相关概念。首先，该章节从网络，应用和基础设施三个层面介绍了云原生的概貌。其次介绍了云原生的关键技术服务网格和它的数据面（边车代理）实现的功能，部署方式和通信架构等内容。最后，介绍了新型硬件 DPU 的通用模型和主要功能。

第三章分析了当前主流的服务网格代理 Envoy 的性能瓶颈。服务网格瓶颈和不足主要包括当前实现架构难以监测部署在网络直连加速器上的服务，代理运行占用过多的 CPU 资源对系统整体的性能产生影响。另外，在该章节分析了服务网格实现在不同硬件平台上的可行性，并得出初步结论。

第四章提出了基于 DPU 的服务网格架构设计和性能优化。该架构天然解决本文在第三章分析发现的服务网格架构和性能方面面临的问题，但在实际的设计过程中本文发现这种新架构存在三个重要的技术挑战，即高吞吐的挑战、多种功能动态组合的挑战以及多租户场景对性能和安全隔离的挑战。针对新架构存在的上述技术难点本文提出了相应的解决方案。设计方案总体采用软硬协同的设计方法，使用分类处理的架构以满足高吞吐的需求，使用多层互连多线程的硬件模型以满足功能多样数据流动态映射的需求，通过服务和代理通信深度优化

以满足性能和安全性的要求。最后，该章节介绍了基于驭数 DPU K2-Pro 的虚拟网络和服务网格实现方案。

第五章针对本文提出的新架构 FlatProxy 进行了原型验证。该章节分别测试了基于 DPU 的虚拟网络和服务网格 FlatProxy 的性能，并与几种典型服务网格优化方式进行了比较。最后，该章节对两类服务与代理通信的优化方式进行了性能测试和比较。

第六章对本文工作进行了总结以及对未来工作进行了展望。

第 2 章　相关研究工作

针对微服务通信管理在云原生演进中存在的不足，本文探究了服务网格如何在新的服务组织架构中提供更高效的服务治理能力。该研究涉及的相关领域包括云原生、服务网格以及 DPU 三个部分。其中，云原生既是大的发展背景，也是应用新的组织形式的体现；服务网格是维持应用服务运行的关键技术；DPU 则是本文用以实现服务网格的硬件平台。接下来，本章将针对以上三方面的发展现状及相关研究进行详细介绍。

2.1　云原生

云原生是云计算发展历程中一种应用开发、部署、运维的新范式[1,22]。本节将从应用、网络和基础设施三个层面对云原生服务的通信管理相关内容进行介绍。

2.1.1　云原生应用

云原生应用以微服务的形式进行开发和部署，实现了云计算按需、弹性、共享以及可测量的需求，但业务规模的不断增长以及高性能应用的部署对服务运行的性能提出挑战。为应对该挑战，将计算密集型负载通过专用硬件（如 GPU、VPU、TPU 等）进行加速被视为一条可行的路径。基于此类异构平台，部署在 CPU 端的服务调用加速器时需要在主机侧和加速器端进行多次数据传递，造成系统性能下降，如图2-1(a)所示。对此，已有的优化选择将特定逻辑部署在加速器或 SmartNIC 上，以规避 CPU 的过度参与，从而提高整体性能[13,14]，如图2-1(b)所示。

基于微服务部署在加速器的想法，一些研究提出了“以数据为中心”的应用部署方式，即无需主机侧 CPU 干预。常见优化方法包括：（1）全卸载，即无 CPU 的设备直连，英伟达公司通过 GPUDirect 实现了 GPU 与 NIC 的直连[17]，NVMe-oF 的卸载实现了存储设备与 NIC 的直连[23]。（2）部分卸载，如将部分应用逻辑卸载到 SmartNIC 上进行加速，而 CPU 端仍需进行控制信息的处理[24] 或在 SmartNIC 上部署简单的函数式服务[14]。除此之外，许多基于异构平台的应用运行方式的探索也为云原生提供了新的发展机遇，如 Lynx 展示了如何构建以数据为中心的服务架构以提高性能[13]，XPU-Shim 展示了如何在异构平台上构建无服务架构（Serverless）以提高服务部署的密度[16]。本文分析总结了异构平台云原生应用的各种部署方式，如图 2-1 所示。图2-1(c)展示了一种理想的以数据为中心的服务部署架构。其中，各类服务根据其需求被部署在不同的硬件平台上而无需主机侧 CPU 进行数据中转控制。

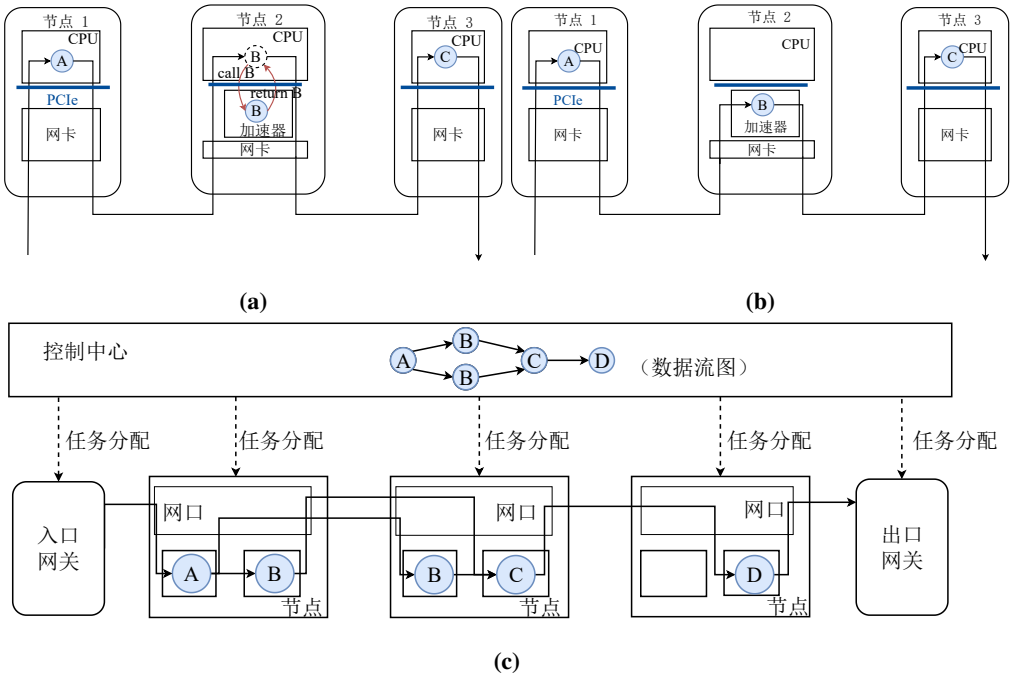


图 2-1 加速器调用方式。(a) 服务部署在 CPU 上 (b) 服务部署在加速器上 (c) 以数据为中心的服务部署

Figure 2-1 The calling procedure of accelerators. (a) The CPU-centric calling (b) The data-centric calling (c) The service deployment in the data-centric architecture

2.1.2 云原生网络

云原生服务按需启用和动态伸缩的特点使得服务运行的真实物理节点也需动态变化，这导致了服务的网络地址无法直接用于真实网络通信。因此，云原生多采用虚拟网络的方式进行服务通信。

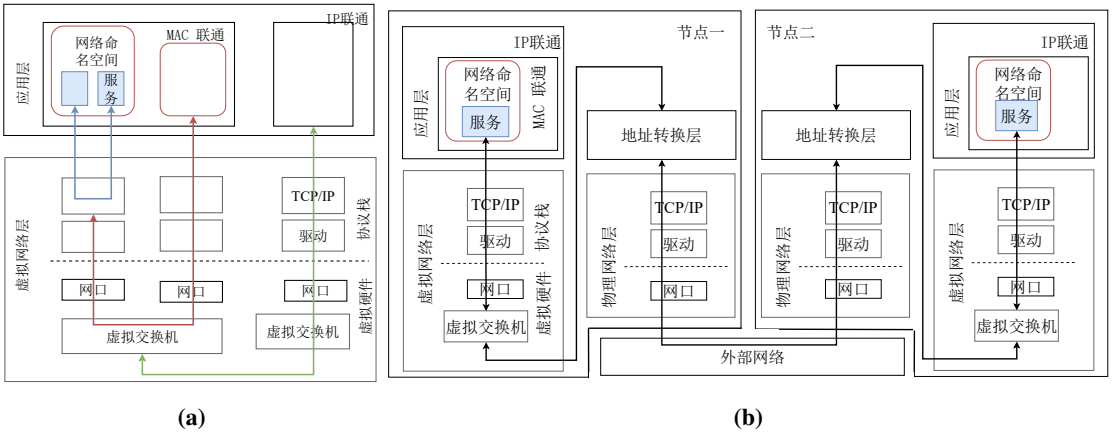


图 2-2 虚拟网络模型。(a) 节点内通信 (b) 节点间通信

Figure 2-2 The model of virtual network. (a)The inner-node communication (b)The inter-node communication

虚拟网络整体分为覆盖网络（overlay network）、虚实网络地址转换以及底层网络（underlay network）。其中，应用直接感知的覆盖网络是运行在真实网络之

上的虚拟的网络，它的逻辑网络拓扑与传统网络一致，但在网络收发节点的网络协议处理存在差异，具体与服务运行的虚拟环境相关，如虚拟机采用独有网络协议栈，容器复用主机协议栈，但一般服务的网络空间均独立拥有完整的网络处理逻辑。对于虚拟网络的硬件，其依赖于宿主机，通常包括二层的虚拟交换设备和三层的虚拟路由设备等。基于上述组件，同宿主机内的不同虚拟机中的应用进行通信的路径如图2-2(a)所示，不同宿主机间的应用依赖物理网络和地址转换进行通信的路径如图2-2(b)所示。

覆盖网络的地址转换需要借助数据包的地址标识信息（如 IP，端口、MAC 等），转换方式依据实现方式不同可分为三类，即隧道封装，Host-GW 和 NAT。隧道封装技术将覆盖网络的数据包封装在真实网络的报文中，虚实地址的映射关系通常是应用全局可知；Host-GW 虚拟地址可被作为真实地址使用，但依赖于物理网络的路由能力；NAT（Network Address Translation）技术，地址转换只发生在源地址一侧，映射关系也只被宿主机感知，但需保证目的地址信息与底层网络地址信息一致的要求。

表 2-1 虚拟网络实现方式特征比较

Table 2-1 The comparison of characteristics of Virtual Network Implementation Methods

实现类型	物理地址与虚拟地址的关系	路由对虚拟网络是否感知	是否修改虚地址
隧道封装	不相同	无感知	不修改
Host-GW	相同	感知	不修改
NAT	不相同	无感知	修改

第三部分底层网络即为真实的数据传输网络，遵循传统网络的地址识别以及路由规则。底层网络不受覆盖网络的影响，但底层网络能够支持的通信方式及通信限制将影响覆盖网络地址与真实网络地址的转换方式。

2.1.3　基础设施

为部署百万量级服务规模的大型应用，仅仅依靠人工方式很难进行有效管理，因此一系列的云计算基础设施被设计提出以确保服务的正常稳定运行。本文重点关注服务通信管理相关的内容，其涉及的主题包括如何定位动态部署的服务、如何进行服务流量的监测和管理、如何保证通信的稳定性，容错能力等等。相关内容经历多次迭代发展，逐步由最初与业务应用紧耦合的方式发展为松耦合独立的中间件方式。当前，服务通信管理最通用的方式之一是服务网格，本文将在下一小节对相关内容进行介绍。

2.2　服务网格

当大规模服务被部署在云端，首要的任务是找到服务在哪里，也就是服务发现。当前，服务发现通常通过分布式信息同步机制实现，具体由单个控制中心

负责收集服务的路由、流量控制、监测要求等信息，然后通知其他服务。在服务发现的基础上，关键问题转变为如何保证通信的健壮性，解决方法包括负载平衡、流量控制、断路等多种流量控制方法[6,7]。此外，随着系统复杂性的不断增高，保持网络的可观察性以准确定位通信问题也变得越来越重要。公有云对于安全性也存在严格的要求。上述需求共同构成了服务治理的关键内容。

为了降低服务规模增加引起的应用开发复杂度，当前服务治理实现多采用服务网格的方式。它通过一个与服务完全解耦的专用中间层实现服务治理的功能[25]，如图2-3(a)所示。该中间层通常包括一组部署在各个节点的代理软件和分发配置信息的控制中心构成。部署服务网格的服务通过边车代理（Sidecar Proxy）与外界通信，如图2-3(b)所示，下一小节将详细介绍。

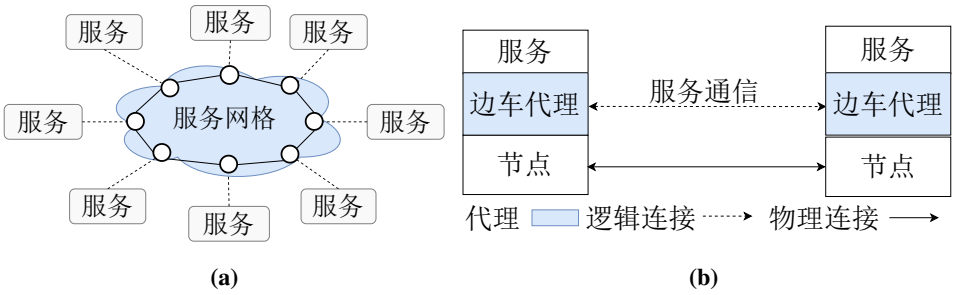


图 2-3 服务网格模型。(a) 服务网格全景 (b) 部署服务网格的服务通信

Figure 2-3 The model of service mesh. (a)The landscape of service mesh (b)The communication of service with sevice mesh.

2.3 边车代理 (Sidecar Proxy)

Pod 是 Kubernetes 中创建和管理的、最小的可部署的计算单元。Pod 内包括一组容器，它们能够共享存储、网络以及怎样运行容器的声明[26]。边车代理即是与服务部署在同一 Pod 内的代理软件，两者运行在不同的容器中。边车代理接管所有进出服务的流量，并实现服务治理的大部分功能，完成了服务网格的主要数据处理需求[9,27,28]。本小节从功能和通信两方面对代理进行介绍。

2.3.1 功能模型

边车代理功能主要分为配置、控制和数据处理功能。其中，配置功能根据服务网格控制中心分发的配置信息设置数据处理或控制规则；控制和数据处理功能包括流量控制、安全检查和网络监控；流量控制是边车代理最基本的功能，它包括表2-2中列出的各种功能。

根据网络通信层次可以对功能进行如下划分，主要包括以包（packet）为处理单位的低层（L3/L4) 功能和以消息（message）为处理单位的高层（L7）功能。每一层次的功能均分为控制和数据处理，其中控制功能通常以连接/会话为管理对象，涉及到硬件难以实现的多个连接/会话状态管理；数据处理功能涉及每个

数据包和消息是 I/O 密集型或计算密集型操作，如路由、加解密等等。在当前主流的服务网格实现方式中，代理的功能根据流的标识信息动态的构建功能链，然后进行相应的处理。

表 2-2 服务网格功能列表。(a) 传输层的功能 (b) 应用层的功能

Table 2-2 The feature list of service mesh. (a) These features in the L3/L4 (b) These features in the application layer

a

网络层次	控制功能	数据处理功能
L3/L4	接收处理新的连接	解析元数据
	事件限流	隧道封装/解封装
	熔断	路由
	连接数限制	负载均衡
	连接跟踪	过滤

b

网络层次	控制功能	数据处理功能
L7	重试	解析协议
	路由表管理	重定向
	熔断	协议升级
	限流	路由、缓存
		负载均衡

边车代理数据处理包括三个部分，协议解析、功能处理和数据包封装。图2-4中显示了一个典型的数据处理流程。数据被解析为元数据和有效载荷，然后解析结果被发送到根据元数据动态组合的函数/过滤器链。处理后的数据作为原始数据被发送到下一个网络层，直到数据在每个层中被解析和处理。最后，每个网络层封装处理后的数据和新的元数据。

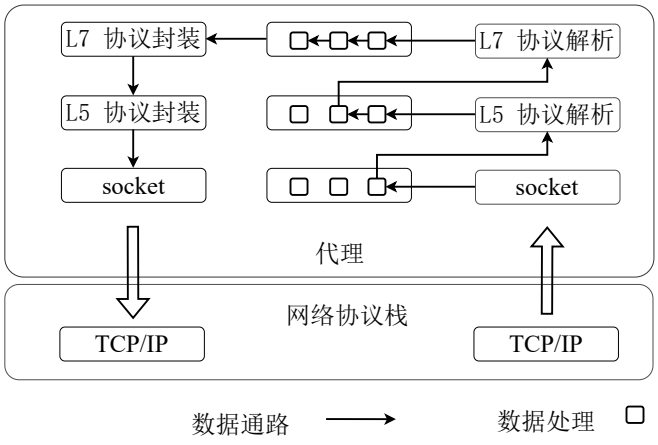


图 2-4 代理数据处理流程

Figure 2-4 The process of proxy processing data.

2.3.2　通信模型

图 2-5中所示的通信过程包括 Pod 内的通信和 Pod 之间的通信。Pod 内的通信位于同一个网络命名空间，依次通过了 TCP/IP 协议栈、环回接口。Pod 之间的通信是不同网络命名空间的数据通信，需要通过 TCP/IP 协议栈、虚拟交换机或网卡。当一个服务向不同节点中的另一个服务发送数据时，通信路径包括六次

网络堆栈处理、两次虚拟交换机转发和十多次数据复制。在实际系统中，可能还存在虚拟网络到物理网络的转换，以及虚拟路由器的转发等。

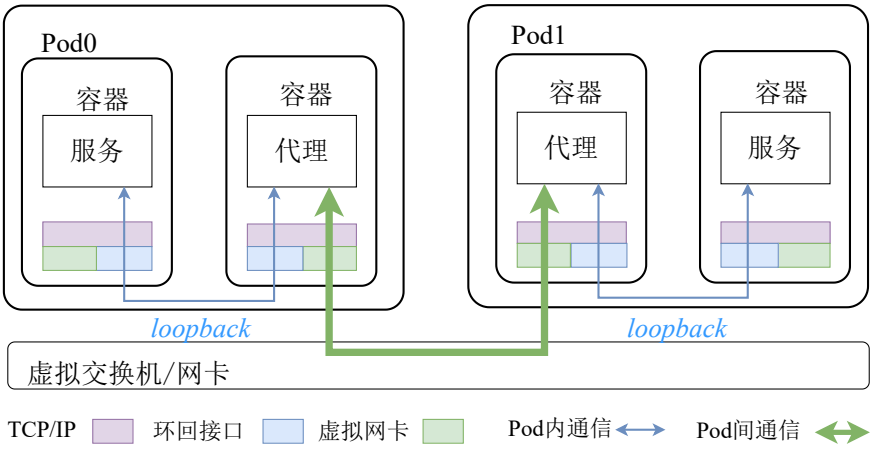


图 2-5 代理与服务通信架构

Figure 2-5 The communication architecture between proxy and services.

