分类号_____ 学号 M201472689

学校代码 10487 密级_____

华中科技大学

硕士学位论文

基于 DPDK 的三层转发技术研究

学位申请人： 孙 贻 妙

学 科 专 业： 计算机技术

指 导 教 师： 刘 景 宁 教授

答 辩 日 期： 2016 年 5 月 29 日

A Thesis Submitted to Academic Evaluation Committee of Huazhong University of Science and Technology for the Degree of Master of Engineering

The Research of DPDK-based Layer-3 Forwarding Technology

Candidate : Sun Yimiao

Major : Computer Technology

Supervisor : Liu Jingning

Huazhong University of Science and Technology

Wuhan, Hubei 430074, P. R. China

May29,2016

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　年　月　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本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和汇编本学位论文。

本论文属于　保密□，在_____年解密后适用本授权书。

不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　年　月　日　　日期：　年　月　日

摘要

随着 Intel QuickAssist 技术，PCI-E，QPI 等接口及多核 CPU 架构的发展，通用服务器硬件的 I/O 性能得到极大的提升。同时，网络接口速度也得到飞跃式的提升，以太网卡的带宽从 1000Mbit/s 发展到了 10Gbit/s 及 40Gbit/s，甚至已经有相关企业和研究机构在研究 100Gbit/s 的网卡相关技术。鉴于通用服务器硬件性能的提升，在通用服务器进行网络设备开发与研究，成为工业界和学术界的研究热点。现有 Linux 内核的网络 I/O 存在诸多性能瓶颈，不能充分发挥现代数据中心多核服务器的硬件能力。于是，英特尔开发了数据平面开发套件 DPDK，帮助用户开发高性能的网络应用，日益得到业界的认可。

从 Linux 内核网络栈存在的问题入手，分析了 DPDK 的架构的优点。通过 DPDK 提供的库，结合 DPDK 的特性，实现了基于 DPDK 的三层转发系统 D-L3 Forwarding，为网卡端口建立多个接收包与发送包队列，将收发包队列绑定在不同逻辑 CPU 核，在每个逻辑 CPU 核上对所绑定的收发队列的 IPv4 包进行并行处理，采用 SSE 指令集批量处理包，使系统具有高性能的包转发能力。结合三层转发系统及 DPDK 原理，提出可能对系统性能产生影响的因素。

通过对比实验表明，在 10Gbit/s 网卡环境下，D-L3 Forwarding 系统能够对 IPv4 包进行高效的转发。系统对 NUMA 架构的配置会对转发性能产生较大的影响，而内存通道数与巨页大小的变动并不会影响到 10Gbit/s 网卡环境下系统的性能。

关键词：数据平面开发套件，三层转发，非统一内存访问，网络功能虚拟化

Abstract

With the emergence of Intel QuickAssist Technology, PCI-E, QPI, etc., the I/O capacity of general purpose servers has a significant improvement. At the same time, the network adaptor's speed increases from 1000Mb/s to 10Gb/s as well as 40Gb/s. Even some research institutions and enterprises are trying to do some research on the 100Gb/s technology. The study of network applications with commodity servers become an advanced research hotspot. The current kernel-based solution for network packet processing has several bottlenecks, which limit the performance of servers. Considering this, Intel develops a tool called Data plane Development Kit to bypass the Linux kernel network stack so that the programmer can develop high performance network applications by using DPDK.

This paper is based on DPDK and aims at designing a L3 forwarding system based on DPDK. At first this paper will introduce the background of why DPDK is developed and some researches related to DPDK from home and abroad in detail. Then we will analysis the Linux kernel network stack and find out some performance limiting factors on it. After that we introduce DPDK and analysis its architecture, and also introduce the features of DPDK. We will then implement the L3 forwarding system named D-L3 Forwarding for fast IPv4 packets forwarding. D-L3 Forwarding system implements multiple receive and send queues for each NIC port and bind each of them to a single logic core for parallel IPv4 packets processing. We also use SSE and batch processing to improve system's performance. After that we will analysis some critical factors that may have effect on system's performance.

Finally we run D-L3 Forwarding system and do tests on it . The tests has shown that D-L3 Forwarding gets a very fast packet forwarding in case of 10Gb/s NIC. Tests also prove that NUMA can have a critical effect on system's performance, while huge page size and the number of memory channels can't influence system's performance.

Keywords：Data Plane Development Kit, Layer-3 Forwarding, Non-Uniform Memory Access, Network Function Virtualization

目 录

1 绪论

1.1 研究背景

随着计算机系统结构的发展，全球信息化进程突飞猛进，大量的信息在计算机网络上生成和传输。为此，数据中心对海量数据的存储与大数据的处理成为研究热点。分布式计算与分布式存储技术给大数据的处理与海量数据的存储带来希望，同时对数据中心计算机网络需求越来越高。为了满足计算机应用的需求，近年来计算机相关硬件性能不断的提升。例如，随着 CPU 架构的不断改进，现代数据中心服务器使用多 CPU、多核的架构，应用程序能同时在多 CPU 或多个核上并行运行，大大加速了应用程序对信息数据的处理速度。网络接口速度也得到飞跃式的提升，Ethernet 网卡从 1000Mb/s，10Gb/s[1]，到如今的 40Gb/s。甚至已经有相关企业和研究机构在研究 100Gb/s 的网卡相关技术。

2008年，计算机巨头Intel 公司发布白皮书《Design Patterns for Packet Processing Applications on Multi-core Intel® Architecture Processors》，拉开了基于通用计算机硬件设计的网络设备开发的大幕。尽管如此，NUMA（Non Uniform Memory Access Architecture）作为现代数据中心服务器所采用的主流CPU架构[2]，给应用程序的并行处理带来挑战。许多高性能计算研究机构或者大数据社区都对NUMA架构做了一系列的研究测试[3][4][5][6]，这些研究测试均表明，若不能合适的利用NUMA架构，那么应用的运行性能将会大大的降低。此外，传统的基于内核的网络处理技术无法满足数据传输与处理的需求[7]。为此，英特尔提出了数据平面开发套件DPDK（Data Plane Development Kit）[8]。通过UIO（Userspace I/O）[9]，轮询[10]，CPU亲合等技术[11]，DPDK可以让用户在单个英特尔至强处理器[12]上获得超过80Mpps的吞吐量。同时，DPDK让用户可在处理数据包的同时通过英特尔或其他处理器执行其它工作负载，从而降低硬件成本、简化应用程序开发环境和缩短上市时间。DPDK可以在软件定义网络（Software Defined Network，SDN）[13]及网络功能虚拟化（Network Function Virtualization，NFV）[14]下发挥重要作用，对比传统的linux网络栈，DPDK出色的数据面性能优化能力，以及与通用计算平台优秀的契合度，使之迅速成为众多SDN和NFV开发者或厂商用以优化自身技术和方案的关键技术，也成为了基于通用计算机

平台网络应用的热门解决方案。

本文以三层转发应用作为典型进行研究，设计与实现了D-L3 Forwarding系统。D-L3 Forwarding基于Intel DPDK套件进行开发，并使用万兆（10Gb/s）以太网卡，在多核、多处理器平台上进行测试研究。此外，本文描述了D-L3 Forwarding在通用服务器环境下（10Gb/s网卡+NUMA多核架构）的性能测试方案以及测试结果。测试结果表明，D-L3 Forwarding能高效的转发IP包，达到10Gb/s的带宽。测试结果表明，一些典型配置，如NUMA多核的分配等，会影响D-L3 Forwarding系统的性能。

1.2 国内外研究现状

DPDK 于 2010 年 9 月 17 日正式发布，并在 2013 年 4 月，DPDK 脱离英特尔，成立了一个开源社区，成为一个独立的开源项目。DPDK 包含了一系列的驱动及库，目的是为了更高效的执行包处理。目前，DPDK 已经能够支持 X86，IBM Power 8[15]，EZchip 的 TILE-Gx[16]和 ARM[17]等架构处理器。

由于 DPDK 能够对网络包进行高效的处理，DPDK 开源库，得到越来越多的生态系统合作伙伴的支持与认可，如华为、中兴、Wind River、 6wind、Xilinx、 Tieto、Netronome 和 Cavium 等都对 DPDK 表示支持，并将针对 DPDK 进行开发工作。在 DPDK 版本的开发之中，越来越多的来自华为、英特尔、IBM、VMware、红帽、思科、6wind、博科、飞思卡尔、EZchip、Mellanox 等公司的程序员对源代码提供支持。在 DPDK 优秀的应用价值之下，许多产品也都考虑结合 DPDK，来提升整体的系统性能。