2.3.3 主要特点

边车代理的主要特征如下：（1）功能多样。边车代理实现的功能涉及整个网络协议栈，包括流量处理，网络监测，安全检查等多个方面。同时，这些功能包括各种类型，如配置、控制和数据处理。（2）软件定义数据处理。边车代理通过软件定义实现数据处理，数据处理流程可以根据配置信息或传递的数据内容实时进行动态组合。（3）透明运行。边车代理独立于服务，服务对代理运行不可知。

2.3.4 相关工作

当前服务网格是构建云原生最重要的基础设施之一，其提供了强大的服务管理能力，简化了应用的开发和部署，但其存在的性能问题也成为阻碍企业大规模部署服务网格的重要原因。为提高服务网格的性能，工业界和学术界从以下三方面对服务网格的相关能力进行了优化。

为减轻服务网格引入导致的性能损失，最简单直接地方式是进行通信路径优化。如图2-5所示，可以看出在整个通信路径中，包含多次数据拷贝，网络处理以及数据转发。因此，现今的解决方案多采用零拷贝的技术以减少内核态和用户态的数据拷贝。举例来说，在服务网格的优化技术中，开源项目 Cilium 利用 eBPF 技术开发了内核态服务网格功能减少了上下文切换。同时，eBPF 也可以通过旁路 TCP/IP 协议栈实现 POD 内服务和代理 socket 通信以减少网络处理带来的开销[28]。另一类优化通信过程的方式是实现共享边车代理并且与原先的流量网关进行融合[29]，该优化方式能够减少 50% 左右的资源开销，qps 提高 80%[30]。但共享代理的方式使得流量处理被集中起来，给计算密集型的操作带来巨大挑战。为解决该问题，业界采用了硬件加速的方式，如阿里云在 EC2 产品中提供了 TLS 的硬件加速能力[31]。

当前的协议优化常用方式是使用用户态协议栈去减少数据拷贝和上下文切换带来地开销。可以使用 VPP 部署用户态高性能协议栈提高数据传输能力，减少内存拷贝[32,33]。此外，还存在一些基于不同优化目标的用户态协议栈能够为服务网格提供更高效的通信能力，如 Fstack 提供的低延迟网络处理[34]。

针对代理的优化主要是云供应商进行的代理定制化开发，如阿里云实现的 SOFAMesh 和华为云实现的 Mesher，它们通过优化路由、调度和定制协议等提高性能[35,36]。其中阿里云使用 Envoy 进行的定制化开发中发现其细粒度的统计操作导致了严重的内存开销和性能损失，故其开发了可控的统计操作在不需要的时候可以关闭其监测功能，SOFAMesh 是阿里开发的针对网络遥测进行性能优化的服务网格开发项目。学术界目前针对服务网格优化研究还比较少，但对于服务网格所具有的关键功能，如负载均衡，网络遥测，安全检查则存在大量研究工作。

2.4　DPU(Data Processing Unit)

DPU 是一种新型的数据处理单元。在本节中介绍了 DPU 通用模型，其定义了 DPU 的主要功能。同时，本小节也展示了一个典型的硬件实例来说明当前的 DPU 所具有的能力。

2.4.1　DPU 通用模型

如图 2-6(a)所示，DPU 包括五个部分，系统 I/O、网络 I/O、控制平面、数据平面和内存。系统 I/O 是 DPU 与 CPU、FPGA 和 GPU 等其他处理单元之间的通信接口。网络 I/O 包括高速网络接口，实现 DPU 与网络之间的通信。控制平面的功能包括（1）通过监视设备状态来管理所有资源；（2）控制数据平面运行；（3）将计算任务分配给硬件。数据平面以与网口相同的线速度处理数据，通常包括特定领域加速器和具有可编程能力的组件。存储器用于存储中间结果、设备状态和配置信息[37–41]。

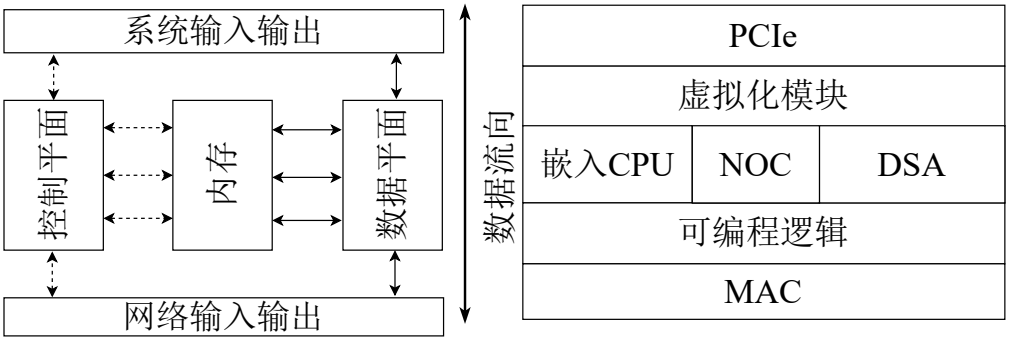


图 2-6 DPU 的模型。(a) 参考模型 (b)DPU 实例模型

Figure 2-6 The model of DPU. (a) The reference model (b)The instance model

2.4.2 DPU 主要功能介绍

图 2-6(b)所示的硬件实例包括高性能 PCIe 和网络接口、虚拟化模块、可编程逻辑、一组 DSA、嵌入式 CPU 和 NoC[41]。

- **虚拟化模块（SR-IOV）** DPU 通过 SR-IOV 实现高性能 I/O 虚拟化[19]。VM 或容器可以通过 SR-IOV 绕过主机内核直接访问设备，从而使虚拟 I/O 设备性能接近物理设备。此外，虚拟化可以通过 IOMMU 实现服务通信的隔离。
- **DSA（网络处理引擎）** DPU 实现各种加速器以加速网络处理。例如，DPU 提供 TOE 来加速 TCP 处理。它还加速了其他计算密集型操作，如加密和解密。
- **可编程逻辑（vSwitch&NP）** DPU 实现可编程逻辑，如 vSwitch 和 NP。VSwitch 支持 L4 以下的基本路由和高级功能，如 VLAN、VXLAN、NAT 和 TC。NP（网络处理器）可以实现更复杂的功能，如 HTTP 解析、L7 路由和负载平衡。
- **通用处理器** 与其他加速器不同，DPU 集成了通用处理器，如嵌入式 CPU，对实现控制功能至关重要。
- **其他功能** DPU 为应用程序提供了加速器，并为应用程序安全和监控构建了完整的框架。例如，它支持“零信任”安全架构。它也可以通过统计、日志和跟踪连接来观察数据流。

2.4.3 DPU 主要特征

基于 DPU 进行应用程序卸载和实现的特点和优势包括：（1）**强大的控制能力**。例如，硬件资源管理、计算任务分配和构建数据路径等操作，增强了硬件的可适用性。（2）**灵活的数据处理能力**。可编程硬件和灵活的互连可以在灵活性和性能方面进行平衡。（3）**智能互联能力**。各类接口提供了加速器，新型存储，网络，主机相互之间的直连能力，为不同的场景提供了高效的通信路径。同时，可编程逻辑可通过指令或编程的形式实现软件定义的数据路径，实现动态的数据传输路径选择。

2.4.4 DPU 硬件相关研究

硬件加速是解决云数据中心性能问题的重要方法，DPU 为数据中心的数据处理提供了新的解决方案。自 2017 年 AWS 提出 Nitro[42] 以来，SmartNIC 凭借其出色的数据处理和可编程能力，在学术界和工业界取得了重大进展[24,43,44]。随后，自 SmartNIC 发展而来的 DPU，IPU，CIPU 提供了更强大的网络、存储和计算能力，同时提供了支持云环境的虚拟化能力为云原生基础设施的发展提供了强力的硬件基础[37–41]。另外针对云原生的发展，谷歌于 2022 年 7 月份发布了第一款基于 ARM 架构的云原生处理器 TAU，T2A[45]；阿里在 2022 年 11 月 3 日展示了基于 ARM 的云原生倚天 710 芯片，提供了云原生应用的硬件加速能力。上述硬件的出现为服务网格等基础设施提供了潜在的发展机会。在硬件发展的基础上，服务网格相关的基础设施也获得了极大的发展。目前，RedHat 在 BF2 上实现了 OVN 卸载，其重点是 L7 层以下网络功能的加速[46]。另外一些研究探索

了在 SmartNIC 上的各类网络协议低延迟实现，为上层更复杂的功能加速提供了基础支撑[47–51]。

2.5　本章小结

本章介绍了主要背景知识，包括云原生，服务网格和 DPU。云原生是本文主题的背景，在2.1小节介绍了云原生应用，网络和基础设施的相关内容。服务网格是本文主要的研究内容，在2.2小节概述了服务网格的相关概念，2.3小节详细介绍了服务网格核心组件边车代理的功能，通信架构等内容。DPU 是本文实现服务网格架构设计的主要硬件平台，在2.4小节介绍了 DPU 的通用模型和相关实现。

第 3 章　服务网格性能分析

本章基于当前主流服务网格数据面实现软件 Envoy，分析了其在性能以及资源消耗两个方面的特征，为本文基于 DPU 的服务网格架构设计进行前期探索。服务网格将边车代理与服务部署在同一 Pod 内，二者位于同一个网络空间通过回环的方式进行通信。同时，由代理负责与外界通信，接管所有进出服务的流量。依据服务网格的此类运行和部署方式，本文探讨的问题主要包含，（1）当前服务网格在性能和资源使用方面存在什么问题，如何影响服务的正常通信？本文在3.1小节从多个角度对服务网格带来的影响进行了测试。（2）将代理功能卸载到 DPU 上是否可行？本文在3.2.2小节通过实现一种 on-path 的服务网格架构对该问题进行了验证。

3.1　服务网格对服务通信的影响

基于服务网格的功能模型和通信模型，可以清楚的看出当前的服务网格架构中还存在许多待优化的部分。先前对服务网格的优化工作主要集中在软件的部署方式，此类优化可以有效的缓解服务网格带来的性能和资源开销，但在 I/O 数据持续增加的情况下 CPU 处于超载运行状态，单纯优化软件难以满足数据中心的数据处理性能需求[52]。因此，本文提出了软硬协同，纵向多层整合的系统优化策略。基于此，本小节对典型的服务网格通信方式进行了性能分析，从通信延迟，数据吞吐，资源使用三个方面分析了服务网格对服务通信的影响。本小节实验对象是不部署服务网格和部署服务网格两种情况。为简化分析过程，服务网格的代理仅开启路由功能。

3.1.1　通信延迟

在端到端延时测试中，本文分别从绝对延时和稳定性两个角度进行了测试。其中稳定性主要体现在延时抖动和长尾延时两个方面。如图 3-1(a)所示，在服务网格仅执行简单路由的情况下部署服务网格的 L3/L4 数据包传输延时达到 240us 是未部署服务网格传输延时的 2 倍以上。同时，如图 3-1(b)所示，L7 的请求响应延时远高于不使用服务网格的情况，平均延时可达 5-6 倍。通过对延时占比的详细分析，如表3-1所示，在数据包从 vSwitch 到应用服务的传输过程中，由 Envoy 引入的延时占比超过 60%，其中大部分时间都耗费在数据的拷贝和通信过程中。实际上，如果需要代理对数据进行更复杂的操作则绝对延时将更长。表3-2进一步展示了 L7 的延时占比情况，可以看出高层协议代理相关处理操作延时占比将更高。

上述的分析主要关注的是一些延时占比较高的操作，而其他的一些如上下文切换，函数调用，数据拷贝等等的软件操作则会造成传输延时的抖动。如图3-1(c)所示，在部署代理后包的传输延时抖动是无代理数据传输的 2 倍以上，这种

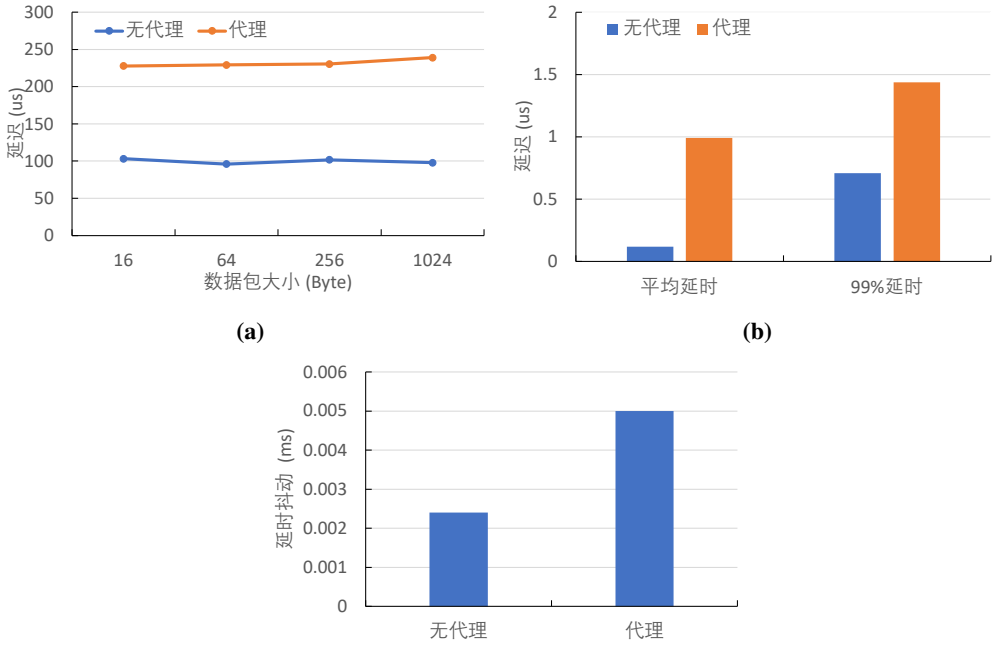


图 3-1 服务网格对服务通信延迟性能的影响。(a) 延迟随包大小变化 (b) 请求响应延迟 (c) 包传输延时抖动

Figure 3-1 The impact of service mesh for latency of service communication. (a)The latency over varying the packet size (b)The respond latency of request (c)The jitter of packet forwarding latecy

表 3-1 使用 envoy 实现 TCP 代理延时分析

Table 3-1 The latency breakdown of TCP proxy using envoy.

组件	延时占比	具体操作
vSwitch	9%	数据转发
TCP/IP stack	27%	数据包解析和处理
Proxy 相关的操作:		
TCP->Proxy	20%	数据拷贝
Proxy	7%	数据处理
Proxy->TCP	10%	数据拷贝
loopback	27%	proxy 与应用的通信

不稳定性可能导致部分应用运行故障。在应用层，图3-1(b)中展示了部署代理的数据通信长尾延时大约是无代理的 2 倍左右，该结果对于实时性要求较高的业务将产生不良影响。

表 3-2 使用 envoy 实现 HTTP 代理延时分析

Table 3-2 The latency breakdown of HTTP proxy using envoy.

组件	延时占比	具体操作
vSwitch	3%	数据转发
TCP/IP stack	8%	数据包解析和处理
Proxy 相关的操作:		
connection& statistical	26%	建立连接以及统计信息的处理
D-T	16%	数据传输
D-I,D-P	12%	数据结构初始化和解析
D-CK,D-OP,D-M	22%	数据检查，处理，修改
D-DP	6%	数据按协议格式封装
loopback	7%	proxy 与应用通信

3.1.2　数据吞吐

在数据传输吞吐的测试中，本文从 TCP 吞吐，HTTP 请求响应速率和 UDP 转发性能三个方面进行了分析。其中，图3-2(a)显示了 TCP 吞吐随流量大小的变化情况，图中显示在流量大于 64MB 后不使用代理的吞吐高于使用代理的情况，最高可有 2 倍以上的性能差距。对于前期流量小于 4MB 时，代理的吞吐高于无代理的情况主要与测试工具的计算方法以及代理与同 Pod 的应用传输延时较短有关，并不能代表实际的数据传输能力。图3-2(b)展示了 TCP 连接对吞吐的影响，可以看出不部署代理的情况下吞吐很快就达到峰值接近 25Gbps(网口的速率为 25Gbps), 而部署代理的情况下吞吐量随 TCP 连接数逐步增加，在建立 16 条连接时达到峰值约为 20Gbps。之后，64 连接情况下吞吐下降可能原因是在 NUMA 架构下，测试负载和代理位于不同 CPU 将导致内存访问瓶颈或不同连接之间产生资源竞争。由于该内容非本文关注的重点，本文在此不做深入讨论，后续工作将会进一步分析具体原因。图3-2(c)展示了 CPU 核对吞吐的影响，首先总体而言不部署代理的情况下单核的性能已经足够处理 25Gbps 的带宽而部署代理后吞吐在 8 核左右才能达到峰值且峰值小于不部署代理的情况。同时，在部署代理的情况下，对代理进行核约束将比仅对应用进行核约束造成的影响更大。在 8 核之前，对代理进行核约束的吞吐都远低于不对代理进行核约束的情况。最后是当本文将代理使用的核与测试负载使用的核完全绑定在一起后，峰值大于两者核隔离的情况，这里本文分析认为多核情况下，随机的进行核隔离会导致内存访问瓶颈（NUMA 机制的原因）。

如图3-2(d)、3-2(e)和3-2(f)所示，对于应用层的请求响应测试与 TCP 吞吐具有相同的规律。实验结果表明，不论是在多连接还是多核的情况下，部署代理后

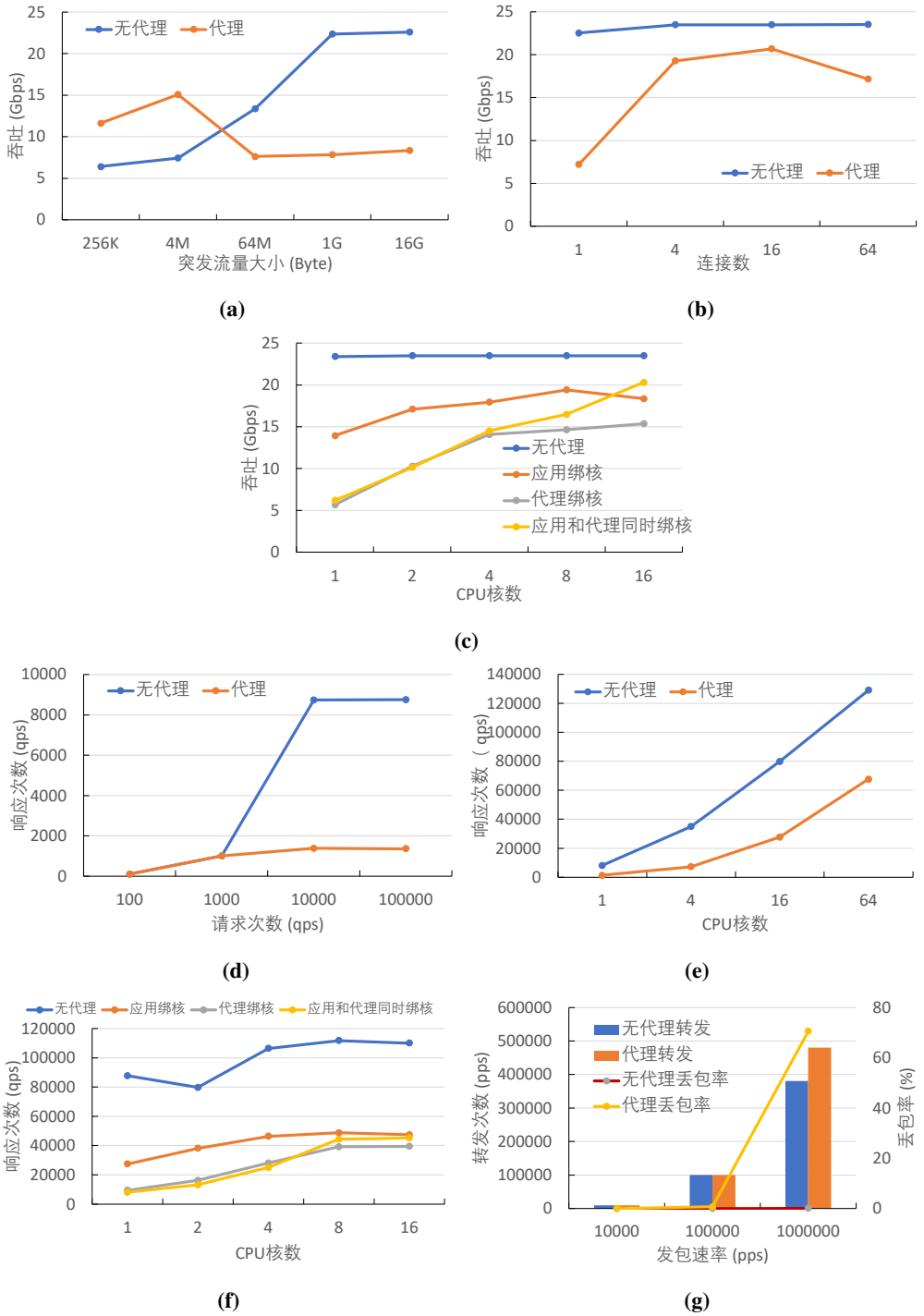


图 3-2 服务网格对服务通信吞吐性能的影响。(a) 吞吐随突发流量大小变化 (b) 吞吐随连接数的变化 (c) 吞吐随 CPU 核数的变化 (d) 响应速率随请求速率的变化 (e) 响应速率随连接数的变化 (f) 响应速率随核数的变化 (g) 包转发的性能 (最大包转发数和丢包率)

Figure 3-2 The impact of service mesh for throughput of service communication. (a)Throughput over varying traffic size (b)Throughput over varying connection numbers (c)Throughput over varying core numbers (d)Resond rate over varying request rate (e)Respond rate over varying http connections (f)Respond rate over varying core numbers (g)Performance of packet forwarding.

服务的通信能力都存在较为严重的下降。在 UDP 转发能力测试中，图3-2(g)展示了在 1000000pps 的压测条件下，尽管部署代理的 UDP 包转发速度高于不部署代理的情况但丢包率达到了 71%, 显然是由于代理对 UDP 处理能力不足导致。

3.1.3　资源占用

本小节主要关注的是对 CPU 资源的使用情况。本文使用测试负载按照 10Gbps 的速度发送数据包。从图3-3可以看出部署代理后 CPU 的占比高达 103%（使用资源超过一个 CPU 核），此时不部署代理的 CPU 使用率仅为 15%。在实际的生产环境中，根据各家厂商公布的数据显示云基础设施的 CPU 占比高于 30%[53]，极大的增加了应用上云的成本。

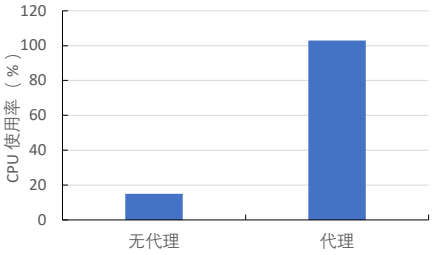


图 3-3 10Gbps 带宽条件下 CPU 使用率比较

Figure 3-3 The comparison of CPU utilization at 10Gbps bandwidth.

3.1.4　主机侧实现服务网格的不足

通过调研和实验分析表明主机侧实现服务网格存在诸多不足之处。最主要的体现在以下三个部分：

传统服务网格难以监管部署在网卡直连加速器上的服务　传统服务网格代理部署在主机侧，其通过网络监听的方式感知服务流量并进行流量截获。当服务部署在加速器上旁路 CPU 直接通过网卡与外界通信时，代理网络监听无法感知此类通信过程，从而难以对那些部署在加速器上的服务进行监管。

代理运行时的 I/O 密集和计算密集的操作占用大量 CPU 资源　边车代理截获了所有进出服务的流量涉及大量的 I/O 操作，同时其承担的服务通信管理的功能，如安全验证涉及的加解密操作和 RPC 调用涉及的序列化和反序列化操作，包含很多计算密集型的任务。上述任务消耗大量的 CPU，不仅影响业务服务运行，而且在 CPU 性能提升缓慢的大背景下，越来越高的 I/O 带宽将不可避免的导致 CPU 过载，从而无法满足应用服务的性能需求。

更高的延迟和更低的吞吐　根据本文分析，如图3-4所示，传统服务网格的服务通信过程多次经过虚拟机、主机和 NIC，并在用户态和内核态频繁切换，带来了冗长的数据路径和各类软件开销。一次单边的服务通信包括六次网络堆栈处理、两次 vSwitch 转发，十多次数据复制以及其他各类系统调用。该通信过程将显著降低系统整体的性能，根据实验结果显示通信延迟增长了 2 倍以上，吞吐降低 2-3 倍，请求响应能力最高降低 6 倍以上。

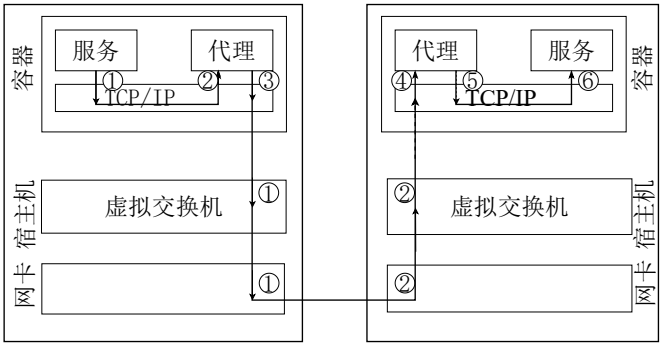


图 3-4 引入服务网格的服务通信路径

Figure 3-4 The communication path of service with service mesh.

各类系统开销导致数据传输不稳定 代理的功能涉及多个方面，如流量控制，安全检查，网络遥测，这些功能并不是完全相互依赖的，但目前的代理在运行过程中主要是基于串行的方式，这就导致功能之间相互阻塞。同时，上下文切换，数据拷贝，函数调用等等操作也导致了大量的系统开销，其不仅带来了严重的性能损失同时严重干扰了服务通信的稳定性，从实验结果可以看出引入代理导致数据传输的抖动增长了 2 倍以上。

3.2 卸载的可行性分析

通过服务网格的性能分析可以看出基于 CPU 实现的服务网格尽管能够完成服务治理的相关功能，但其存在性能，资源占用等方面的问题。同时，其对运行在加速器上的服务无法监管成为服务网格进一步发展的限制条件。因此，本文提出了以“数据”为中心的服务网格架构设计。为充分适应异构平台的云原生应用发展，释放硬件潜能，新的架构不仅需要提供相关的数据处理加速能力，还需要提供智能互连的能力以支持正在发展的新的服务部署架构。基于此，本文考察了当前一些研究中提供的硬件解决方案以寻求一个最适合服务网格实现的硬件平台。

- **近网计算** 本文主要关注节点处近网计算的硬件架构，该架构实现了旁路 CPU 的网络和设备直连，如 NIC 和存储设备的直连[23],NIC 和加速器的直连[17]。另外，一些研究将数据处理转移到 NIC，从而消除数据网络和主机侧的数据传输[13,14,24]。上述两种方式遵循了类似的优化理念，将数据处理放在数据源附近，可以有效地提高系统性能。然而，此类技术仅针对一些特殊的体系结构和场景，不适合更广泛的使用。
- **SmartNIC** SmartNIC 主要的目的是用于加速计算密集型的网络任务，如 AWS 的 Nitro[42]，阿里的神龙[54] 和其他一些加速研究[24,43,44]。尽管存在一些研究针对特定场景使用 SmartNIC 用于控制功能的卸载，但大部分采取的方法是由主机侧进行控制，此类方式对于控制类的功能存在一定的延迟，同时无法避免对主机侧应用的性能干扰，也不利于业务应用与基础设施的安全隔离。

• **DPU**　DPU 是智能网卡的进一步发展。它不仅加速了数据处理，而且提供了强力的控制能力。同时，根据 DPU 的相关调研可以看出，它具备连接加速器、主机 CPU、网络和其他设备的能力，这为各类不同类型的设备直连提供了途径[37–41]。另外，DPU 提供的可编程能力也可以提供更智能的流量处理和流量分发方式。

表 3-3 硬件平台特性比较

Table 3-3 The comparison of characteristics of hardware

硬件类型	网络加速功能	互连能力	控制能力	可编程能力
近网计算	无	弱	无	部分有
SmartNIC	有	无	部分有	有
DPU	有	强	有	有

通过上述的调研分析，如表3-3所示各类硬件方案在不同方面存在不足和劣势。综合考虑，本文提出了基于新型硬件单元 DPU 构建服务网格的构想。在下一小节中，本文利用 DPU 上的通用处理单元尝试实现服务网格，探讨该方案的可行性。

3.2.1　服务网格 on-path 架构

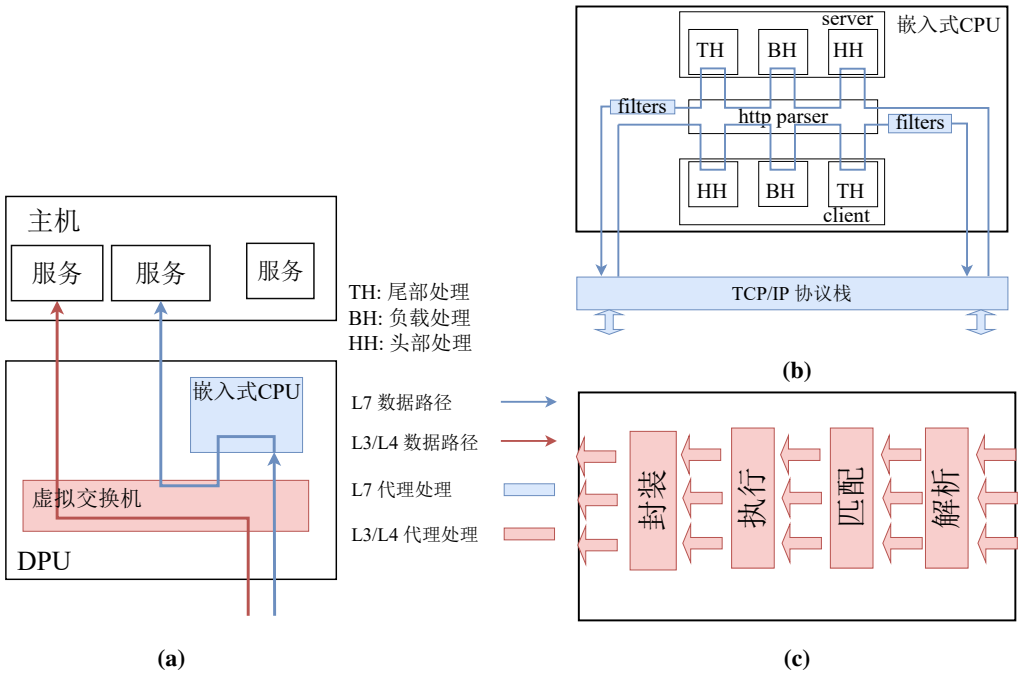


图 3-5 边车代理 on-path 架构。(a) 数据通路 (b)L7 层功能的处理逻辑 (c)L3/L4 层处理逻辑

Figure 3-5 The on-path architecture of sidecar proxy. (a)The data path (b)L7 process logic (c)L3/L4 process logic

在初步的基于 DPU 的服务网格架构探究中，本小节主要关注 DPU 是否足够支撑相应的功能。如图3-5所示，该设计利用基本的虚拟交换机单元实现一部

分 L3/L4 的服务治理功能，而应用层的高级功能只有在低层协议解析后组成消息（message）才能进行相关的操作，最简单的方式就是在 DPU 的嵌入式处理器上利用软件以 on-path 的架构实现。如图3-5(a)所示，当流量到达 DPU 时，如果流量需要由 L7 中的功能处理，它将通过虚拟交换机单元转发到嵌入式的处理器上由代理进行相关的操作。代理处理完数据后通过交换单元发送到服务。但如果流量需要进行的操作能够在交换单元上完成处理，它将直接发送到服务。通过上述方式，本文可以在 DPU 上实现服务治理的相关功能，并获得一定的加速能力。图3-5(b)显示了，高层（L7）协议 HTTP 通过软件实现进行数据处理的过程，该过程串行的完成协议解析和相关的各种操作。图3-5(c)显示了，低层（L3/L4）协议流水线的处理架构。

3.2.2 架构的实现与评估

在利用 DPU 部署好代理后，本文从低层协议处理和高层协议处理两个层面进行了性能评估，结果如图3-6所示。从图3-6(a)和3-6(b)中可以看出，对于利用交换设备实现的路由功能其在延迟和吞吐两方面相比原本边车代理的方式都有较为明显的提升。而对于使用嵌入式 CPU 实现的高层协议处理功能，从图3-6(c)和3-6(d)可以看出卸载后反而造成性能的下降。根据实验结果及3.1小节主机侧代理的性能分析可知，性能下降的原因包括三点：（1）服务网格代理运行需要占用大量 CPU, 而嵌入式 CPU 没有足够的数据处理能力，很容易发生过载的现象；（2）基于 CPU 的代理执行过程为串行执行，容易造成数据阻塞从而导致系统整体性能下降；（3）嵌入式 CPU 上执行代理与主机上执行代理类似，具有各类系统调用，如上下文切换、数据拷贝、资源竞争等等，这些操作都会导致系统性能下降和传输过程的不稳定。

基于上述初步的服务网格卸载探索可以看出 on-path 的架构无法支撑代理对性能的要求，故一种利用硬件加速的软硬协同的架构成为本文后续工作主要的探究目标。如图3-7所示，本文提出初步的设计构想是使用 DPU 完成数控分离将数据处理的任务都卸载到硬件进行加速处理，控制功能则利用 DPU 上的嵌入式 CPU 实现以减轻硬件设计负担。

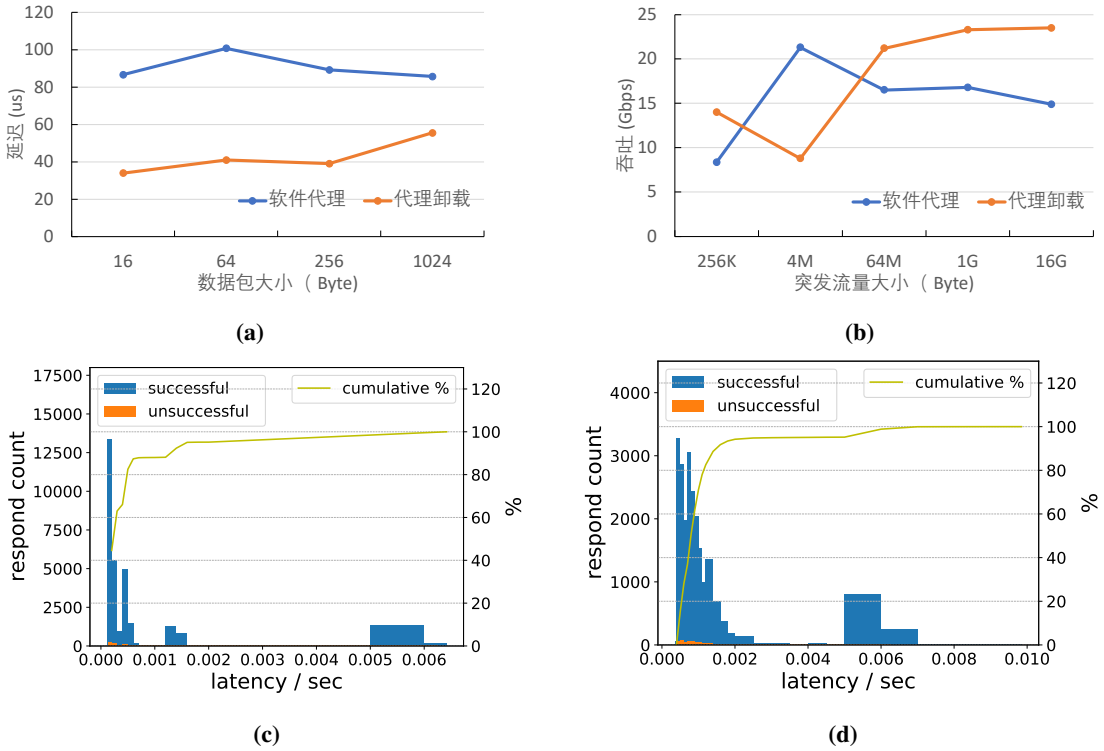


图 3-6 硬件 on-path 架构的代理性能。(a)L3/L4 的延迟 (b)L3/L4 的吞吐 (c) 主机侧软件代理的请求响应分布 (d)DPU 侧硬件 on-path 架构代理的请求响应分布

Figure 3-6 The performance of proxy implemented in on-path architecture. (a)the latency in the L3/L4 (b)The throughput in the L3/L4 (c)The distribution of respond queries using software proxy on the host (d)The distribution of respond queries using on-path architecture on the DPU

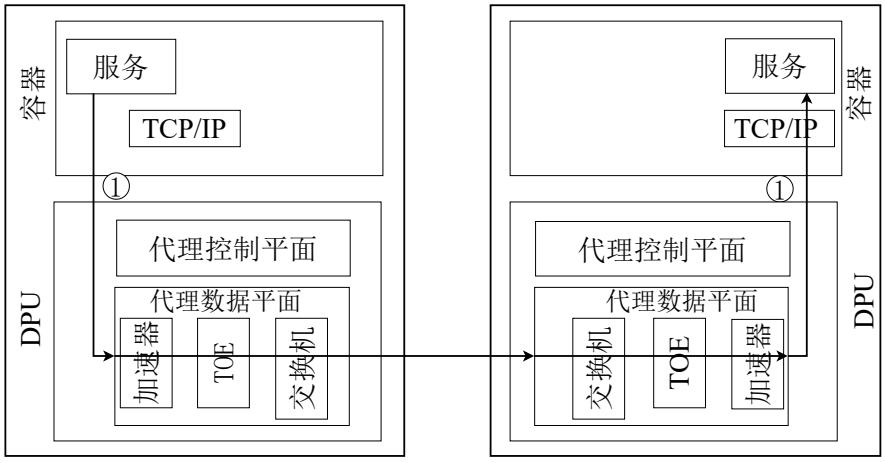


图 3-7 基于 DPU 的服务网格架构

Figure 3-7 The architecture of service mesh based on DPU.