Open vSwitch（简称 OVS）[18]是在 Apache2.0 许可下开发的一个产品级的，多层次的虚拟交换机，它的设计目的是通过可编程扩展，实现大量的网络自动化，同时支持标准的管理接口和协议，比如 NetFlow，sFlow，SPAN，RSPAN，CLI，LACP，802.1ag，OpenFlow[19]等。在 2.2 版本的 Open vSwitch 当中，提供了 DPDK 的支持[20]。而在 Open vSwitch 的 2.4 版本当中，新增了使用 DPDK 优化的 vHost 路径的功能。在 OVS 当中，一些热点区域的性能问题也能通过 DPDK 的库函数得到优化。通过 DPDK，可以优化转发平面，使转发平面能够在用户空间使用 vswitch 后台程序的单独的线程去执行。通过实现 DPDK 优化过的 vHost 客户机接口，也能使得虚拟机跟虚拟机或者物理机与物理机之间的使用案例变得更加高效。

相比传统机械硬盘，固态硬盘极大地降低了延迟，提高了 IO 的读写速度[21]。随着对非易失存储（Non-Volatile Memory，NVM）的研究的深入[22][23][24]，新型存储介质有可能带来接近内存的访问速度。为此，存储设备逐渐摆脱之前 I/O 性能瓶颈的标签。同时相关 I/O 软件也有待改进和提高。为了满足未来存储应用的快速访问需求[25][26]，提高存储设备硬件利用率，英特尔开发了 SPDK(Storage Performance Development Kit)[27]，提供一整套端到端的存储参考架构。SPDK 的目标是把硬件平台的计算、网络、存储的性能充分发挥出来。

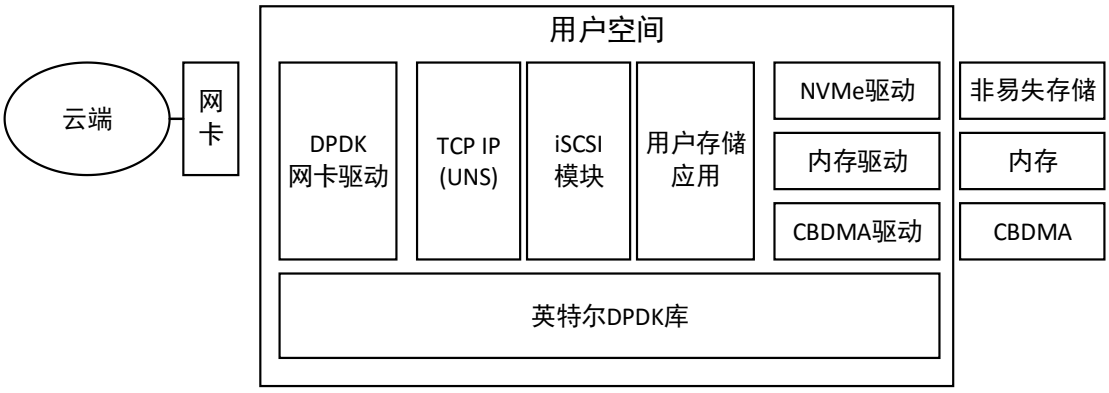

图 1.1 SPDK 架构图

图 1.1 显示的是 SPDK 的架构图，从图 1.1 可以看出，整个 SPDK 架构基于英特尔的 DPDK 库开发。通过 DPDK 库和 user space NVMe 设备驱动，将网卡 I/O 和存储 I/O 处理完全在用户态运行，完全绕过内核，大大提高整体性能和效率。此外，对应的服务均运行与用户态，如用户态 TCP/IP 处理替代 Linux 内核 TCP/IP 协议栈，以避免使用传统 TCP/IP 协议栈的所来的性能限制瓶颈。总之，SPDK 提供了一套 API 框架，让厂商能够插入自己定义的处理逻辑（即图 1.1 中的用户存储应用模块）。通过这种机制，用户可在用户态运行例如缓存、压缩、加密、去重、RAID 计算，或擦除码(Erasure Coding)计算等功能。

作为英特尔在中国的首批 DPDK 合作伙伴之一，XSKY(星辰天合)团队率先将 DPDK 与 Ceph 用户态文件系统 BlueFS 整合[32]，有效地利用了 NVM 设备的存储性能，大幅度提高 Ceph 在 NVMe 介质上的读写效率。下一步，XSKY 将发起针对整个 Ceph OSD 的重构，全面面向 SPDK 进行优化并接纳英特尔 DPDK 作为网络层交互方案，性能有望得到进一步提高。

综上，DPDK 因为其在处理网络包上的高性能及其通用性，引起了广大 SDN、NFV 以及分布式存储应用开发者的高度关注，表明未来 DPDK 具有更广泛的使用价值。

除 DPDK 之外，也存在一些加速包处理的方案，比如 netmap [33]与 PF ring。Netmap 采取了数据批处理及 mmap 等技术来解决内核包处理时的动态内存分配，系统调用以及内存拷贝所带来的开销。而 PF ring 用于高速执行对包的抓取、过滤和分析操作。PF ring 采用 Linux 的 NAPI 驱动来轮询网卡得到包，并将其拷贝到一个环形缓冲区中，然后通过另一个轮询把环形缓冲区的内容读取到用户态程序当中。PF ring 可以有多个并行环形缓冲区，这让多个 CPU 并行获得包数据。

现在也有基于 GPU 硬件网络包处理加速的方案，比如 PacketShader[36]。Packetshader 实现了一个高度优化的 IO 处理引擎，使用批量处理多技术减小包的平均处理开销，同时利用 GPU 的并行处理能力，以达到高效地网络包处理。总之，网络 I/O 处理加速已经成为提高系统性能的一个关键点，今后在各领域应用系统中也将会有广泛的应用，尤其是基于通用服务器的网络设备或 NVF 等。

1.3 论文主要工作及组织结构

本文研究分析了 DPDK 的基本原理与架构，设计并实现了基于 DPDK 的三层转发系统（D-L3 Forwarding）。同时，分析了影响 D-L3 Forwarding 的主要性能因素。并设计相应的实验，实验表明表明该系统的可行性和高性能，能轻易达到最高带宽。同时，NUMA 与多核的任务分配等因素会影响该系统的性能。

文章总共有五个章节，其简要概述分别如下。

第一章介绍了课题的研究背景及国内外研究现状，说明了 DPDK 在如今 SDN 和 NFV 领域以及分布式存储领域上的作用，具有广泛的应用价值。

第二章分析了主要分析 DPDK 的架构和原理，主要与传统内核对比，描述 DPDK 的创新之处。

第三章详细描述 D-L3 Forwarding 系统的设计和实现，并分析可能影响 D-L3 Forwarding 系统性能的一些因素。

第四章设计多组对比实验，对系统进行了对比测试。包括测试环境和测试过程，并根据测试结果，对其进行了各自的分析。

第五章对全文进行了总结，再一次论述了文章的主旨和研究内容，并论述了系统中待研究及完善的地方，并对未来的工作进行了展望。最后是致谢和参考文献。

2 Linux 网络栈及 DPDK 架构

DPDK 提供了一个基本框架，能更快地执行包的收发处理，让开发者可以快速地开发出高性能的网络应用系统。本章将对比 Linux 内核网络处理，分析研究 DPDK 的基本原理和架构。

2.1 Linux 网络栈

Linux 网络栈提供了一系列通用的网络接口，接口范围从协议无关层到各种网络协议的具体层。本节将先分析内核从网卡接收/发送处理网络包的流程，之后会分析 Linux 内核网络栈性能的瓶颈因素。

2.1.1 网卡收发包流程

网络之间机器数据通信的基本单位是包，数据被封装成一个个包在机器间进行传输[37]。图 2.1 描述了包的收发流程，具体解释如下。

1. 网卡初始化后，会有准备对应的 DMA buffer，是个环形队列，称作环形缓冲区（ring buffer）。这一环形缓冲区的作用主要用于保存指向 sk_buff 结构的指针描述符，一个网卡通常会有两类环形缓冲区，一个用于包的接收，另一个用于包的传输；当收到一个包的时候，这些指针描述符会被使用，通过描述符初始化一个 sk_buff，为接收到的包分配内存空间。网卡可以通过 DMA 操作，来实现包从网卡到内存的传递。当上述都成功的时候，一个包便被 sk_buff 结构成功表示并准备好进行下一步处理（传递给上层软件处理），否则这个包会因为缺少包描述符被网卡丢弃[40]；

2. 之后便以 NAPI（New API）[38][39]中断处理的方式，通知内核已经有包被网卡接收到。

3. CPU 在收到中断之后会调用中断处理程序去响应这个中断。中断处理程序会把网卡添加到一个 poll_list 列表当中，并执行一个软中断的调度。当这个中断被处理时，CPU 会轮询每个在 poll_list 表中的设备，用以从环形缓冲区中去除接收到的包。之后，对于每个被获取的包，内核会调用一个函数传递给内核网络协议栈；

4. 网络栈会从网络层开始，一层一层地处理每个接收到的包。首先要执行一些基本的检查工作，这包括完整性检查以及防火墙安全规则的检查。如果包是非法的

话，那么将会被丢弃，否则包将会被发送到路由子系统。路由子系统会决定一个包是被发送到本机用户空间的程序进行使用，还是被直接转发到另一台主机上。这些决定的产生会基于已经部署的路由算法，在路由表内进行查找，寻找出目的 IP 地址的最佳的匹配项；

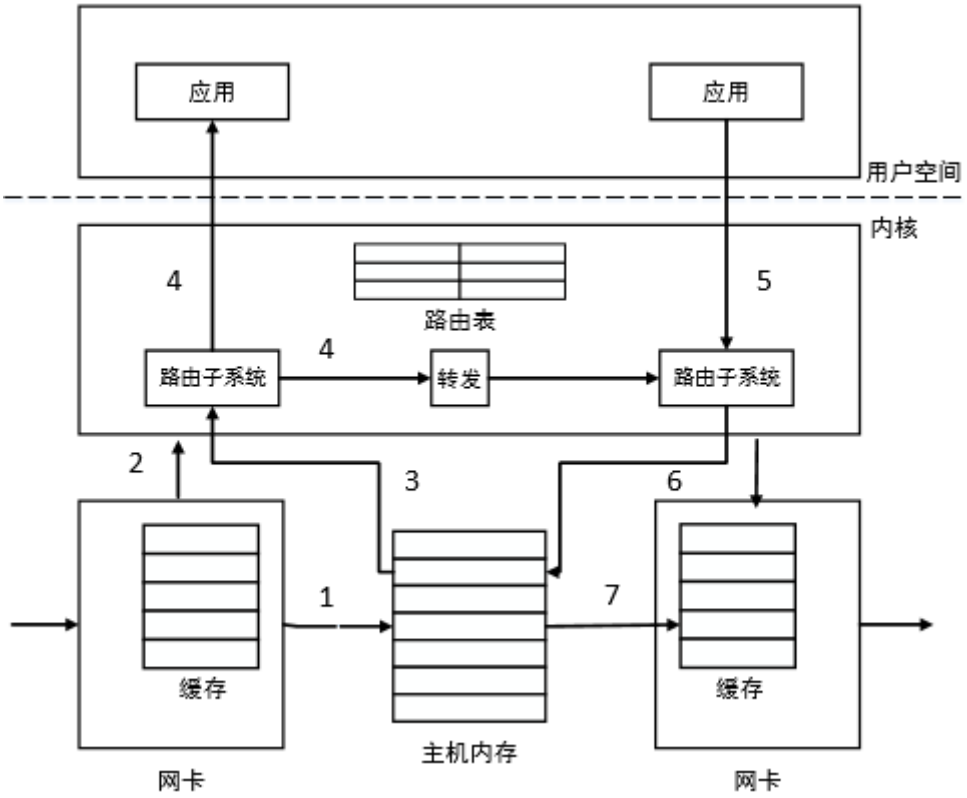

图 2.1 Linux 网络栈的包处理流程

5. 如果包的 IP 地址符合本地所配置的某个 IP 地址，这个包会在本地传递，进入到传输层。Linux 的网络栈提供了 TCP 跟 UDP 协议的完整实现。在经历了传输层之后，这些包便可以最后传递给上层的应用程序。通过使用 socket 接口，可以把 sk_buff 的数据拷贝到用户空间。到这一步之后，应用程序便有了对所收到的包的完全访问权限；应用程序为了发送包，也可以将包传递到网络栈，这依然会通过查找路由表来获得目的 IP 地址的下一跳目标；如果一个包是要直接转发到其他主机，那么这个包不会经过网络层之上的其他层；

6. 为了将一个包转发到别的主机上，网络层的主要工作是减少包头的 TTL 域的

值，当该值变为 0 的时候，会丢掉该包，并且发送一个对应的 ICMP 信息。此外，可以接着执行一些完整性检查、安全性检查、以及分片处理。之后包便会被发送到网络栈的发送部分；对于二层包的处理并不会区分它是来自本地产生的包还是转发过来的包，会基于路由表的查找结果，通过使用 ARP 协议来得到下一跳的二层地址。

7. 在满足了以上条件之后，系统会调用一个函数，通知网卡可以开始传输包了。首先为了传输一个包网卡需要读取指示 sk_buff 在主存位置的包描述符，并放入发送端缓存中。之后便会通知网卡有可发送的包。最后网卡会通过一个中断通知 CPU 可以释放 sk_buff 结构了。

2.1.2 影响网络栈性能的因素

设计 Linux 内核的网络栈的主要目的为了实现通用的网络。它实现了很多功能，包括路由功能，以及不同网络层次下的各种协议（比如在网络层的 IPv4 和 IPv6 协议，以及在传输层的 TCP 与 UDP 协议）。这些设计在通常情况下，能够满足一般应用程序的使用，并给其带来便利，但是当网卡的性能从 1Gbit/s 飞跃到 10Gbit/s 的时候，操作系统处理包的速度就跟不上网卡的收包速度，于是开始大量的丢包。下面将会继续分析影响系统处理包性能的一些原因。

第一个原因是 CPU 方面的因素。一个 CPU 每秒只能执行有限的时钟周期。当包的处理流程越复杂时，CPU 在处理的时候就需要越多的时钟周期，这意味着 CPU 每秒能处理包的数量会受到限制。因此为了达到更高的吞吐量，降低处理每个包的时钟周期是一个主要目标。

第二个限制网络栈性能的因素是对内存的使用。存在三个方面的问题：一，是包的内存分配和释放开销。对于每个包，它的 sk_buff 需要经历分配操作，并在之后这个包会被传递到用户态或者被转发出去，最后会被释放，这一过程需要浪费相当一部分的时钟周期。二，是 sk_buff 结构太复杂而庞大。Linux 网络栈想全面兼容各种网络协议所致。如图 2.2 所示，sk_buff 这个结构包含了一些协议的元数据，而这些元数据在不需要使用这些协议的时候是用不到的，这些元数据导致该数据结构变得庞大，因此减慢了包处理的速度。三，是存在多次的内存拷贝开销。包被应用程序使用前后，需要在不同的地址空间上（内核态和用户态）被处理，这导致了额外的内存拷贝。一次是在收包的时候，需要从 DMA 内存区域拷贝到 sk_buff 所在内存区域，另一次则是在应用程序获取包的时候，从有一次从 sk_buff 内核内存到用户态

内存的拷贝。相关研究资料表明，对于一个 64 字节的包来说，有 63%的 CPU 耗费在了对 sk_buff 内存的操作当中[41]。

第四个原因是系统调用开销。每当用户态的应用程序想要接收或者发送包的时候，都需要发起一个系统调用。这种设计增加了系统的安全性，降低了系统崩溃的概率，因为用户态应用不能直接使用重要的内核函数，但是这样做带来了巨大的 CPU 时钟开销。

```
/* Transport layer header */
union
{
        struct tcphdr *th;
        struct udphdr *uh;
        struct icmphdr *icmph;
        struct igmphdr *igmph;
        struct iphdr *ipiph;
        struct spxhdr *spxh;
        unsigned char *raw;
} h;

/* Network layer header */
union
{
        struct iphdr *iph;
        struct ipv6hdr *ipv6h;
        struct arphdr *arph;
        struct ipxhdr *ipxh;
        unsigned char *raw;
} nh;
```

图 2.2 sk_buff 部分结构图

第五个原因是在使用多 CPU 或者多核架构的时候，由于自旋锁的存在，对于包的处理也会变慢。这种等待机制的锁被 Linux 用于网络栈的许多地方。比如，若要发送一个包，那么需要两个锁，一个用于写，一个用于读，来保护网卡的发包队列，这便导致多 CPU 多核架构不能发挥并行处理的优势，性能达不到理想状况。

以上所提到的设计问题，使得 Linux 网络栈只能是一个通用的网络解决方案，但是并不能满足高速网络应用场景的解决方案。为此，DPDK 应运而生，旨在保证一定通用性的情况下，提供高速网络场景下网络包的处理。

2.2 DPDK 架构和原理

2.2.1 DPDK 组件分析

DPDK（Data Plane Development Kit）是英特尔公司发布的一款用于加速包处理的开源库。DPDK 由一系列的通用库和网卡驱动组成，兼容多种网卡以及多种 CPU 架构。DPDK 运行在 Linux 系统用户态，可以用于代替传统的 Linux 网络栈，执行高效的包处理。DPDK 的简要结构关系图如图 2.3 所示。用户空间的网络应用程序通过 DPDK 的库来进行包处理。在用户态和内核态之间有着一个 EAL（Environment Abstraction Layer）层[42]，这个层将 DPDK 的底层细节屏蔽，提供给上层一个通用的访问接口，并绕过 Linux 传统内核网络栈，通过自身的用户态网卡驱动，直接对网卡进行操作。