3.3　本章小结

本章在3.1小节进行了服务网格的性能分析，从通信延迟，数据吞吐和资源使用率三个方面评估了基于 Envoy 代理部署的服务网格性能。通过对比分析，总结

出服务网格存在三方面的不足 (1) 难以监管无主机侧 CPU 参与的服务通信；(2) 各类 I/O 操作和计算密集型操作占用大量 CPU 资源；(3) 通信路径冗长使得通信延迟较高，同时各类函数调用，上下文切换和资源争用等软件相关操作导致通信稳定性较差。基于性能分析，本章在3.2小节进行了初步的基于 DPU 的 on-path 服务网格方案探索。实验结果表明 on-path 架构难以支撑服务网格对性能的需求，需要进一步探究更高性能的软硬协同方案。

第 4 章　FlatProxy: 基于 DPU 服务网格架构设计

基于异构平台的服务被直接部署在加速器或网卡上，消除了主机侧 CPU 的参与，提高了系统整体性能[13,24,52]。然而，这类服务部署模式给当前基于服务网格的服务通信管理带来了巨大挑战。如图4-1(a)所示，基于 CPU 的服务网格实现和优化要求数据流必须通过主机侧的代理，这导致其难以监管在网卡和加速器之间直通的数据流量。一个有效的解决方案是基于“网卡”的近网络服务网格实现，即将服务网格代理迁移到网卡上以提供对直连数据通信的感知和监测，如图4-1(b)中所示。

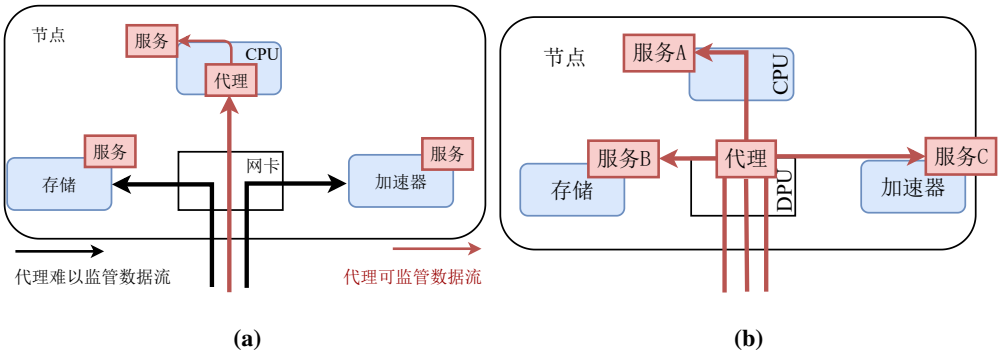


图 4-1 代理的部署方式对比。(a) 基于 CPU 的代理部署 (b) 基于 DPU 的代理部署

Figure 4-1 The deployment of proxy. (a) CPU-based (b) DPU-based

作为近网络服务网格，它不仅需要实现许多关于服务治理的数据处理任务，还需要根据流量标识符，如 IP、Port、URL、Cookier 等，将流量分配到不同的服务。前者需要强大的数据处理能力，因为服务治理存在多种计算密集以及 I/O 密集的操作，这些操作在流量突发的情况下很容易造成 CPU 过载运行。后者则要求智能的互连能力。因此，一个符合上述两个条件的硬件平台成为实现服务网格新架构的关键。设备直连的架构[12,17,23] 通常仅适用于特殊场景的数据传输不利于服务网格灵活多变的功能实现。智能网卡虽然提供了强大的加速功能，但缺乏互连和控制能力[24,42–44,54]。本文综合考虑硬件加速能力、智能互连能力和控制能力三方面，选择了 DPU 做为重构服务网格的硬件基座。为此，本文设计了一种基于 DPU 的以“数据”为中心的服务网格架构 FlatProxy。FlatProxy 能够准确监控所有进出节点的流量，适用于部署在加速器上通过网卡与外界通信的服务通信管理。同时，该架构结合 DPU 提供的加速单元能够有效缓解主机侧数据处理的压力，提高系统性能和安全性。然而，基于 DPU 的服务网格架构设计存在几个主要的难点，高带宽需求，多种数据处理功能动态组合的需求，多租户性能和安全隔离的需求。接下来，本章将围绕存在的难点和相应的解决方法介绍本文主要工作。

4.1 FlatProxy 设计存在的难点

尽管当前的 DPU 提供了诸多硬件加速，连接，控制功能，但作为一个服务网格的实现载体本文仍需根据应用的特征合理的使用已有的部件以及提出本文认为更合理的硬件架构方式。在 DPU 上构建服务网格，存在的难点包括，如何满足集中式服务网格的性能需求、如何实现服务网格多样数据处理功能动态可扩展的需求以及如何保证多租户条件下 FlatProxy 和服务之间通信共享后性能和安全的隔离。

4.1.1 集中式服务网格高带宽的需求

与传统服务网格代理相比，部署在 DPU 中的服务网格不仅需要接管所有进出节点的流量，还需要接管节点内不同服务之间的通信流量，更高的数据带宽成为必须解决的问题。此外，各类计算密集型数据处理也对提高 FlatProxy 数据带宽带来了巨大挑战。为了提高卸载后代理的数据处理能力，本文设计了一种分类处理架构以应对不同的数据类型。该架构采用流水线结构来处理低层（L2、L3、L4) 协议相关功能，采用多核结构处理高层（L7）协议相关功能。此外，采用 DSA 执行一些计算密集型处理，如有效载荷的加密/解密。通过分类处理的方式，FlatProxy 可以通过较为合适的方式处理流量，以提高系统整体的吞吐量。

4.1.2 服务网格多种数据处理功能的需求

服务网格包括服务发现、流量控制、可观察性和安全性[9,11,28] 等类型的功能。每种类型存在许多成熟和开发中的功能以及用户自定义的功能，这些功能具有不同的操作行为，如控制密集型、计算密集型、I/O 密集型等。同时，上述功能执行时会根据数据的内容动态选择执行的操作。服务网格功能多样动态可扩展的特性要求 FlatProxy 不仅需要高性能的数据处理能力，还需要支持丰富的处理功能以及可扩展能力。为了实现该需求，本文首先将服务治理功能分为配置、控制和数据处理功能。基于分类结果，本文针对数据面提出了 RTC 硬件多线程模型。该模型通过匹配动作机制，利用软件定义的方式实现灵活的数据流动态映射。

4.1.3 多租户性能和安全隔离的需求

许多基于 CPU 的服务网格优化和实现并没有考虑云计算多租户场景对性能和安全隔离的要求，而关于性能和安全隔离的应用研究则通常没有关注服务网格的影响。但本文实验表明，服务网格引入的软件操作，如资源调度、上下文切换、中断、资源竞争等，会明显影响应用性能以及安全隔离。为了满足该需求，本文深入优化了服务和 FlatProxy 之间的通信路径，简化了通信过程，减少了主机和协议栈在通信过程中的参与，从而提高系统整体的性能。同时，利用硬件 IOMMU 机制实现多租户隔离，确保了租户通信的安全性。

4.2　FlatProxy 的系统架构设计

上一小节中介绍了设计 FlatProxy 存在的几个主要需求和难点，本小节针对上述难点和需求给出了设计方案。该设计中使用了多种优化方法以获得更高的性能和灵活性。首先，如图4-2(a)展示了 FlatProxy 整体的架构设计，该架构设计主要包含以下四方面的设计优化。

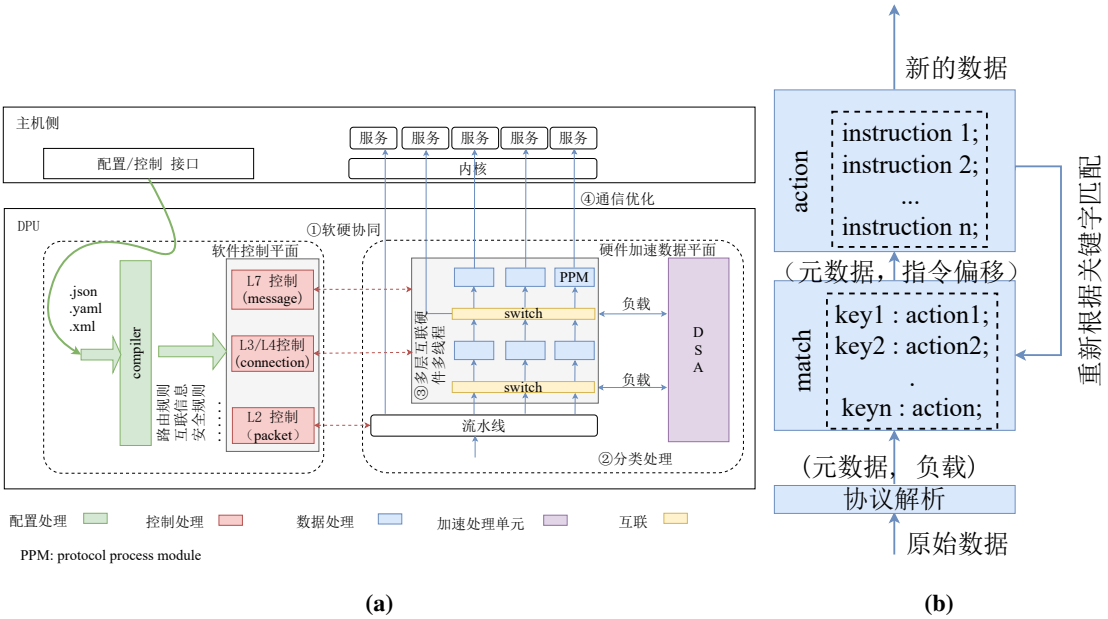


图 4-2 FlatProxy 的顶层设计。(a)FlatProxy 的模型 (b)PPM 的模型

Figure 4-2 The high-level architecture of FlatProxy. (a)The model of FlatProxy (b)The model of PPM

- **软硬协同设计**　服务治理的功能可以分为三种类型，配置，控制和数据处理。FlatProxy 通过专用硬件实现数据处理功能的加速。同时，FlatProxy 利用嵌入式 CPU 实现软件的代理。代理处理控制功能，如连接治理、控制和配置数据处理。此外，它还承担着与控制中心进行通信的职责。通过上述方式，本文可以满足功能需求的同时加速数据处理，并使本文的设计与当前最先进的控制系统 Kubernetes 兼容。
- **分类处理架构**　本文设计了一种分类处理的架构以增加带宽。L2、L3 和 L4 中的数据处理功能可以通过流水线来实现，最大程度满足带宽需求。而 L7 中的功能在协议处理以及功能实现上存在一定的复杂度，可以通过多核架构来处理，从而在性能和灵活性之间取得一定的平衡。此外，一些计算密集型任务需要由 DSA 处理，避免任务阻塞。分类处理架构根据流量类型进行分类处理，从而在保证功能完备的情况下提高系统性能。
- **多层硬件多线程模型**　本文设计了一种多层互连的多线程模型，以满足服务网格功能多样，动态可变的需求。该模型包括与每个网络层对应的包处理单元（PPM）。同一层中的 PPM 不相互通信，相邻层中的 PPM 可以通过数据交换模块传输数据。分层体系结构结合了网络数据的特征和硬件处理能力，形成一

种近似流水的方式减少控制操作的参与。对于 PPM，本文扩展了匹配动作机制，以支持多线程数据处理，实现数据处理链的动态可变。

- **通信路径优化** 本文设计了一种服务和 FlatProxy 的通信优化方式，通过消除服务和服务网格之间的冗余数据处理，以降低延迟和性能干扰。优化利用套接字直通和 SR-IOV 等多种技术，绕过宿主机和容器的内核协议栈实现服务与 FlatProxy 直接通信。同时，可以在硬件中实现通信隔离，确保多租户场景中的服务通信的安全性。

4.2.1 软硬协同设计

服务通信管理包含与链路创建、消除有关的控制功能和与数据传输相关的数据处理功能。其中，数据处理功能大部分是 I/O 密集型或计算密集型的任务，但操作较为简单可以通过硬件实现加速以提高系统性能，降低主机侧资源开销。对于控制功能，本文将其分为控制连接的功能和控制数据平面的功能。例如断路、A/B 测试等控制功能就是为了限制连接，而对数据平面的控制则是为了修改数据处理的动作，如分发流表、收集统计信息等。总之，服务网格的控制功能状态复杂，不易用硬件实现并且通常并不是性能瓶颈。因此，在不损失性能的前提下本文通过软件来实现控制功能，以简化开发过程。最后是一些配置类的功能主要负责从集群的控制中心获取配置信息。因此，本文提出的软硬件协同设计包括以下三个部分，如图4-6所示。

配置平面 该平面从集群控制中心接收配置文件，包括.yaml、.xml 和.json 文件等。然后，其将配置文件解析为控制规则和配置信息，如硬件逻辑的互连顺序、路由规则和检查规则，这些规则和信息决定如何构建和控制数据处理。一种自动配置方式可以参考当前的服务网格实现，本文按照 Istio 接口要求通过 gRPC 的方式实现相关的功能[11]。

控制平面（慢速路径） 控制平面承担的功能包括：实现针对特定连接/会话进行控制的能力；实现数据处理的慢速路径，慢速路径处理新的连接，并将规则分发到数据平面，以构建数据处理链；负责与控制中心通信，以获得控制规则，并发送端点状态，包括连接状态、服务状态、硬件状态等。为了增强可扩展性，本文提出了一种基于网络层次结构的分层控制模型，以实现高效部署。该模型包括多个控制器，例如 L2 中的包处理控制器、L3/L4 中的连接控制器和 L7 中的消息控制器。

数据平面（快速路径） 数据平面是处理流量的快速路径，是通过专用硬件构成的数据处理通路。通常，其将无法处理的新连接和其他流量发送到控制平面。然后，通过收到的配置和控制规则构建数据处理链，当后续来自网络或服务的数据满足处理条件时，则通过配置好的处理单元进行加速处理。

如图4-6所示，FlatProxy 的典型数据处理方式包括三类。第一类是配置平面从控制中心（如 Istio）获取配置信息，然后解析为配置和控制信息，通过控制平面下发给数据平面。第二类是数据处理的慢速路径，其通常用于处理新的连接并

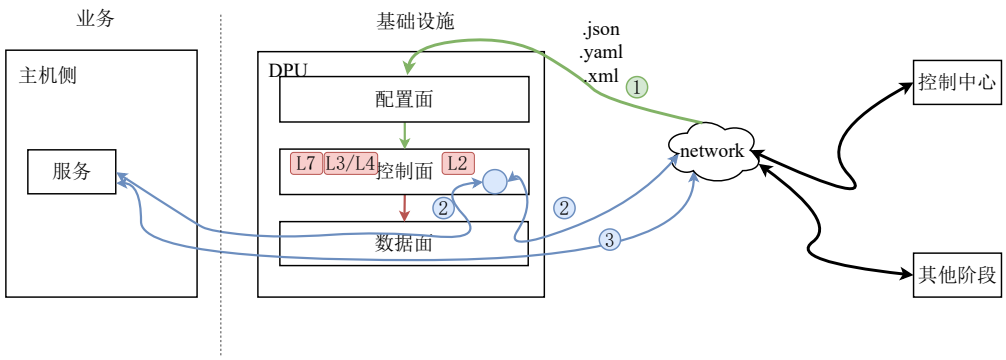


图 4-3 软硬协同设计

Figure 4-3 The software and hardware co-design

根据连接信息下发配置信息给数据平面构建后续数据处理的快速路径。第三类是数据处理的快速路径，该快速路径可以通过静态配置，也可以通过控制平面动态的配置。当数据平面具有数据包的处理规则时，数据包将被快速路径直接处理。通过上述的方式，DPU 不仅能够实现数据处理加速，还能够提前承接了大部分连接的预处理工作减少了对主机侧业务的干扰。

本文还注意到在短连接频繁出现的场景中，新连接的创建、处理销毁将成为性能瓶颈。尽管本文没有详细阐释相关解决方案，但一些研究提出了连接管理的硬件加速方案，这些硬件设计和优化方法可添加到当前的架构中。

4.2.2　分类处理架构

服务网格包括多个网络层次的数据处理功能。其中，每个层次的数据处理对象不同、操作不同、需求也不同。因此，本文针对它们各自的特点提出了一种分类处理的架构。低层协议数据处理具有的特点是，协议的格式固定，处理的数据对象是分组，数据处理功能相对简单无状态，涉及的操作主要包括转发、过滤和修改报头，但其做为网络的出入口需要更强大的数据包处理能力，例如 400Gbps 带宽的要求。尽管 L3/L4 中的功能需要高效的连接管理，并根据 TCP 连接或 UDP 会话处理数据，但本质上来说 L3/L4 中的功能仍然是在处理分组，对负载是无感知的。总体而言，低层数据处理功能具有简单、高带宽和少控制的特点，可利用可编程的流水线架构满足功能需求且提供更高的数据处理带宽。相反，应用层（L7）中的协议处理和服务治理多变复杂，处理对象是消息需要对应用负载的识别，简单的流水线很难满足需求。因此，本文采用众核架构在性能和灵活性之间进行平衡。在本文的原型验证中将 NP（Network Processor）作为基本处理单元，它可以通过指令的形式实现大部分服务治理数据处理的功能，并对相关的网络数据处理进行加速。此外，一些计算密集型操作，如 L3/L4 中的 GSO 和 GRO 以及有效载荷的加密/解密，并不适合使用通用的处理单元处理。它们需要特定的硬件进行加速。上述不同的数据处理架构通过灵活的互连组成异构的计算平台，提升系统整体处理性能。综上所述，本文针对不同层的数据进行分类处理以满足服务网格功能和性能的双重要求。通过该设计方式可以获得三方面

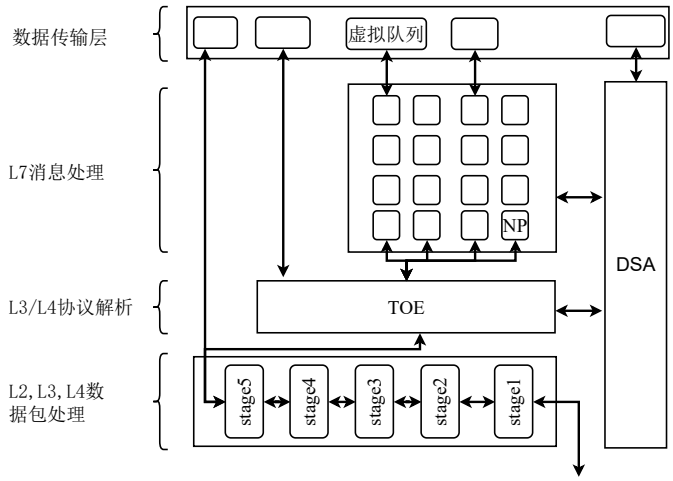


图 4-4 分类处理架构

Figure 4-4 Multi-types mixed data processing architecture

的好处。

更高的性能 在实际系统中，流量类型多样。本文的设计可以对流量进行分类，并根据协议标识符将其发送到合适的硬件单元进行处理，这有利于提高数据处理的并行性和处理延迟，充分发挥硬件加速能力。

简化了架构设计复杂度 该分类架构将服务治理拆分为几个子问题，从而降低了系统设计的复杂性。本文可以调用现有技术来实现完整的系统，如 OVS、TOE、NP 等。另外，尽管该设计需要灵活的互连，但网络数据处理层层递进的特点将有利于减少 NoC 的复杂度，提高整体性能。

较好的可扩展性 本文的设计采用了几种主流架构，可有效应对大部分的服务通信治理功能。同时，可编程的流水线和 RTC 众核的架构都具备较强的功能扩展能力，可支持新功能的开发扩展。

4.2.3 硬件多线程模型

本文根据网络数据包处理的特点，每一层只与相邻层通信并且控制操作单一，设计了分层数据流架构。在该架构中本文规定包处理模块（PPM）如图4-2(a)只与逻辑相邻层通信，不与同一层中的其他 PPM 交换数据，即两者之间相互隔离，这将有利于减少控制和互连的复杂度。另外，本文观察到每一层的数据处理功能都遵循相似的模式匹配-动作（match-action）[55]，可采用相同的模型来实现服务治理的大多数功能，并确保系统的可扩展性。因此，为适应更复杂应用场景和协议，本文提出如图4-2(b)所示的扩展模型如下。

$$< parse, (match, action)^{*} >$$

- **parse** 在此阶段，原始数据将解析为数据负载和元数据。由协议头字段组成的元数据包括一组用于选择操作的关键字，如 IP, port, URL, Cookie 等。

- **match**　在此阶段，使用元数据作为关键字从动作列表中选择动作，然后将动作指令加载到动作执行模块中。动作列表可能位于本地缓存，数据库或系统的控制中心。本文主要关注位于本地缓存的动作列表。
- **action**　在此阶段，进行处理数据，并将修改后的元数据和有效载荷发送到下一个 PPM、DSA 或匹配模块进行处理。PPM 可以是用户自定义的，用户可通过各类编程接口对它的具体行为进行控制。

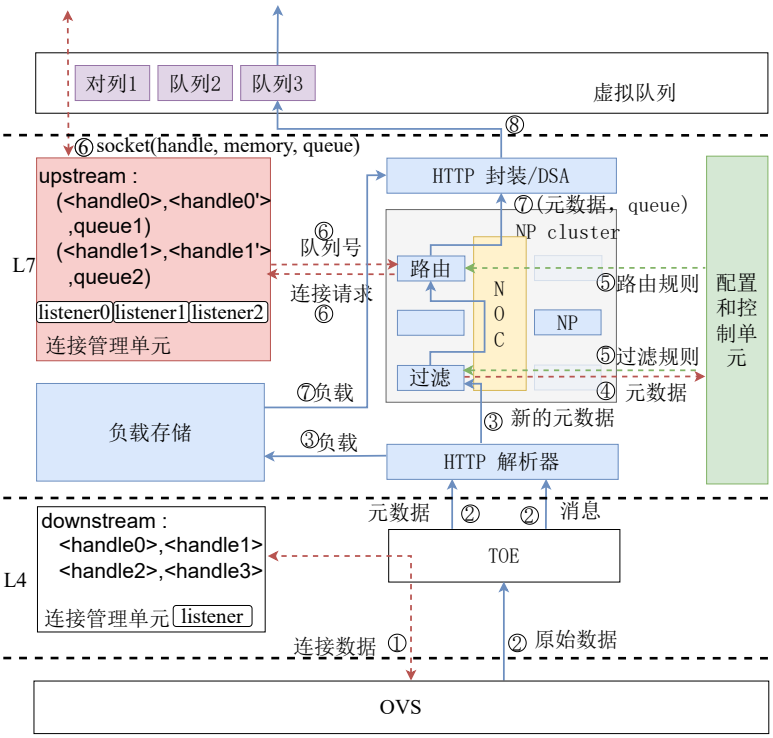


图 4-5 多层互连硬件多线程模型

Figure 4-5 Multi-layers hardware multi-threads model

基于上述架构，本文构建了一个应用层的实例 HTTP 代理路由如图4-5。该代理路由需要接管用户发起的请求，选择合适的服务建立新的连接，将用户的请求内容转发给目标服务。本文使用 NP 实现了 HTTP 路由，其伪代码如算法1所示。

如图4-5所示，HTTP 代理路由的数据处理流程为：

(1)　接收下游客户连接请求建立连接。当用户发起请求时，由下游连接管理模块与用户建立连接。同时，通知 OVS 后续的数据处理分组上送 HTTP 的处理单元。

(2)　协议解析。当数据包到达时，它们会被 TOE 解析并重新组合为一条消息发送到 HTTP 解析模块。HTTP 解析器将消息解析为新的元数据（metadata）包括低层协议的元数据和 HTTP 头字段。

(3)　送入数据处理单元。元数据上送 NP 处理单元，该示例中送入过滤处理单元。同时，负载送入存储单元缓存。

(4) 将元数据送入控制面。过滤处理模块根据元数据的信息如果未检查到对应的处理规则，则将元数据上送至控制平面进行处理。

(5) 控制面下发配置规则。配置控制模块根据元数据将配置和控制规则分发到数据处理模块（NP 集群），以形成数据处理链。

(6) 与上游服务建立连接。当数据处理链准备就绪时，处理单元根据元数据进行匹配执行相关指令。在该示例中，数据处理的流程依次为，过滤，路由。当数据进入路由模块后，如果当前请求数据没有上送服务的虚拟队列，路由模块通知上游连接管理模块与服务建立新的连接，并分配虚拟队列与服务进程的内存绑定。

(7) 数据封装。HTTP 的封装处理单元接收到路由模块传递的元数据和队列号，然后根据元数据获取负载数据进行封装。

(8) 转发数据给服务。封装模块块将封装好的 HTTP 消息发送到分配的虚拟队列，然后上送主机侧服务。

算法 1 HTTP Routing Process

Input: Metadata

Output: Routing action

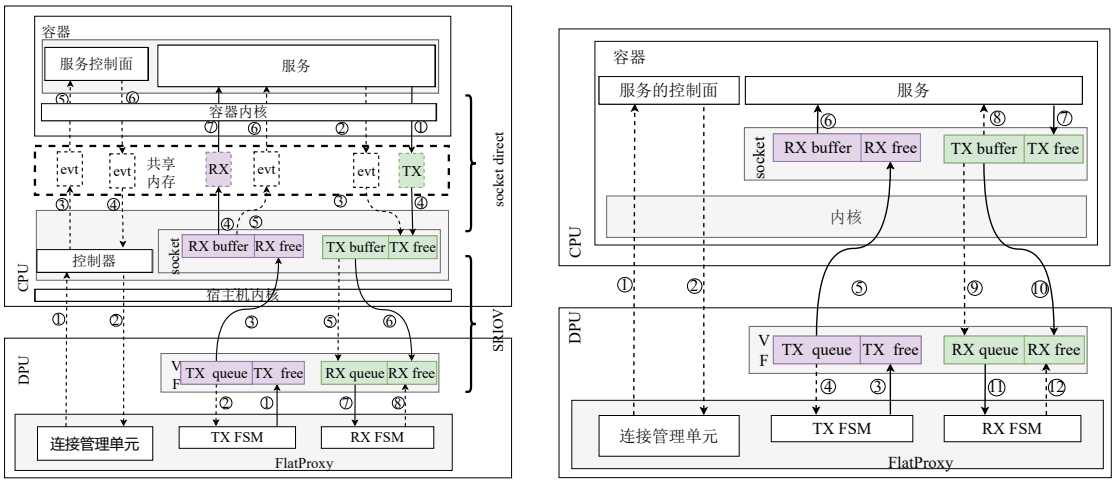
```
key = Key(metadata.dip, metadata.dport)
if Listner[Hash(key)] == 1 then
    if match(Hash(metadata.url.path)) == 1 then
        key= Key(metadata.sip, metadata.sport, metadata.dip, metadata.dport)
        if Queue[Hash(key)] == 0 then
            endpoint = load_balacing(metadata)
            Queue[Hash(key)] = connection(endpoint)
        end if
        metadata.queue = Queue[Hash(key)]
        Update metadata
        Send metadata to http deparser/DSA
        initial metadata
    else if  then
        initial metadata
        break
    end if
else if  then
    initial metadata
    break
end if
```

上述的实例，是一个系统实现后的正常操作流程，忽视了一些实际的问题，如动作指令的存储和获取方式。本文默认它可以存储在片上内存，但实际系统

中规则的数量远大于片上内存容量。常规的解决方法是将规则存储在数据库中，并将热点规则缓存在芯片上。在本文中，设计验证都是基于规则在片上缓存实现的，更多的访存问题将在接下来的工作中进行讨论。

4.2.4　多租户通信路径优化

云原生多租户场景中主机共享对性能和安全性隔离提出了严格要求。当在 DPU 上实现代理时，服务和服务网格之间的原始通信方式需要跨越多个层次造成了严重的性能损失和干扰。为了提高性能，本文对该数据通信路径进行了深度优化以减少冗余数据处理。第一类优化是通过 SR-IOV 实现服务与硬件之间的直接通信，避免了容器、主机和硬件之间的多次数据拷贝。第二类是通过套接字直连技术消除了节点内通信的网络协议处理过程，即服务与代理的通信可以不用进行 TCP/IP 协议栈的处理。通过上述方式，主机侧可以减少 CPU 资源的占用。同时，由于通信路径缩减系统整体性能可有效提高，直连的方式也可以减少性能干扰。另外，SR-IOV 也可以从硬件侧对性能和安全隔离提供支持，而基础设施和业务的物理隔离也减少了性能干扰和安全漏洞。



(a)　　(b)

图 4-6 服务与 FlatProxy 通信优化。(a) 松耦合通信 (b) 紧耦合通信

Figure 4-6 The optimization of the communication between servie and FlatProxy. (a) The loosely-coupled communication (b) The tightly-coupled communication

根据云原生服务运行时长短的特点[48,56]，本文基于上述技术设计了两类不同的通信模型。第一类主要针对运行时间长，数据处理突发的服务。如图4-6(a)所示，通过在服务和 FlatProxy 之间添加一个共享的数据收发中间层可以减少非业务功能对 CPU 资源的占用，但服务和该中间层的通信会造成一些性能损失。第二类针对运行时间短，性能要求高的服务。如图4-6(b)所示，数据收发逻辑被嵌入到服务进程中减少了服务和 FlatProxy 的通信路径，但会导致大量的资源占用。两者的使用各具优缺点可以根据服务类型动态选择。

不论是那种方式，整个通信过程都需要软件和硬件的协同处理。在软件方

面，共享数据转发平面可以通过扩展 VPP 来构建，然后服务通过共享内存的方式与该转发平面通信，转发平面通过 VF 与硬件通信，数据包的收发本文采用 DPDK 实现。另外，VF 需要由 CNI 进行配置，以便与 Kubenetes 兼容。在硬件侧 SR-IOV 是绕过宿主机的关键，它支持容器在没有主机干扰的情况下访问硬件。其他一些技术，如 VDPA，也可以用来实现服务和硬件的通信。此外，本文通过调用 DPDK 和 VPP 提供的库来解决一致性问题，尽管这不是解决此类问题的最佳方法，但它足以验证本文设计的可行性，并且与之前的服务网格实现相比仍具有明显的性能优势。

该小节以图4-6(b)所示的通信方式为例，通信过程包括，内存绑定、数据发送和数据接收。具体过程如下：

- **内存绑定** ①当 FlatProxy 与服务建立新的连接时，连接管理模块通过 PCIe 向服务控制器发送请求，然后服务控制器分配一对内存块作为 RX/TX 队列；②服务控制器将分配的内存地址发送到连接管理模块。连接管理模块将地址与建立连接时已经分配的虚拟化队列绑定。
- **数据发送** ③在内存地址与虚拟化队列绑定后，当数据到达虚拟化队列时，将内存地址发送到 DMA；④返回虚拟队列当前空闲的地址；⑤队列中的数据通过 DMA 发送到 Socket 的 RX 队列；⑥当数据到达内存时，服务轮询获取数据。
- **数据接收** ⑦服务通过 Socket 将数据写入 TX 队列；⑧空闲地址返回给服务；⑨驱动程序通知 DMA 发送数据的内存地址；⑩当 DMA 获得数据地址时，它将数据从 TX 缓冲区传输到硬件的 RX 队列；⑪数据到达 DPU 后，FlatProxy 获取数据；⑫释放 RX 队列。

通过对数据路径的深度优化，FlatProxy 不仅可以提高系统性能，而且硬件可以实现更严格的性能和数据隔离，以保证访问安全。在目前的 DPU 设计中实现了更先进的基于“零信任”的安全访问，这将进一步推动 DPU 在服务通信管理中的应用。

4.3 基于 YUSUR K2-pro 原型验证平台的实现

基于上述的服务网格设计方法，该章节介绍了基于驭数 DPU K2-Pro 原型验证平台的服务网格实现。相关内容涉及广泛，本文主要关注于虚拟网络以及服务网格代理能力的内容。其中，虚拟网络是实现服务网格的基础，负责完成覆盖网络和底层网络之间的转化以实现数据在真实网络中的传递。服务网格的相关操作是基于虚拟网络进行的，它所能感知到的覆盖网络与实际的网络在逻辑结构上一致。因此，虽然服务网格依赖于虚拟网络，但二者在功能上相互不存在影响。接下来，本章对相关内容进行详细介绍。

4.3.1 系统总览

从 DPU 的角度来看，整体的架构如图4-7所示包括配置通道、控制平面、数据平面、数据通道和虚拟网络五个部分。

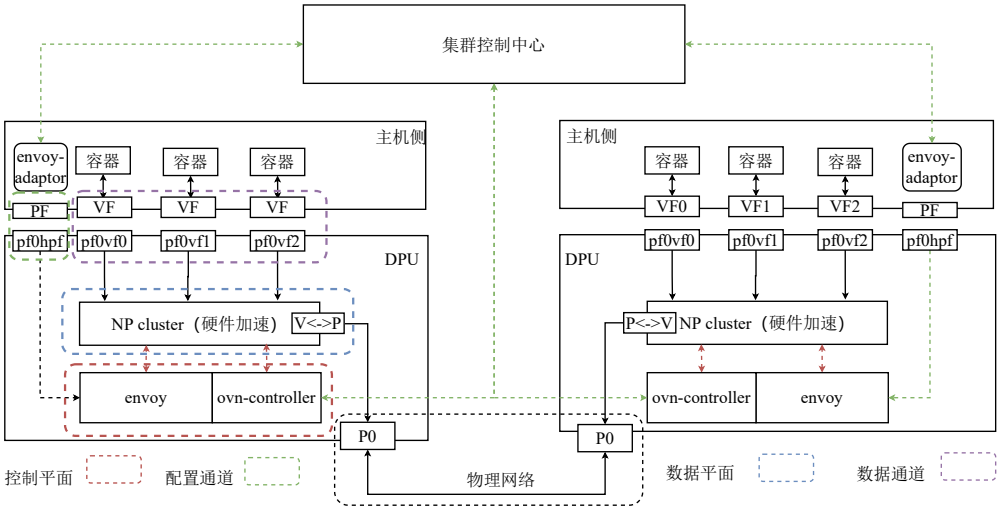


图 4-7 基于 K2-Pro 原型验证平台的服务网格系统架构

Figure 4-7 The system architecture of service mesh based on the K2-Pro platform

配置通道　服务网格的整体架构是由一个控制中心和分布在各个节点上的数据处理代理组成。在本文的实现中使用的控制平面是 Istio，它依托于容器编排系统 Kubernetes 进行部署。为兼容当前系统，基于 DPU 的服务网格实现在主机侧维持配置信息的接收端口。图4-7所示的 Envoy-adaptor 就是用于与 Istio 通信获取配置信息的模块，该模块获取配置信息后将其转换为 Envoy 能够识别的配置和控制信息，通过 gRPC 的方式将配置信息发送给 Envoy，Envoy 部署在 DPU 的嵌入式 CPU 上完成控制功能。