由图 2.3 可以看出 DPDK 的库实现了以下几个功能，并带来了如下优点。

1. 缓存管理：DPDK 会预分配固定大小的缓存到内存池上，这样做可以节约系统反复的分配和释放内存所需开销。

2. 队列管理：实现了无锁的队列，用于取代自旋锁，这允许多个应用同时进行包处理，而不用浪费时间在锁的等待上。

3. 流控制：与英特尔指令集结合，提出了一种在五元信息基础上的哈希方法，用于更快的处理包的转发操作，极大地提高了吞吐量。

4. 轮询模式驱动：DPDK 为百兆和万兆以太网卡提供了基于轮询的驱动。这样可以避免中断处理带来的开销，从而加速包的传递。

进一步的 DPDK 的组件构造如图 2.4 所示。由图 2.4 可以看出， DPDK 的组件主要可以分为以下几类。

1. 核心库：核心库为开发高性能的包处理程序提供了所有的元素。里面最重要的是 EAL 库，它负责对访问诸如硬件及内存空间这些底层资源，并向上层隐藏了底层的环境，为上层的程序及库提供了一个通用的接口。它负责初始化路径，决定内存空间，PCI 设备等资源的分配。而 MEMPOOL 库用于内存池管理，它负责为对象分配内存池。一个内存池由名称标识，并使用一个环结构来存储对象，此外还提供对象缓存及对齐工具。MALLOC 库用于内存分配，通过调用 MALLOC 库的接口，程序可以在巨页创建的内存区域中分配内存。RING 库用于环结构的管理，环结构提供了一个有限大小的无锁的生产者——消费者的 FIFO 队列。它有着无锁队列的优

点，并且容易实现，且支持批量操作。环可以作为 CPU 核之间的交流工具。MBUF 库用于管理包缓存，提供了创建和销毁缓存的功能，这些缓存可以用于存放应用程序的信息缓存（用于控制信息）或者是包缓存（用于存储网络包）。TIMER 库则是提供了时间服务，用于 DPDK 的执行单元，给予其异步执行函数的能力。

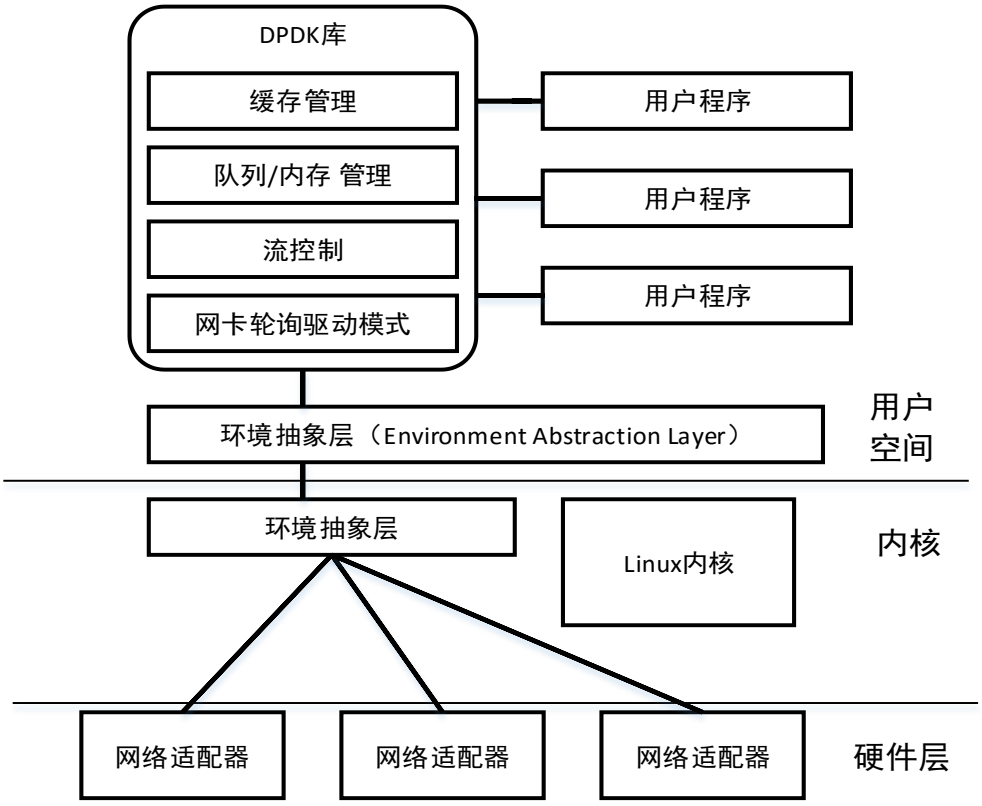

图 2.3 DPDK 简要结构关系图

2. 轮询驱动模块：该模块为多种网卡提供了基于轮询的用户态驱动。其中 ETHDEV 的作用类似于 EAL 库，它为上层应用提供通用的基于轮询的用户态驱动，方便了上层对网卡驱动进行调用。

3. 流分类模块：该模块为包的路由转发提供了一些库，包括 EXACT MATCH 库，它用哈希表来实现快速的查找，以及 LPM 库，它使用最长前缀匹配算法来决定路由路径。此外还有 ACL 库，用以实现一些流分类规则，达到访问控制的目的。

此外，在 Linux 内核 IGB_UIO 库借助 UIO 技术，用于将网卡绑定到 DPDK 的 IGB_UIO 模块，而并非是内核自带的网卡驱动。KNI（kernel NIC Interface）库则是

提供了一个 DPDK 到内核网络栈的接口，让 DPDK 用户可以继续使用的 Linux 内核网络栈及其功能。

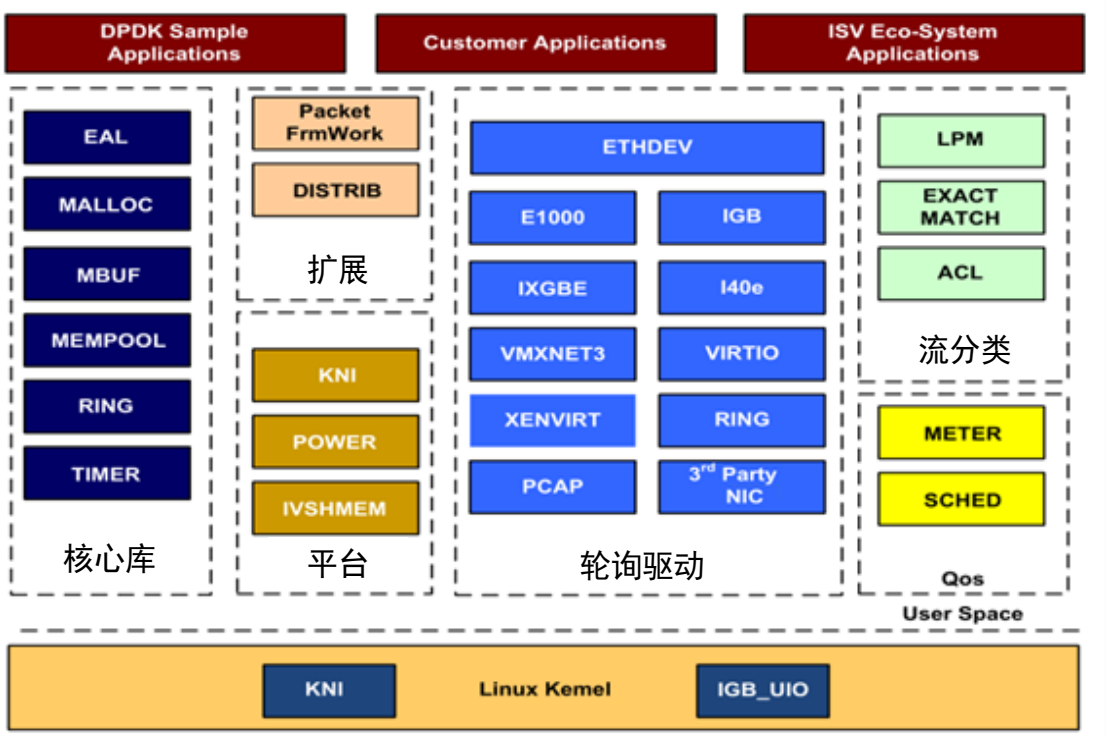

图 2.4 DPDK 组件图

2.2.2 DPDK 内存结构分析

为了管理各种类型的队列，RING 库（librte_ring）提供了环结构 rte_ring，如图 2.5 所示。该结构有如下特点。（1）环的大小是固定的，（2）是无锁的 FIFO 队列，（3）支持单个或者多个生产者——消费者的入队/出队场景，（4）此外，还支持一次性批量的包的入队/出队。该环保存的是一系列的指向对象的指针，并采用了两个指针对，用于分别描述生产者和消费者的队头和队尾。

相比于普通的队列，该 rte_ring 结构的设计有如下优点。（1）由于采用了无锁的设计，在对 ring 结构进行操作的时候可以避免锁带来的开销，获得更好的性能；（2）由于结构成员是指向对象的指针，可以容纳更多的成员，因此在批量操作的时候会产生较少的不命中。而其缺点在于由于其是固定大小的，总是要占据最大数量的空间，因此相比链表实现的队列而言会占据更多的内存。

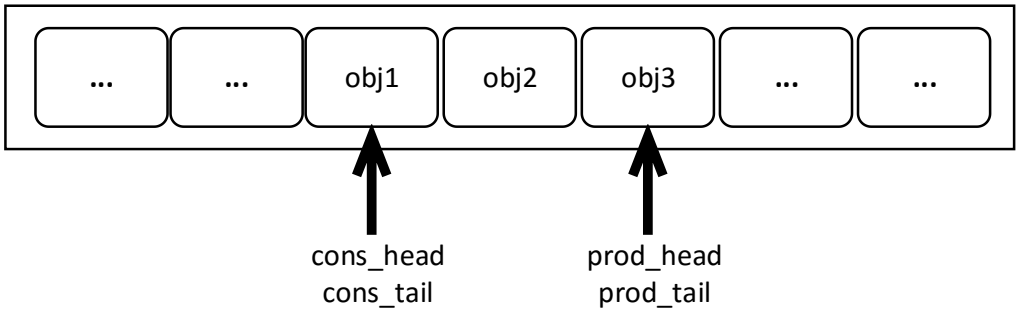

图 2.5 ring 结构

Rte_ring 结构可以用于在应用之间交流信息，也可以作为管理内存池的基本单元，或者用于保存包缓存。

EAL 层具有提供映射物理内存的功能。它会创建一个描述符表 rte_memseg，表中每个成员均指向一段用若干巨页表示的物理上连续的内存，这些连续的内存被称为内存段（Memory Segment），如图 2.6 所示。而这些内存段又被划分成内存区域（Memory Zone）。这些内存区域是应用及 DPDK 库所使用的基本内存单元。可以在内存区域上分配对象，这些对象可以是个环结构，或者是块内存池。

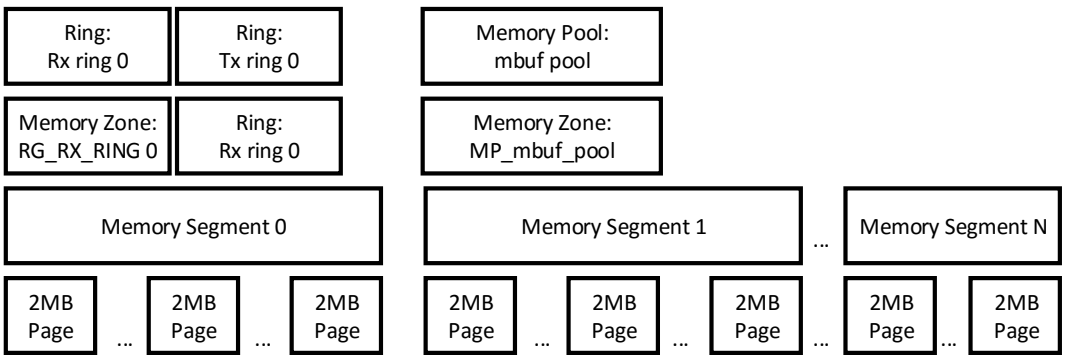

图 2.6 DPDK 内存管理图

分配的内存池对象在 DPDK 用 rte_mempool 结构表示，如图 2.7 所示。内存池在分配的时候已经固定大小，它可以用于存储包或者描述符。CPU 的核在内存负载平衡的情况下，可以通过条带化的方式，从不同的内存通道同时获取内存的数据来加快读取速度。

另外，为避免频繁执行 CAS（Compare And Set）无锁操作导致过大开销，每个核在 rte_mempool 结构内都有专属的本地 cache 并通过锁去访问这块区域。当该 cache

用满或者为空的时候，都会使用批量操作，来与内存池中的环结构交换对象。这样做可以减轻环的压力，并能更高效地使用环。

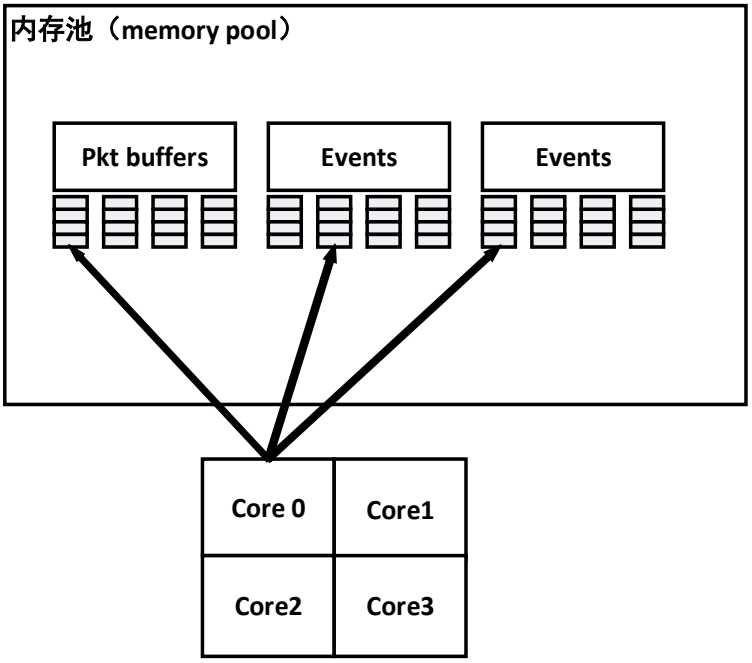

图 2.7 内存池结构

MBUF 库（librte_mbuf）为每个包提供了缓存，称为 rte_muf。这些缓存在系统开始运行之前就已被创建并被存储到内存池当中。在系统运行时，可以指定某一内存池，来使用该池中的 rte_mbuf 结构。当对该结构执行完毕，只需归还该结构，并不需要释放该空间。

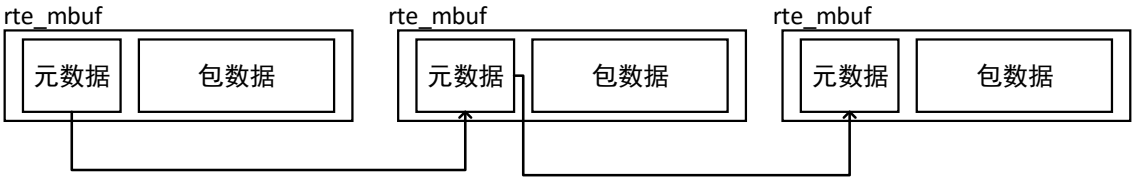

图 2.8 多个 rte_mbuf 结构链接以表示一个较大的包

一个用于描述包的 rte_mbuf 结构很小，它能够被直接放入一个缓存行（cache line）里面。一个 rte_mbuf 包括元数据（buffer 类型，长度，数据起始偏移，指向下个 buffer 的指针）及真实数据（包数据或者控制信息）。当要描述较大的包时，会

通过元数据成员 pkt.next 指向下一个 rte_mbuf 结构，通过该种方法便可以通过多个 rte_mbuf 结构共同表示一个包。图 2.8 描述的就是上述情况[43]。

2.3 本章小结

本章节先从 Linux 内核的网络处理着手，分析了网络收发包的工作流程，指出内核包处理存在的问题。因为系统调用及通用性等原因导致其无法充分利用万兆级网卡的性能，包括（1）处理单个包所占的 CPU 时钟周期太长；（2）包在内存的分配和释放等操作占用太多 CPU 使用率；（3）因为容纳各种协议的结构，导致包太冗长，占用过多内存；（4）网络栈路径过长，有额外的内存拷贝；（5）系统调用导致的用户态与内核态之间的切换开销；（6）自旋锁影响多核并行处理包等。其后分析了 DPDK 的架构和原理，研究了 DPDK 采用的先进思想和技术，以更好的使用 DPDK 进行网络应用开发。

3 基于 DPDK 的三层转发设计与实现

设计实现了基于 DPDK 的一个三层转发系统 D-L3 Forwarding，绕过了 Linux 的内核协议栈，通过多核绑定进行 IPv4 包收发处理。并分析影响系统性能的关键因素。