控制平面　控制平面在嵌入式 CPU 上通过软件实现，主要包括虚拟网络的控制模块 ovn-controller 和服务网格的控制平面 Envoy。该控制平面接收到配置信息后，首先在控制平面内部形成慢速处理过程，例如处理链路的组合。等到有新的连接到达的时候，根据配置内容对连接进行处理。如果需要对后续数据进行硬件加速，则下发控制和配置信息到数据平面。

数据平面　当前的实现主要使用网络处理器（NP）做为硬件加速模块。该模块由一组网络处理器组成，具有灵活的可编程能力和较好的性能。在驭数 K2-Pro 的板卡上支持 P4 网络编程，也可以通过指令的方式进行自定义的网络操作。在本文中涉及的功能包括路由，数据包协议信息修改，封装解封装等等。

数据通路　数据通路的实现主要包括两部分，软件上在主机侧和 DPU 侧形成网络设备的对应关系，如主机侧的 VF 网口和 DPU 侧的 pf0vf0 网口。对应的两个接口之间可以完成主机侧和 DPU 侧数据的直接传输，类似于网络中的 veth 对。硬件上通过 SR-IOV 硬件虚拟化，可以支持总共 511 个虚拟的设备实现服务与 DPU 的直接通信。

物理网络　物理网路是真实的网络，是真实的传输数据的网络。在云原生中，不同节点上的服务进行通信时通常需要利用虚拟网络进行。在数据平面的介绍中包括对数据包的包头修改，封装和解封装等，这些操作可以完成虚拟地址和

物理地址的转换。在完成相应的转换后，数据包就可以被传输到 DPU 对外的网口进入到物理网路中。

4.3.2 虚拟网络的实现

虚拟网络是云原生应用运行的基础平台，也是服务网格运行的前提条件，其主要完成数据包在覆盖网络与底层网络的转化。当前的各类基于软件实现的虚拟网络方案在应对高带宽（>100Gbps) 场景时难以提供所需的吞吐能力，软硬协同成为改善系统性能的关键方式。本小节介绍以驭数 DPU 作为基础硬件平台开发出满足云原生的虚拟网络平台，实现开箱即用，高性能，适用性广的虚实转化能力。

根据调研分析，当前虚拟网络的构建方式主要分为三种类型，隧道封装，Host-GW 以及 NAT。总体而言，不论是那种方式，数据包都需要经过覆盖网络路由，覆盖网络和底层网络地址转化，底层网络处理。其中，覆盖网络路由依赖于虚拟交换机，底层网络的处理主要是 DPU 将转换后的数据从指定的物理网口中发送出去。覆盖网路和底层网络地址转换依赖于虚拟地址和物理地址的映射关系，在云中这种映射关系通常通过一个集中的数据库进行存储，各个节点在使用时通过远程访问的方式获取映射关系。

基于以上认识，本文介绍了基于驭数 K2-Pro 软硬件协同加速的虚拟网络实现。该实现不仅利用 NP 硬件加速数据包处理，而且按照当前虚拟网络基于 Kubernetes 的 CNI 构建方式[57] 进行实现，在提高性能的同时充分融入当前的软件生态。该实现将网络的处理分为两部分，虚拟网络前端（CNI plugin frontend）和虚拟网络后端（CNI plugin backend)。其中前端部署在主机侧遵循 CNI 标准用于与 Kubernetes 进行通信，获取地址映射信息以及与后端的交互，后端位于 DPU 侧包括控制和数据处理两部分是处理网络数据的主要部件，其中控制面接收前端的控制信息，形成慢速的数据处理通路。然后，这些慢速数据通路可映射到专用的硬件加速单元 NP 集群上，通过硬件进行数据处理加速。整体架构的设计和实现包括以下几个方面。

地址映射同步 虚拟网络实现最重要的就是虚拟地址与物理地址之间的转化。在基于驭数 DPU 的虚拟网络中地址的映射关系由部署在主机侧的 CNI 插件从集群的控制中心获取，然后下发给 DPU 侧的 OVS 控制器。在下发给 OVS 控制器之前，配置的内容会被转换为符合 ovs 的指令如下。

```
ovs-vsctl add-port ovs-br0 vxlan0 -- set interface vxlan0 type=vxlan options:remote_ip
=172.16.10.200
```

数据转发

根据配置信息，DPU 侧的控制面首先会形成相应的慢速数据路径用于处理首包或快速路径无法处理的数据内容。在本小节中，如图4-8所示 DPU 的嵌入

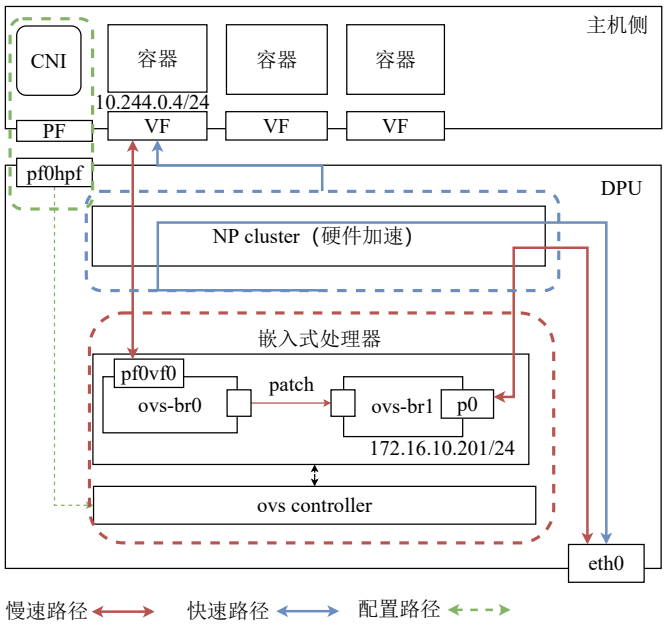


图 4-8 基于 DPU 的虚拟网络系统架构

Figure 4-8 The system architecture of virtual network based on the K2-Pro platform

式处理器上根据配置信息构建两个虚拟网桥，网桥 ovs-br0 负责转发本节点内部的流量，当流量需要传递到其他节点时，数据包会通过直连的通道 patch 传递到 ovs-br1 然后通过 p0 端口发送到网络中。在该过程中，数据包进行封装完成覆盖网络到底层网络的转换。上述的过程在开启硬件卸载功能后会被映射至 NP 集群上，此时所有的后续流量在 NP 集群中进行转发而不需要上送到嵌入式处理器中。

上述的实现将数据处理的相关操作由主机侧转移至 DPU 侧，一方面减少主机侧资源开销，另一方面实现性能提升。同时，该实现充分兼容当前基于 Kubernetes 的网络架构，减少配置面重复开发的难度。

4.3.3　服务网格的实现

本小节介绍的服务网格实现主要关注 TCP 代理的加速，7 层代理的处理逻辑大体类似可参考4.2.3小节。TCP 代理的主要功能就是接收下游用户侧的连接，然后根据连接信息进行负载均衡选择上游服务建立新的连接。当有数据到达后先送到 TCP 代理，然后由 TCP 代理转发到相应的容器中进行数据处理。在实现服务网格的 TCP 代理功能过程中主要包括获取配置信息、下发控制规则、数据处理加速。

获取配置信息

如图4-9所示，配置信息的获取程序为部署在主机侧的 envoy adaptor，它通过 gRPC 与部署在 master 节点的控制中心 Istio 进行通信获取配置信息。该配置信息规定了 envoy 监听的端口和负载均衡的节点如下。

```
static_resources:
```

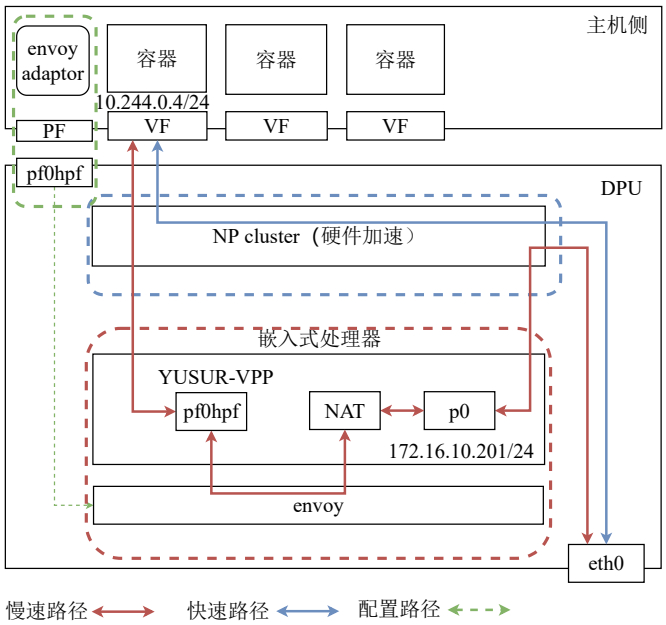


图 4-9 基于 DPU 的 TCP 代理系统架构

Figure 4-9 The system architecture of TCP proxy based on the K2-Pro platform

```
listeners:
- name: listener_in
  address:
    socket_address:
      address: 0.0.0.0                          #监听地址
      port_value: 15001                         #监听端口
  filter_chains:
  - filters:
    - name: envoy.filters.network.tcp_proxy
      typed_config:
        "@type": type.googleapis.com/envoy.extensions.filters.network.tcp_proxy.v3.
  TcpProxy
        stat_prefix: destination
        cluster: cluster_local
        max_connect_attempts: 5

clusters:
- name: cluster_local
  connect_timeout: 10s
  type: logical_dns
  # Comment out the following line to test on v6 networks
  dns_lookup_family: V4_ONLY
  lb_policy: round_robin
  load_assignment:
    cluster_name: cluster_local
    endpoints:
    - lb_endpoints:
      - endpoint:
          address:
            socket_address:
              address: 10.244.0.4               #负载均衡地址
              port_value: 10000                 #负载均衡端口
```

下发控制规则

根据上述的配置信息，部署在 DPU 的 YUSUR-VPP 首先修改路由，在 NAT 模块中加入修改指定数据包的能力。在本小节，YUSUR-VPP 的 NAT 将修改所有从 p0 进入的数据包的地址和端口，然后上送给 envoy。Envoy 根据配置信息进行负载均衡，然后通过 pf0hpf 端口上送给主机侧的部署在容器中的服务。如果 envoy 开启了卸载的能力，它将下发对应的规则到 NP 集群形成快速数据处理路径。一个控制规则的例子如下所示。

```
#进入节点的原始数据的目的地址（服务地址）将被修改为负载均衡后的地址
ovs-ofctl add-flow br_name "cookie=0,priority=40001,in_port=p0,ip,nw_dst=172.16.10.201,
    tp_dst=15000 action=mod_nw_dst:10.244.0.4,mod_tp_dst:10000,output:pf0hpf"
#返回的数据包源地址将被修改为服务的地址
ovs-ofctl add-flow br_name "cookie=0,priority=40001,in_port=pf0hpf,ip,nw_src=10.244.0.4,
    tp_src=10000 action=mod_nw_src:172.16.10.201,mod_tp_src:15000,output:pf0"
```

加速数据处理

根据控制平面下发的规则，如图4-9所示，所有满足匹配条件的数据包将直接通过 NP 集群的处理上送至主机侧的容器中，而不需要先通过 Envoy 的处理。通过上述过程 TCP 代理的功能可以有效的卸载至硬件侧获得加速能力。

本小节基于驭数 K2-Pro 的描述尽管只是服务网格的一小部分功能的实现，但其实现的合理性和有效性可被证实，类似的功能也可以遵循相同的逻辑进行处理。

4.4　本章小结

本章介绍了基于 DPU 的服务网格架构设计与实现。在4.1小节分析了将服务网格代理卸载到 DPU 上存在的三个关键需求，更高的带宽、多样的数据处理功能以及性能和安全隔离的需求。基于此，4.2小节采用了四种设计优化方法，软硬协同设计、分类处理架构、硬件多线程模型和通信路径优化。基于设计方法，4.3小节展示了基于驭数 K2-Pro DPU 实现的服务网格的两个案例，虚拟网络和服务网格的 TCP 代理。

第 5 章　实验设计与性能评估

本章从吞吐，延迟以及数据传输稳定性等方面评估了 FlatProxy 的性能。通过对比实验，本文展示了 FlatProxy 与传统服务网格相比取得的性能优势，并展示了每部分优化取得的性能提升。

5.1　实验环境及配置

实验平台

本文搭建了一个端到端的测试系统，该系统包括两台部署了 DPU 的服务器，服务器通过 25G 光纤互联。服务器 CPU 型号为 Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz，共具有 80 个核。DPU 为驭数自研 K2-Pro，本文采用基于 Xilinx ALVEO™ U200 FPGA 板卡实现的原型验证平台进行相关性能的测试。同时，部分应用层数据的评估依赖于 NP 模拟器，该模拟器可实现时钟级别的性能评估。

实验负载

本实验针对的场景是服务间通信，通信的数据流量根据网络层次的不同可以分为低层（L2,L3,L4）的数据包（分组）和应用层（L7）的数据（消息）。其中针对低层的服务网格测试，实验关注的测试指标包括系统的吞吐能力，传输延时以及传输的稳定性，影响这些系统性能的因素包括包的大小，突发流量的大小，连接数，以及 CPU 处理核的数量。因此，在测试过程中本文采用 iperf[58] 做为吞吐测试的工具，其可以模拟不同大小突发流量做为测试负载。同时，本实验采用 sockperf[59] 做为延时的测试工具，其可以模拟不同大小包的收发并做延时统计。对于传输的稳定性测试，本文采用 iperf 发送 UDP 包测量延时抖动做为衡量标准。

在本文中应用层服务通信主要关注 HTTP 协议，该通信过程一般注重请求响应的能力，包括响应速度，响应的平均延时和长尾延时。在本实验中采用 fortio[60] 做为负载的生成器，其能够以不同的速率发送请求，同时支持不同并发条件下的请求发送。另外，本实验也采用单侧系统测试的方式测试服务和代理通信优化的性能。该系统一端部署待测试系统，另一端部署负载收发程序，采用的负载生成器是 wrk。

上述的负载发生和服务部署都位于主机侧，DPU 仅负责处理传输的数据。

基准测试

本实验选择了三类基准系统进行测试，分别是传统的服务网格部署、利用 sockmap 优化的服务网格部署、利用已有的硬件技术优化的服务网格部署。其中，sockmap 优化主要是针对服务和服务网格 Pod 内的通信，已有硬件技术优化主要是除服务网格以外的硬件优化，如 SR-IOV、TOE、OVS 卸载。本实验复现了上述的三类基准系统与本文提出的 FlatProxy 进行比较。

5.2 微测试结果及分析

本章节在部署服务网格的条件下，分别对虚拟网络，服务网格和通信路径优化三部分进行了性能测试和评估。

5.2.1 虚拟网络的测试

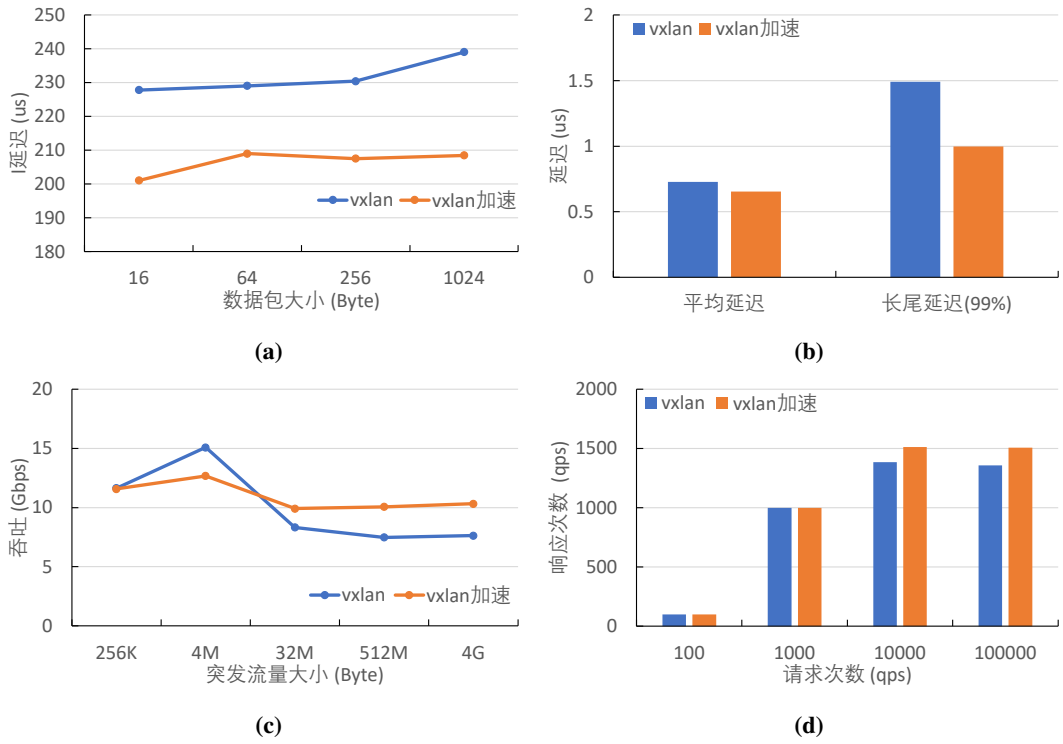


图 5-1 软件 VXLAN 与硬件加速 VXLAN 的比较。(a) 延迟随包大小变化 (b) 请求响应延迟 (c) 吞吐随突发流量大小的变化 (d) 请求响应速率

Figure 5-1 The performance comparison of virtual network. (a)The latency over varying the packet size (b)The respond latency of request (c)The throughput over varying the traffic size (d) The respond rate

本小节进行了服务网格中虚拟网络性能的测试，测试的双方是软件 VXLAN 和硬件加速 VXLAN。从图5-1(a)中可以看出 VXLAN 硬件卸载后，TCP 传输的延迟降低了 20us 左右，大约占比 9%。图5-1(b)表明在应用层的传输过程中，VXLAN 加速对整体性能的提升较小，但可以有效减少长尾延时，这主要是因为软件进行 VXLAN 处理时需要进行路由地址的查询，包头的封装，解封装等等严重依赖于 CPU 资源的操作。后续的图5-1(c)反映了 TCP 通信流量的比较情况，可以看出在较大流量的测试中，卸载 VXLAN 后可以提高吞吐的性能。在较小流量的测试中，出现的反常情况根据测试分析是由于卸载 VXLAN 后的发送延时比较大，这与工具内部的具体负载生成逻辑有关，本文暂不做具体分析。图5-1(d)展示了 HTTP 的请求通信速率，其与响应延迟成反比。