3.1 设计目标

1. 实现基于（目的 IPv4 地址，IPv4 地址掩码，目的转发端口号）的三元组的路由转发功能。

2. 在性能上面，充分利用 10Gb/s 的带宽，执行高速的转发功能。

3. 基于通用服务器平台，使用现有的用户态网络 I/O 处理接口，保障系统的通用性。

3.2 系统架构

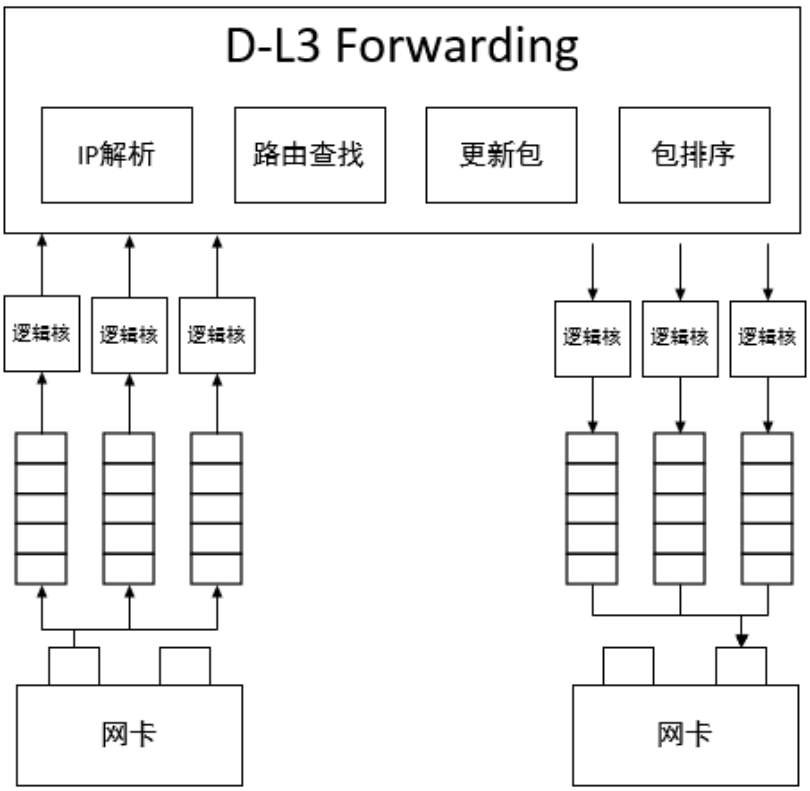

图 3.1 D-L3 Forwarding 系统架构图

D-L3 Forwarding 的系统架构图如图 3.1 所示。系统中，实现了 IPv4 解析，路由查找，更新包数据以及包排序的功能。在 D-L3 Forwarding 系统运行时，会为网卡的端口建立多个收发包队列，并绑定在不同的逻辑核，同时在每个逻辑核上执行三层转发处理功能，对该逻辑核上不同端口的接收队列的包同时进行处理，并转发到对应端口进行发送。系统对包的处理包括四个模块。

1. IP 解析模块负责对收到的包进行包头的提取，解析并得到包的目的 IPv4 地址。

2. 路由查找模块负责根据目的 IPv4 地址，在 LPM 查找表中查找对应的目的转发端口。

3. 更新包模块负责在接收包之后，更新包的源和目的 MAC 地址。

4. 包排序模块负责将所有待转发的包，根据目标端口重新排序，将拥有相同目标端口的包分成一组，加速包的转发。

3.3 系统实现

3.3.1 环境初始化

D-L3 Forwarding 在运行之前，先要经历初始化阶段，该阶段用于初始化 CPU、内存和网卡端口收发队列的配置。整个初始化阶段的流程图如图 3.2 所示。

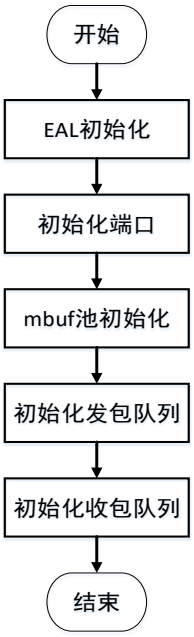

图 3.2 初始化流程图

1. EAL 初始化。整个过程通过调用 rte_eal_init 接口进行 DPDK 的运行环境初始化。EAL 初始化的流程如图 3.3 所示。

rte_eal_init 会通过如下初始化函数执行不同的初始化任务。

（1）调用 rte_eal_cpu_init 函数获取系统的 CPU 数目，读取的数据会放在 rte_config 全局结构当中。

（2）调用函数 eal_hugepage_info_init 函数获取来自/proc/mounts 下的 hugetlbfs 挂载目录，并把系统的巨页基本信息存放在 internal_config 结构中的 hugepage_info 结构成员，以用于后续内存的初始化。

（3）调用 rte_config_init 函数，配置 rte_config 结构及其中的共享内存结构指针 mem_config。此函数会在/var/run 或者用户的 home 目录创建.rte_config 文件存放 rte_config 结构，并初始化 rte_config.mem_config。

（4）调用 rte_eal_pci_init 函数，扫描系统中所有的 PCI 设备，并创建对应的 device 结构链到 pci_device_list 列表中。通过 rte_pci_device 结构记录每个位于表中的设备结点。

（5）调用 rte_eal_memory_init 函数，若是主进程，调用 rte_eal_hugepage_init 函数，在巨页文件系统 hugetlbfs 新建多个映射文件，并将巨页内存映射到这些映射文件当中，接下来更新 mem_config 数据结构及以/var/run/.rte_hugepage_info 文件，在里面存放 hugepage_file 结构体。若是副进程，调用 rte_eal_hugepage_attach，在副进程遍历 DPDK 运行时所配置的内存段，找到形成这些内存段的巨页，之后把它们映射到虚拟内存空间的一个连续的块当中。在副进程执行获取映射内存的过程当中，需要读取.rte_config 文件.rte_hugepage_info 文件，副进程对这个文件只有读权限。

（6）调用 rte_eal_memzone_init 函数来初始化内存。DPDK 采用内存段（用 rte_memseg 结构表示）以及内存区域（用 rte_memzone 表示）这两个数据结构来管理巨页物理内存，它们的结构都保存在 rte_config.mem_config 当中。可以通过 rte_mem_virt2phy 函数，从/proc/self/pagemap 当中读取信息，来得到巨页的物理地址。每个 rte_memseg 表当中的 rte_memseg 成员均表示一段连续的内存空间，rte_memzone 则是使用该内存空间的单位。结构 rte_memzone 可以在主进程被保留，这样它可以被副进程通过访问共享文件当中的 rte_mem_config 来被查找到，之后就可以得到 rte_memzone 及 rte_memseg 在巨页内存中的物理地址。

（7）最后，会调用 pthread_create 函数为其他逻辑核创建一个从线程，并建立

起主线程与从线程的通信管道，接着调用 eal_thread_init_master 函数，将主线程绑定到第一个逻辑核当中，然后 eal_thread_loop 函数会在其他核当中开始执行，负责与主核的管道通信。DPDK 内部的 lcore_config 表会为系统保存每个核所运行的线程 ID，与主线程的管道，线程需执行的函数等配置。

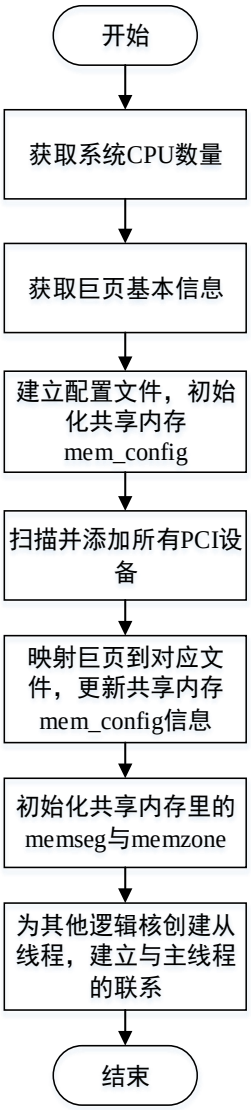

图 3.3 环境抽象层初始化流程图

2. 初始化物理端口。在执行完 eal 初始化之后，会开始初始化所有物理端口。首先，通过 rte_eth_dev_count 函数获知物理端口的个数，然后通过 check_port_config 函数检查物理端口的配置。对于每个端口，会通过 get_port_n_rx_queues 函数来获取其收包的队列数，并设置发包队列的数量为所绑定逻辑核的个数（不能超过限制的

大小），之后通过 rte_eth_dev_configure，根据端口号，收发队列的个数来配置物理端口，给物理端口创建收发队列，并更新全局静态结构 port_conf 的信息。随后会通过 rte_eth_macaddr_get 函数获取每个端口的 MAC 地址，记录到 ports_eth_addr 数组当中，最后通过 ether_addr_copy，将每个端口的源 MAC 地址和目的 MAC 地址初始化为该端口的 MAC 地址。

3. Mbuf 池初始化。一旦 eal 初始化，并且系统成功传入参数之后，mbuf 池就会被创建。Mbuf 池包含了一系列的 mbuf 对象，驱动和系统可以通过 mbuf 对象来存放网络包中的数据。三层转发系统通过 init_mem 函数来包装 mbuf 的初始化过程。在 init_mem 当中，通过 rte_pktmbuf_pool_create 函数在每个 NUMA 插槽（socket）创建一个 mbuf 池；init_mem 函数会为每个 socket 创建一个 LPM 路由表。该路由表的地址会被保存到 l_conf 结构当中，每个逻辑核都有一个 l_conf 结构，用于保存逻辑核的配置信息。

4. 初始化发包队列。系统会为每个二元组（lcore,port）初始化一个发包队列。对于每个端口，系统先会遍历启用的逻辑核，通过 rte_eth_dev_info_get 获取端口信息，保存在临时变量 dev_info 当中，并从中得到发包队列的默认配置结构 txconf。接着通过 rte_eth_tx_queue_setup 函数根据端口号和发包队列配置结构 txconf 来配置发送队列，之后在该逻辑核的 l_conf 结构上添加该发包队列的 ID，最后 ID 自增，以继续配置下一个（lcore,port）的队列。

5. 初始化收包队列。一开始，在 main 函数中，会调用 init_lcore_rx_queues 函数来根据程序传入的参数，修改每个核的 l_conf 的收包队列信息。系统先会遍历启用的逻辑核，然后读取该逻辑核的配置信息 l_conf，接着会读取出该逻辑核上的所有收包队列信息，得到要建立的收包队列的端口号和队列 ID，根据这两个参数，用 rte_eth_rx_queue_setup 在端口所处的 socket 上的 mbuf 池分配收包队列内存空间。

当上述步骤完成之后，main 函数会为每个端口调用 rte_eth_dev_start 函数使端口处于工作状态。接着通过 rte_eal_mp_remote_launch 函数，在每个线程执行 l3_loop 函数，开始包的处理。

3.3.2 IPv4 包转发处理

包处理阶段的实现流程可以用图 3.4 所表示。

首先，每个线程会通过 rte_lcore_id()获取线程所在逻辑核的 id，通过该 id，获取

该逻辑核的配置信息结构 l_conf。然后，线程会检测该逻辑核是否有接收队列绑定，如果没有接收队列，则直接结束 l3_loop。如果有接收队列，则之后会进入死循环，重复进行包处理。

该循环在一开始会调用 rte_time 库的函数来计算时间，每隔一段时间，就会检测所有端口的发包队列，将 lcore 结构的 mbuf_table 成员所存放的包对象全部发送出去，并在成功发送之后释放包对象所占内存。

接着会根据 lcore 结构内所记录的该逻辑核所绑定的队列的端口号和队列号，调用 rte_eth_rx_burst 函数从每个接收队列读取包，通过一个临时 mbuf 变量 pkts_buffer，将所有接收的包的指针缓存到 pkts_buffer 变量当中。

在读取到包数据之后，可以对包进行分析及处理。系统会把所有从接收队列读取到的包，以 4 个为一组进行分批处理。处理过程可以分为 4 步。

第一步是对包 IP 的解析，提取包的目的 IPv4 地址。包的解析过程需要用 rte_pktmbuf_mtod 函数，跳过 mbuf 的元数据，获取 mbuf 所存放的包的包头指针。然后从包头读取相应偏移下的目的 ipv4 地址，然后将 4 个 32 位的目的 IPv4 地址统一存放到一个 128 位的寄存器当中。

第二步是对包进行 LPM 表的路由查找，用以得到包的转发出口。该过程需要查找之前初始化过程中，在每个 socket 所创建的 LPM 表，根据目的 IPv4 地址查找下一跳的目的端口号，如果查找失败，则会用包一开始被接收时所在的端口号来设置下一跳的目的端口号。查找结束之后，会将包的下一跳目的端口记录在 dst_port 数组。

第三步是对包的源和目的 MAC 地址进行更新。首先需要用 rte_pktmbuf_mtod 函数，跳过 mbuf 的元数据，获取 mbuf 所存放的包的包头指针。然后根据包对应的下一跳目的端口，读取 val_eth 获取在初始化时设置的该目的端口的源和目的 MAC 地址。之后用该源和目的 MAC 地址，更新包的以太网包头中的源和目的 MAC 地址。

第四步是对读取到的包进行重排序。系统会根据 dst_port 所记录的目的端口地址，将拥有相同目的端口的包放在同一组，保证每 4 个包为一组的包集合当中有着相同的目的端口。

由于每次所读取的包不一定都能被 4 整除，因此需要考虑这些剩下的包，对于这些剩下的包，系统将对每个包进行单独处理，即单独获取其 IPv4 目的地址，对其进行 LPM 路由表查找，更新其源和目的 MAC 地址，并在最后把每个包与之前的包进行重排序操作，同样是为了保证每 4 个包为一组的包集合中有相同目的端口。

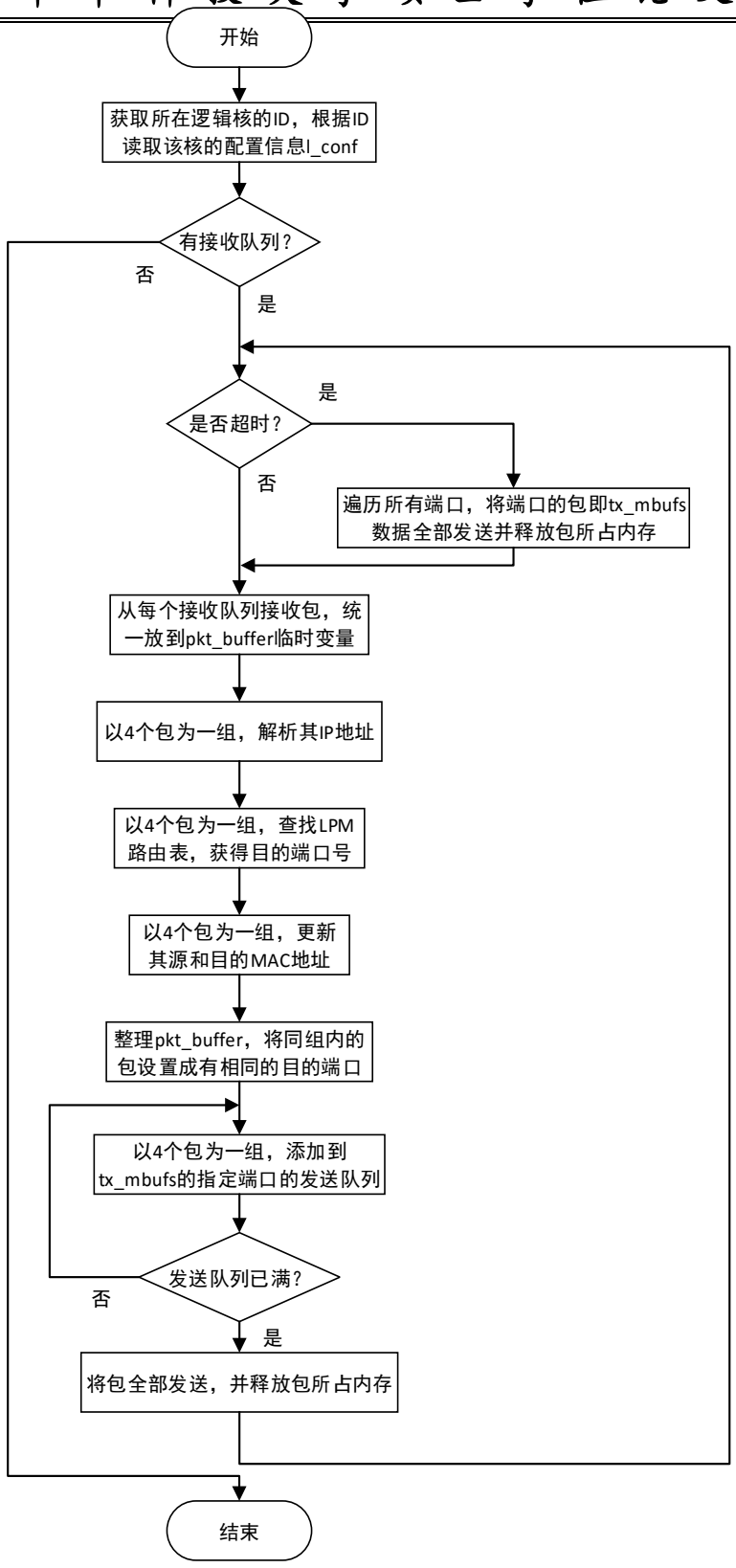

图 3.4　l3_loop 阶段的处理流程图

在执行完上述 4 个步骤之后，便可以对包执行发送操作。经过前面的分批处理过程，在 pkts_buffer 数组当中，以第一个包开始，对于其顺序的每 4 个包，都有着相同的转发目的端口。因此，可以以每 4 个 pkts_buffer 元素为一组，将其同时保存到 l_conf 维护的该目的端口的发送队列进行发送。如果该发送队列为空，并且 pkts_buffer 所要发送的包大于发送队列，那么系统会调用 rte_eth_tx_burst 直接发送这些包。如果发送队列仍有空间可以容纳这些包，那么系统会先把这些包缓存到发送队列当中，在这些包被缓存之后，如果接收队列已满，则系统会把这些包通过 rte_eth_tx_burst 发送出去，并在完成包的转发之后，释放包内存。