5.2.2　服务网格的测试

数据传输延迟评估

本小节 FlatProxy 的延迟评估包括 TCP 通信延迟比较，HTTP 平均延迟和长尾延迟比较以及 UDP 延迟抖动的比较，如图5-2所示。

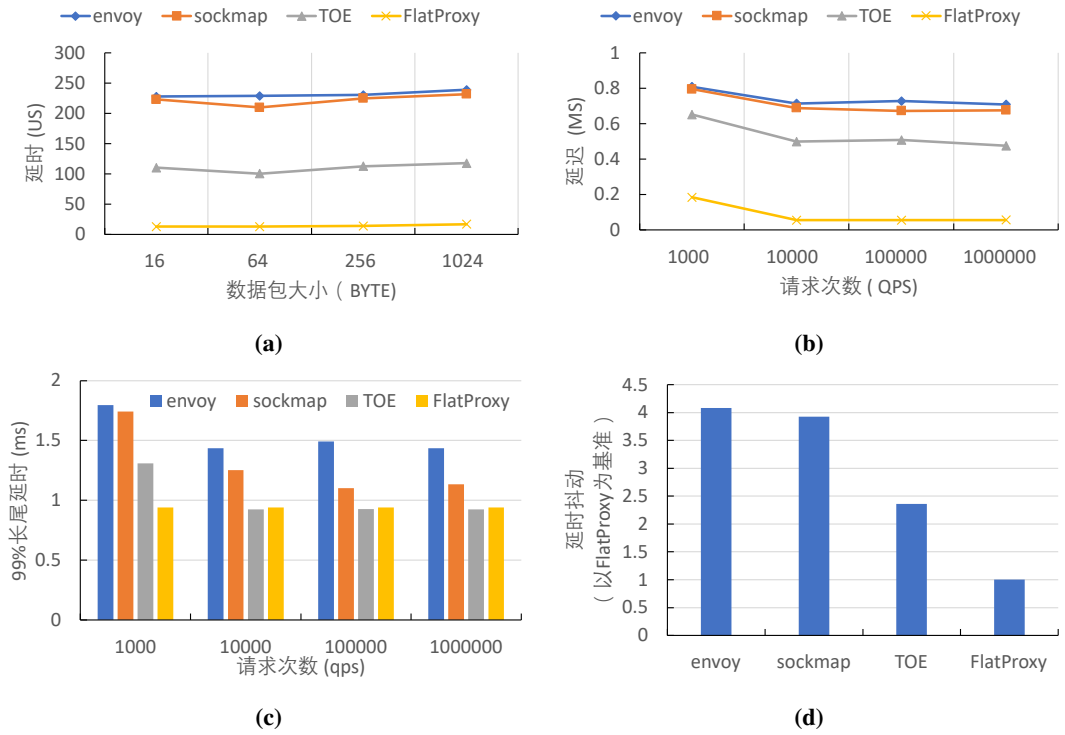


图 5-2 不同优化策略的通信延迟比较。(a) 延迟随包大小变化 (b) 请求响应平均延迟 (c) 请求响应 99% 延迟 (d) 包传输延时抖动

Figure 5-2 The impact of service mesh for latency of service communication. (a) The latency over varying the packet size (b) The respond average latency of request (c) The respond 99% latency of request (d) The jitter of packet forwarding latency

图5-2(a)展示了不同优化方式下 TCP 通信的延时，可以看出 sockmap 这类软件优化带来的收益较小，而通过硬件卸载则可以有效提高系统性能。其中，本测试中 TOE，SR-IOV 以及 OVS 的硬件优化（简称 TOE 优化）使延迟从 240us 降低至 110us 左右，服务网格的硬件加速（FlatProxy) 则使通信的延迟降低至 20us 左右，总体而言可使延迟降低超过 90%。表5-1展示了传统服务网格和 FlatProxy 关键操作的延时比较，表中显示 FlatProxy 消除了 TCP/IP 协议栈和代理（TOE->proxy）之间的数据复制，环回通信则被套接字直连（VQ->service）取代。同时，与其他的优化方式相比，图5-2(d)表明 FlatProxy 的通信更稳定抖动更小，这对服务通信的质量（QoS）至关重要。

对于 HTTP 通信的延迟测试，如图5-2(b)所示为数据传输的平均延时。数据显示的结果与 TCP 数据传输类似，不同点在于服务网格的硬件优化对系统性能提升更大，这主要是因为代理处理高层协议的操作更复杂、延时也更高，通过卸

表 5-1 FlatProxy 和 envoy 实现 TCP 代理延时分析比较

Table 5-1 The comparison of the TCP proxy latency breakdown between FlatProxy and envoy

envoy	50us	4.6us	FlatProxy
vSwitch	9%	26%	OVS
TCP/IP protocol	27%	1%	TOE
proxy operation:			
TCP -> proxy	20%	–	TOE -> MAL
data processing	7%	66%	match-action logic(MAL)
proxy -> TCP	10%	–	MAL -> VQ
loopback	27%	7%	VQ -> service

载可以获得更多收益。TOE 硬件加速主要优化的是低层的协议，该优化可降低延迟 30% 左右。而 FlatProxy 是在低层协议加速的基础上完成了 HTTP 协议通信和相关处理的加速，其可降低延迟 85% 以上。如表5-2所示，FlatProxy 消除了环回通信路径和数据初始化（D-I）等软件开销。硬件上的数据传输延时可以忽略不计，远远快于主机数据复制（D-T）。此外，FlatProxy 的 HTTP 解析/分离和匹配操作逻辑等数据处理操作比软件更快。FlatProxy 低延迟的另一个原因是它没有实现统计功能和连接管理功能，这些功能并不是数据传输过程中的必须环节，理论上可以并行实现并不会影响 FlatProxy 数据处理延迟。其他需要注意的是，如图5-2(b)所示，请求速率较低时的平均延迟非常高，这是因为代理需要在请求开始时初始化一些设置，从而造成了更高的延时。当请求查询超过 10000 时，初始化开销可以忽略不记。

如图5-2(c)所示，HTTP 通信的长尾延迟在利用硬件加速后同样明显降低。FlatProxy 的长尾延时相较于 TOE 优化提高不明显，本文分析原因是由于此时的延迟抖动主要是服务带来的，目前的优化难以避免这一部分的影响，后续的工作中将进行进一步分析。

数据传输吞吐评估

数据传输吞吐的评估测试中，本文从突发流量大小，连接数以及 CPU 核三个维度进行了评估。首先是针对 TCP 通信的的吞吐评估。如图5-3(a)所示，sockmap 优化基本对吞吐没有影响说明 Pod 内通信并不是通信路径上的瓶颈。而初步的协议卸载 (TOE) 将吞吐提高至 12Gbps，服务网格的代理卸载 FlatProxy 则将吞吐提高至接近 25Gbps。在多连接测试中，如图5-3(b)所示，FlatProxy 在单条连接时数据传输吞吐已达到 24.5Gbps, 其他的优化方式如 TOE 硬件卸载随连接数增加吞吐可增加至 25Gbps，但在连接大于 16 后吞吐降低，反映了软件代理在连接管理方面存在的性能问题。在多核测试中，如图5-3(c)所示，没有进行服务网格代理的卸载的测试，性能受到 CPU 核数量影响较大。在其他的多核测试中，本文将测试负载和代理绑定到不同的 CPU 核上测出其具有类似的规律，可见代理对吞吐的重要影响，也证明了卸载代理的必要性和重要性。

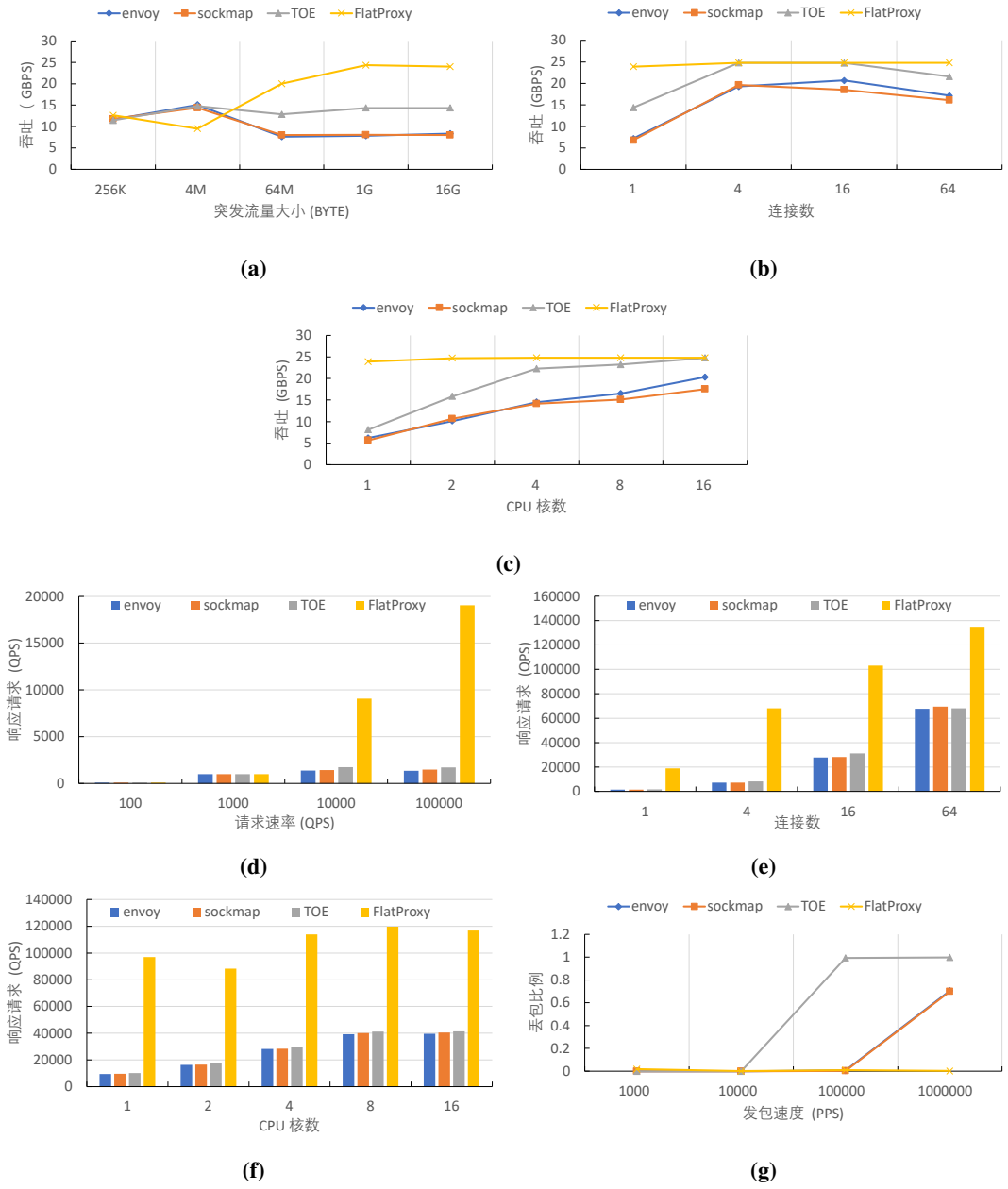


图 5-3 不同优化策略服务通信吞吐性能的比较。(a) 吞吐随突发流量大小变化 (b) 吞吐随连接数的变化 (c) 吞吐随 CPU 核数的变化 (d) 响应速率随请求速率的变化 (e) 响应速率随连接数的变化 (f) 响应速率随核数的变化 (g) 包转发的丢包情况（Envoy 的测试与 sockmap 测试结果重合）

Figure 5-3 The impact of service mesh for throughput of service communication. (a)Throughput over varying traffic size (b)Throughput over varying connection numbers (c)Throughput over varying core numbers (d)Resond rate over varying request rate (e)Respond rate over varying http connections (f)Respond rate over varying core numbers (g)Performance of packet forwarding.

表 5-2 FlatProxy 和 envoy 实现 HTTP 代理延时分析比较

Table 5-2 The comparison of the HTTP proxy latency breakdown between FlatProxy and envoy

envoy	62.5us	17.6us	FlatProxy
vSwitch	3%	11%	OVS
TCP/IP protocol	8%	1%	TOE
proxy operations:			
connection, statistical	26%	–	–
D-T	16%	–	data transmission
D-I, D-P	12%	28%	http parser
D-CK, D-OP, D-M	22%	29%	match-action logic
D-DP	6%	28%	http deparser
loopback	7%	3%	VQ -> service

接下来，本文评估了 HTTP 请求响应的速率随连接和 CPU 核数的变化。如图5-3(d)所示，在三种基准测试中请求响应的提升并不明显，而 FlatProxy 最高可将请求响应的性能提高 6 倍以上。该实验结果体现了服务网格代理是影响 HTTP 请求性能的关键因素。在多连接测试中，图5-3(e)进一步体现了 FlatProxy 的优势，最高响应速率可达到 130000 qps。多核测试具有类似的结果。

最后，本文也评估了包转发的性能。如图5-3(g)所示，FlatProxy 可保证在包转发大于 1000000 pps 的情况下仅发生极少的丢包，而其他的优化方式则在转发速率大于 100000 pps 时就发生严重丢包。一个异常现象在于基准测试 3（TOE 优化）的丢包率远高于其他情况，根据相关分析，可能的原因在于硬件卸载加速后，转发能力提高，而软件测对于大流量的处理能力和缓存能力并没有进行相应的适配，从而导致出现了上述的问题。总体而言，当前使用的服务网格代理对于 UDP 的转发优化存在一定的问题，具体的原因将在未来的工作中进一步分析。

资源占用评估

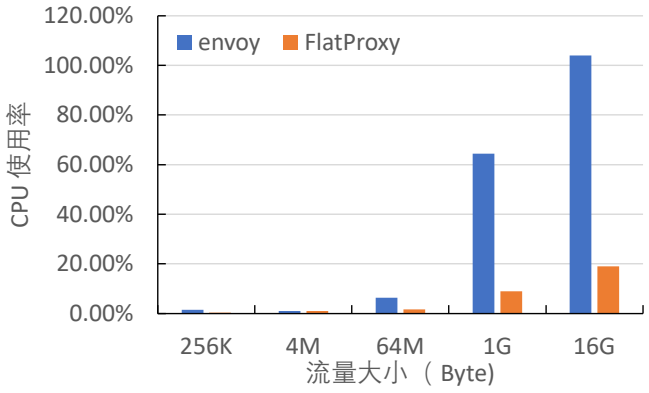


图 5-4 CPU 资源使用率随突发流量大小的变化情况

Figure 5-4 CPU utilization over varying traffic size

将更多的 CPU 资源用于租户业务运行是云提供商的主要目标。与传统的服务网格解决方案相比，FlatProxy 可以节省许多 CPU 资源。在限定 10Gbps 带宽的前提下，从图5-4中可以看出，FlatProxy 占用的 CPU 资源很少，留给服务的 CPU 资源更多。而在传统的服务网格部署中，代理占用了 70% 以上的 CPU 资源，是负载测试程序的三倍多。

5.2.3　两类通信方式的测试

该小节的测试，主要是针对5.2.3小节提出的两类服务和 FlatProxy 的通信方式进行测试。为减少其他进程的性能干扰，本测试利用单系统的方式进行，即只在一端部署测试系统，另一端为负载收发程序 wrk。该测试的双方是松耦合的通信方式和紧耦合的通信方式。其中，松耦合的设计使用了一个专门的共享数据收发中间层将服务和 FlatPrxoy 进行了隔离，这也是通常的网络通信过程实现的 IO 模式，可以减少资源的占用。紧耦合的通信方式将数据收发与服务集成为同一个线程，理论上可以减少收发延时。根据图5-5(a)所示，第二类优化方式可以降低通信延时 27%。在多连接性能测试中，两种方式都无法避免由于多连接带来的延迟增加，但紧耦合的通信延迟基本能够稳定的优于松耦合的通信方式，结果如图5-5(b)所示。从请求响应次数的角度看，如图5-5(c)所示，第二类通信方式同样能够带来一定的性能优势。

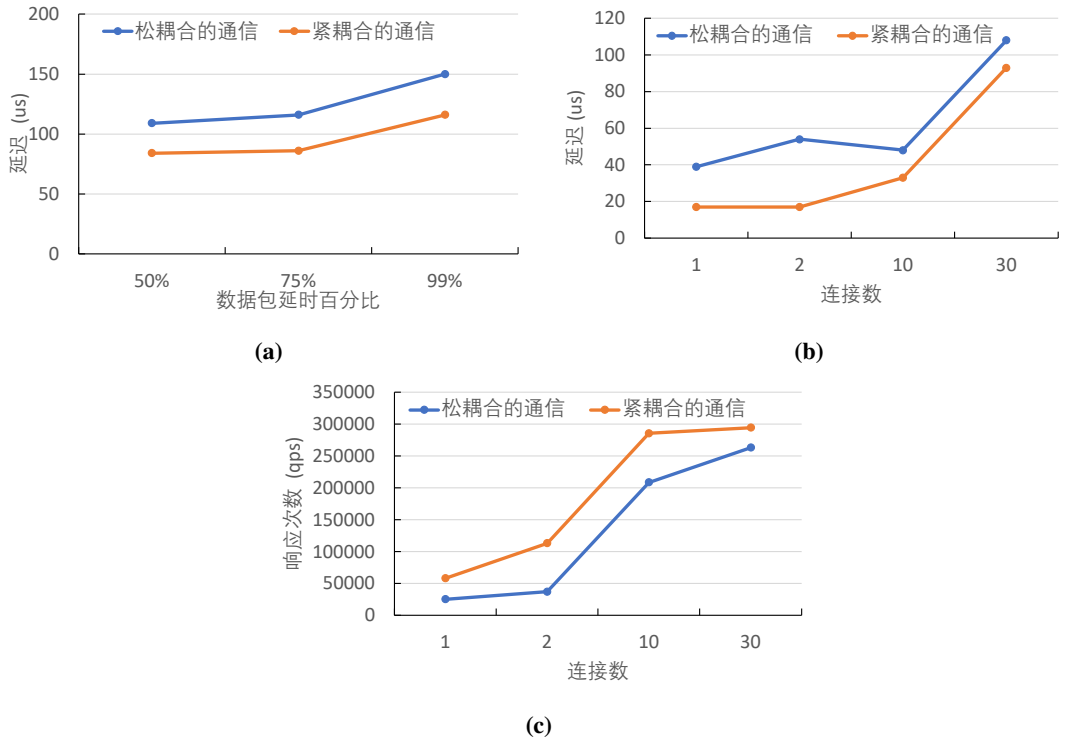


图 5-5 两种服务与 FlatProxy 通信方式的性能比较。(a) 不同比例包的请求延时 (b) 响应延时随连接的变化 (c) 请求响应速率随连接数的变化

Figure 5-5 The performance comparison of communication between service and FlatProxy. (a) The respond latency over the proportion of packages (b) The respond latency over connections (c) The respond rate over connections

5.3　实验总结

本章基于一个端到端的系统进行了 FlatProxy 的测试评估。测试的内容包括虚拟网络，服务网格和通信优化。其中，虚拟网络是服务网格运行的基础，通信优化是服务网格与服务传递数据的必须环节。在5.2.1小节展示了虚拟网络加速带来的影响，可以看出其性能提升有限，并不是服务通信主要的性能瓶颈。在5.2.2小节展示了 FlatProxy 和之前的各类优化方式的性能比较，结果显示服务网格的硬件加速在延时、吞吐、资源使用方面都具有全面的优势。在5.2.3小节展示了两类通信方式的性能对比，可以看出使用紧耦合的方式能够稳定的带来一定的性能优势，但其需要对应用进行改造提供专用的通信接口。

第 6 章　工作总结与展望

本文对云原生场景中服务治理存在的挑战进行了研究探索。主要解决的问题是当前服务网格在云原生场景中无法对部署在非主机侧的服务进行有效监管，以及部署服务网格后系统性能严重下降、资源占用过高的问题。针对上述问题，本文提出了以“数据”为中心的服务网格架构，即在节点数据出入口网卡处完成节点内服务治理的任务。针对新的服务网格架构实现所需的硬件平台，本文从网络加速能力，互连能力，控制能力和可编程能力四方面调研了“近网计算”、SmartNIC 和 DPU。最终，选择了 DPU 做为新架构的实现平台。基于 DPU 的服务网格设计需要满足高带宽，多功能，性能隔离等需求。为此，本文提出了四种架构优化的方法，软硬协同，分类处理，硬件多线程和通信路径优化。基于该架构设计，本文介绍了基于驭数 K2-Pro 实现的虚拟网络和服务网格的实例。最后基于 K2-Pro 原型验证平台的实验验证显示服务网格的卸载加速能够在系统性能和资源使用方面取得较为明显的优势。

在上述工作中，本研究完成了初步基于 DPU 的服务网格设计和验证，但针对实际服务网格系统的实现还存在很多待解决的问题。例如，主机侧多线程服务如何与卸载后的服务网格高效通信在本工作中还没有进行探究，卸载后的多种功能如何进行数据级和任务级的并行调度还需近一步探索，在实验中存在的一些反常之处可能与主机侧的内存管理等机制的影响有关也需在下一步工作中查找具体原因。除此之外，当前的设计主要还是针对服务网格数据面流量控制相关的内容，在可观测性和安全性的研究也是未来工作的研究重点。

参考文献

[1] CNCF. Cncf projects are the foundation of cloud native computing [EB/OL]. 2023. https://www.cncf.io/.

[2] Bushong V, Abdelfattah A S, Maruf A A, et al. On microservice analysis and architecture evolution: A systematic mapping study [J]. Applied Sciences, 2021, 11(17): 7856.

[3] Dragoni N, Giallorenzo S, Lafuente A L, et al. Microservices: Yesterday, today, and tomorrow [M/OL]. Cham: Springer International Publishing, 2017: 195-216. https://doi.org/10.1007/978-3-319-67425-4_12.

[4] Söylemez M, Tekinerdogan B, Kolukısa Tarhan A. Challenges and solution directions of microservice architectures: A systematic literature review [J/OL]. Applied Sciences, 2022, 12 (11). https://www.mdpi.com/2076-3417/12/11/5507. DOI: 10.3390/app12115507.

[5] James Lewis M F. Microservices [EB/OL]. 2014. https://martinfowler.com/articles/microservices.html.

[6] Niknejad N, Che Hussin A R, Prasetyo Y A, et al. Service oriented architecture adoption: A systematic review [J/OL]. International Journal of Integrated Engineering, 2018, 10(6). https://publisher.uthm.edu.my/ojs/index.php/ijie/article/view/2821.

[7] Niknejad N, Ismail W, Ghani I, et al. Understanding service-oriented architecture (soa): A systematic literature review and directions for further investigation [J/OL]. Information Systems, 2020, 91: 101491. https://www.sciencedirect.com/science/article/pii/S0306437920300028. DOI: https://doi.org/10.1016/j.is.2020.101491.

[8] Indrasiri K, Siriwardena P. Microservices governance [M/OL]. Berkeley, CA: Apress, 2018: 151-166. https://doi.org/10.1007/978-1-4842-3858-5_6.

[9] linkerd/linkerd2:ultralight, security-first service mesh for kubernetes. main repo for linkerd 2.x. [EB/OL]. 2022. https://github.com/linkerd/linkerd.

[10] Calcote L. The enterprise path to service mesh architectures [M]. O'Reilly Media, Incorporated, 2020.

[11] Google. istio/community:istio governance material [EB/OL]. 2022. https://github.com/istio/community.

[12] Fang K, Peng D. Netdam: Network direct attached memory with programmable in-memory computing isa [J]. arXiv preprint arXiv:2110.14902, 2021.

[13] Tork M, Maudlej L, Silberstein M. Lynx: A smartnic-driven accelerator-centric architecture for network servers [C]//Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020: 117-131.

[14] Choi S, Shahbaz M, Prabhakar B, et al. λ-nic: Interactive serverless compute on programmable smartnics [C]//2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2020: 67-77.

[15] Li W, Lemieux Y, Gao J, et al. Service mesh: Challenges, state of the art, and future research opportunities [C/OL]//2019 IEEE International Conference on Service-Oriented System Engineering (SOSE). 2019: 122-1225. DOI: 10.1109/SOSE.2019.00026.

[16] Du D, Liu Q, Jiang X, et al. Serverless computing on heterogeneous computers [C/OL]// ASPLOS '22: Proceedings of the 27th ACM International Conference on Architectural Sup-

port for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2022: 797–813. https://doi.org/10.1145/3503222.3507732.

[17] Nvidia. Developing a linux kernel module using gpudirect rdma [EB/OL]. 2023. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[18] Ma Xiaoxiao W Z Y G A X, Yang Fan. Survey on smart network interface card [J/OL]. Journal of Computer Research and Development, 2022, 59(1640940039501-65674089): 1. https://crad.ict.ac.cn/en/article/doi/10.7544/issn1000-1239.20200629.

[19] Edouard Bugnion J N, Tsafrir D. Hardware and software support for virtualization [M]. Morgan 8 Claypool Publishers, 2017.

[20] Balaji P, Narravula S, Vaidyanathan K, et al. Sockets direct protocol over infiniband in clusters: is it beneficial? [C/OL]//IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004. 2004: 28-35. DOI: 10.1109/ISPASS.2004.1291353.

[21] Goldenberg D, Dar T, Shainer G. Architecture and implementation of sockets direct protocol in windows [C/OL]//2006 IEEE International Conference on Cluster Computing. 2006: 1-9. DOI: 10.1109/CLUSTR.2006.311918.

[22] Cloud native computing foundation (“cncf”) charter [EB/OL]. 2023. https://github.com/cncf/foundation/blob/main/charter.md.

[23] Rob Davis S, Mellanox Ilker Cebeli. Accelerating nvme™ over fabrics with hardware offloads at 100gb/s and beyond [EB/OL]. https://nvmexpress.org/wp-content/uploads/Accelerating-NVMe-over-Fabrics-with-Hardware-Offloads.pdf.

[24] Moon Y, Lee S, Jamshed M A, et al. Acceltcp: Accelerating network applications with stateful tcp offloading. [C]//NSDI: volume 20. 2020: 77-92.

[25] Morgan W. The service mesh [EB/OL]. 2022. https://buoyant.io/service-mesh-manifesto/.

[26] Google. Pod [EB/OL]. 2023. https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/.

[27] Envoy. envoyproxy/envoy:cloud-native high-performance edge/middle/service proxy [EB/OL]. 2022. https://github.com/envoyproxy/envoy.

[28] Cilium. ebpf-based networking, observability, security [EB/OL]. 2022. https://cilium.io/.

[29] Cloud A. microservices gateway [EB/OL]. 2022. https://www.aliyun.com/product/aliware/mse.

[30] 阿里云. 微服务用户为什么需要云原生网关 [EB/OL]. 2022. https://segmentfault.com/a/1190000041462841.

[31] 阿里云. 云原生网关支持 TLS 硬件加速 [EB/OL]. 2022. https://developer.aliyun.com/article/870630.

[32] IV F F K. State of vpp in network service mesh [EB/OL]. 2019. https://www.youtube.com/watch?v=-_-dNMOoCjc.

[33] Warnicke E. Vpp/envoy integration: Improving the istio service mesh [EB/OL]. 2018. https://www.youtube.com/watch?v=FtuCnBVZs-E.

[34] F-Stack. Faster than the fastest [EB/OL]. 2023. http://www.f-stack.org/.

[35] Group A F S. Sofastack [EB/OL]. 2022. https://www.sofastack.tech/projects/.

[36] huawei. Mesher [EB/OL]. 2022. https://github.com/apache/servicecomb-mesher.

[37] The fungible dpu™: A new category of microprocessor for the data-centric era : Hot chips 2020 [C/OL]//2020 IEEE Hot Chips 32 Symposium (HCS). Los Alamitos, CA, USA: IEEE Computer Society, 2020: 1-25. https://doi.ieeecomputersociety.org/10.1109/HCS49909.2020.9220423.

[38] Burres B, Daly D, Debbage M, et al. Intel' s hyperscale-ready infrastructure processing unit (ipu) [C/OL]//2021 IEEE Hot Chips 33 Symposium (HCS). 2021: 1-16. DOI: 10.1109/HCS52781.2021.9567455.

[39] Burstein I. Nvidia data center processing unit (dpu) architecture [C/OL]//2021 IEEE Hot Chips 33 Symposium (HCS). 2021: 1-20. DOI: 10.1109/HCS52781.2021.9567066.

[40] Galles M, Matus F. Pensando distributed services architecture [J/OL]. IEEE Micro, 2021, 41 (2): 43-49. DOI: 10.1109/MM.2021.3058560.

[41] Ltd Y T C. The technical white paper of data processing unit [EB/OL]. 2021.

[42] Service A W. Aws nitro system [EB/OL]. 2022. https://aws.amazon.com/ec2/nitro/.

[43] Lin J, Patel K, Stephens B E, et al. PANIC: A High-Performance programmable NIC for multi-tenant networks [C/OL]//14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 2020: 243-259. https://www.usenix.org/conference/osdi20/presentation/lin.

[44] Yang M, Baban A, Kugel V, et al. Using trio: Juniper networks' programmable chipset - for emerging in-network applications [C/OL]//SIGCOMM '22: Proceedings of the ACM SIGCOMM 2022 Conference. New York, NY, USA: Association for Computing Machinery, 2022: 633–648. https://doi.org/10.1145/3544216.3544262.

[45] Cloud G. Expanding the tau vm family with arm-based processors [EB/OL]. 2022. https://cloud.google.com/blog/products/compute/tau-t2a-is-first-compute-engine-vm-on-an-arm-chip.

[46] Eric Garver Y F, Rashid Khan. Optimizing server utilization in datacenters by offloading network functions to nvidia bluefield-2 dpus [EB/OL]. 2021. https://www.redhat.com/zh/blog/optimizing-server-utilization-datacenters-offloading-network-functions-nvidia-bluefield-2-dpus.

[47] Ibanez S, Mallery A, Arslan S, et al. The nanopu: A nanosecond network stack for datacenters [C/OL]//15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). USENIX Association, 2021: 239-256. https://www.usenix.org/conference/osdi21/presentation/ibanez.

[48] Lazarev N, Xiang S, Adit N, et al. Dagger: efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics [C]//Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021: 36-51.

[49] Pismenny B, Eran H, Yehezkel A, et al. Autonomous nic offloads [C/OL]//ASPLOS '21: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2021: 18–35. https://doi.org/10.1145/3445814.3446732.

[50] Arslan S, Ibanez S, Mallery A, et al. Nanotransport: A low-latency, programmable transport layer for nics [C/OL]//SOSR '21: Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR). New York, NY, USA: Association for Computing Machinery, 2021: 13–26. https://doi.org/10.1145/3482898.3483365.

[51] Arashloo M T, Lavrov A, Ghobadi M, et al. Enabling programmable transport protocols in High-Speed NICs [C/OL]//17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). Santa Clara, CA: USENIX Association, 2020: 93-109. https://www.usenix.org/conference/nsdi20/presentation/arashloo.

[52] Pan T, Yu N, Jia C, et al. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches [C]//Proceedings of the 2021 ACM SIGCOMM 2021 Conference. 2021: 194-206.

[53] Kanev S, Darago J P, Hazelwood K, et al. Profiling a warehouse-scale computer [C]// Proceedings of the 42nd Annual International Symposium on Computer Architecture. 2015: 158-169.

[54] 阿里云. 阿里云虚拟化 Enclave(基于第三代神龙架构) [EB/OL]. 2023. https://www.aliyun.com/daily-act/ecs/aliyun-enclave.

[55] Bosshart P, Daly D, Gibb G, et al. P4: Programming protocol-independent packet processors [J/OL]. SIGCOMM Comput. Commun. Rev., 2014, 44(3): 87–95. https://doi.org/10.1145/2656877.2656890.

[56] Li Z, Guo L, Cheng J, et al. The serverless computing survey: A technical primer for design architecture [J]. ACM Computing Surveys (CSUR), 2022, 54(10s): 1-34.

[57] 张磊. 深入剖析 Kubernetes [M]. 人民邮电出版社, 2021.

[58] Iperf. iperf3: A tcp, udp, and sctp network bandwidth measurement tool [EB/OL]. 2022. https://github.com/esnet/iperf.

[59] Mellanox. Mellanox/sockperf: Network benchmarking utility [EB/OL]. 2022. https://github.com/Mellanox/sockperf.

[60] Fortio. fortio [EB/OL]. 2022. https://github.com/fortio/fortio.

致 谢

三年前，我怀着求知的目的选择了这趟旅途。这段旅途于我而言并不容易，不过所幸当初的目标虽有遗憾但大都完成。在这段旅途的终点回忆过往，非常感谢这一路上陪伴的每个人，正是你们使我不断认识到自身的不足和局限，取得一次次的进步。在此，我希望能以寄语的方式聊表谢意，并送上我对大家诚挚的祝福。

首先感谢鄢老师和卢老师三年中的言传声教。鄢老师很严肃，很有智慧，很认真。这三年中，您的每一次的对话都使我受益匪浅，也是您的严格要求使我不断进步。您的勤勉认真将是我永远学习的对象。卢老师在最初的时光对我照顾有加，在我最迷茫的时候给与我鼓励与指引，很有好大哥的风范，今后也请多多指教。

接下来，感谢同期的三位好兄弟。三年里，我觉得樊博和我最合拍，我们一起战斗过，算是穿过一条裤子的人了吧。他很适合做一个组织者，我就不太行，我总是没啥太充足的动力去做这些事，但在最关键的时候你会有点想退缩，希望以后改正哟。以后要好好加油，顺利毕业。廖神科研大佬，很自律也很有韧劲，目标总是很清晰，我想这也是鄢老师很看重你的原因吧。在这方面，我是真的和你学习到很多，但不得不承认我还是欠缺不少，我总是会被莫名的“人生意义不明朗”所束缚。祝你早日达成自己的愿望成为一个很牛的学者。宇军生活小达人，能够感受的到你对学术没有太高的追求，但人生从来就不是一定要追寻什么高大上的目标不可，过好自己的生活才是最重要的。以后，如果被生活毒打了也请不要放弃自己对生活的追求。

另外，十分感激各位师兄、师姐、师弟和师妹的帮助。孔师兄很有亲和力，第一次见面感觉就很有好感，希望以后我们还能常联系。师姐的人设有点神秘，一直想多了解一下但都没什么机会。感谢师姐最后这段时间的辛苦付出，今后要 open 一点呀。赵师兄，人生赢家，球打的好，桌游玩的好，还有漂亮的女朋友，羡慕啦。不过以后要努力加油，多发论文。涵越要有斗志一点，人生还是要冲起来，只争朝夕。丽云也要加油，今后要找到自己喜欢的路，努力的走下去。

最后，感谢多年来无私支持我的父亲、母亲，请不要为我担心，你们将是我人生永远的港湾。

人生路漫漫兮，很高兴也很感谢这一段旅途有你们相伴，我相信每个人都值得铭记，都是人生中的宝贵一页。祝大家今后一切安好，愿来日再见，你我初心不改，仍有不惧困难，奋斗前进的勇气和动力。

2023 年 6 月

作者简历及攻读学位期间发表的学术论文与研究成果

作者简历:

李明，1995 年 8 月生，籍贯四川省南部县。

2014 年 9 月至 2015 年 6 月，天津大学软件学院在读。

2015 年 9 月至 2017 年 6 月，66028 部队服役。

2017 年 9 月至 2020 年 6 月，天津大学智算学部，取得工学学士学位。

2020 年 9 月至 2023 年 6 月，中国科学院计算与技术研究所，导师鄢贵海研究员，计算机系统结构硕士研究生。

已发表（或正式接受）的学术论文:

申请或已获得的专利:

专著:

- 中国科学院计算技术研究主编，中科驭数，中国计算机学会集成电路设计专业组《专用数据处理器（DPU）技术白皮书》
- 中国科学院计算技术研究主编，中科驭数，中国计算机学会集成电路设计专业委员会，中国计量测试学会集成电路测试专业委员会《专用数据处理器（DPU）性能基准评测方法与实现》

参加的研究项目及获奖情况:

参加的研究项目:

- 国家自然科学基金项目 (61872336)，“软件定义计算流图专用处理器体系结构研究”

获奖情况:

- 2021 年　CCC 挑战赛亚军
- 2021 年　所长优秀奖