3.4 性能影响因素

根据前面所阐明的 DPDK 架构原理以及根据 DPDK 所实现的三层转发系统，分析可能影响 DPDK 以及所实现的系统及性能的几个关键因素。

3.4.1 非一致内存访问

从目前来看，商用服务器可以根据系统架构大体可以分为以下三类。（1）对称多处理器结构（Symmetric Multi-Processor，MPP）。在 SMP 架构中，每个 CPU 对称工作，每个 CPU 通过同一条总线共享相同的物理内存，每个 CPU 也会花费相同的时间去访问内存中的任何地址，因此 SMP 也被称为一致存储器访问结构（UMA，Uniform Memory Access）。由于 SMP 架构中的 CPU、内存、I/O 等所有资源都是共享的，因此它的扩展能力十分有限。对内存的访问则是最受限制的，由于每个 CPU 必须通过相同的内存总线访问相同的内存资源，因此如果 CPU 数量越多，内存访问冲突也将越严重，同时总线的负载也会加剧，最终导致 CPU 性能下降。（2）大规模并行处理结构（Massive Parallel Processing，MPP）。由多个 SMP 服务器通过节点互联网络连接而成，每个服务器只能访问本地资源，是一种完全无共享结构，因而扩展能力最好，理论上其扩展无限制。在 MPP 系统中，每个 SMP 服务器可以运行自己的操作系统、数据库等。节点之间的信息交互只能通过节点互联网络实现。为了实现各个节点的负载和并行处理过程，MPP 结构需要一种复杂的机制。（3）非一致内存访问结构（NUMA，Non-Uniform Memory Access）。在 NUMA 架构中有多个 CPU 模块，每个 CPU 模块由多个 CPU 组成，并且具有独立的本地内存、I/O 槽口等。由于其节点之间可以通过互联模块进行连接和信息交互，因此每个 CPU 可以访问整

个系统的内存。

NUMA 的拓扑图如图 3.5 及图 3.6 所示。NUMA 将所有处理器分成若干个节点（node），每个节点可能会拥有多个 CPU 插槽（socket），以拥有多个 CPU，而每个 CPU 又拥有多个物理核，在每个物理核上，又可以通过超线程技术将其当成两个逻辑核。每个节点内部会有自己的总线和内存，也可以通过系统总线访问其他节点的内存。

NUMA 架构的优势在于节点内部的 CPU 可以通过访问本地内存时来提高处理性能，并减小对系统总线的压力。当开发系统时，可以将系统的线程都绑定在同节点的 CPU 当中执行，这样可以避免通过节点之间的总线去访问远端内存，带来性能上的下降。在 D-L3 Forwarding 系统运行的时候，由于系统将收发包队列绑定到了固定的逻辑核当中，所以应当充分考虑 NUMA 架构带来的影响，因为系统在运行时，可能会因为逻辑核之间存在过多的远端内存的通信，使得系统的性能受到影响。

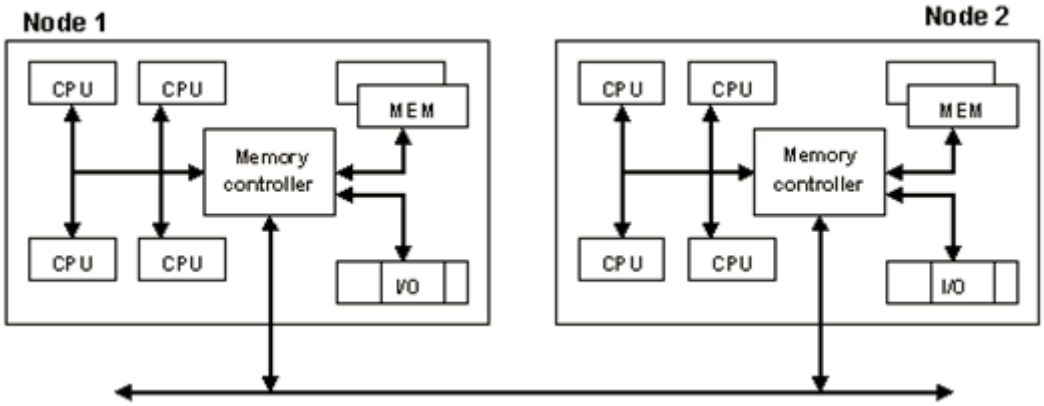

图 3.5 NUMA 总体拓扑图

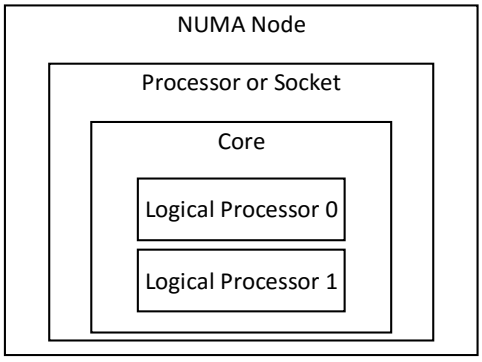

图 3.6 NUMA 单个节点结构图

3.4.2 巨页

对于内存的管理，大多数操作系统采用了分段或分页的方式进行管理。分段是粗粒度的管理方式，而分页则是细粒度管理方式，分页方式可以避免内存空间的浪费。相应地，存在内存的物理地址与虚拟地址的概念。通过前面这两种方式，CPU 必须把虚拟地址转换程物理内存地址才能真正访问内存。为了提高这个转换效率，CPU 会缓存最近的虚拟内存地址和物理内存地址的映射关系，并保存在一个由 CPU 维护的映射表中。为了尽量提高内存的访问速度，需要在映射表中保存尽量多的映射关系。

Linux 的内存管理采取的是分页存取机制，为了保证物理内存能得到充分的利用，内核会按照 LRU 算法在适当的时候将物理内存中不经常使用的内存页自动交换到虚拟内存中，而将经常使用的信息保留到物理内存。通常情况下，Linux 默认情况下每页是 4K，这就意味着如果物理内存很大，则映射表的条目将会非常多，会影响 CPU 的检索效率。因为内存大小是固定的，为了减少 TLB 映射表的条目，可采取的办法只有增加页的尺寸。巨页（Huge Page）是一个较好的解决方案。巨页打破了传统的小页面的内存管理方式，使用比如 2M 以上的大页面来代替原来的 4K 小页面。如此一来映射条目则明显减少。通过使用更少的 TLB 映射表条目可以增大 TLB 所表示的内存区域， TLB 未命中的概率也会下降，这样系统花在 TLB 映射表条目的替换开销就会下降，从而加快地址转换的速度，提高程序访问内存的性能。

D-L3 Forwarding 基于 DPDK 实现，而 DPDK 的内存管理建立在巨页的基础之上，在巨页内存分配不足的情况下，会因为巨页的未命中及替换操作带来额外的性能开销。因此，系统分配的巨页的大小很可能会给系统的运行带来一定影响。

3.4.3 内存通道

多通道内存结构是一种内存访问优化技术，它通过在 DRAM 内存和内存控制器之间添加多个通道，来提高两者之间的数据传输速率。理论上来说这会让数据传输速度随通道的增加而增加。现在的高端处理器如英特尔 I7 Extreme 处理器与 Xeon 处理器都可以支持四通道内存。

图 3.7 描述的是一个支持四通道 CPU 的例子，一个 CPU 的内存控制器可以同时获取 4 个 DIMM（Dual Inline Memory Modules,双列直插式存储模块）的内容。通过采用交叉存取技术（interleaving），可以给内存访问带来高带宽和低延迟。数据被分

成许多小数据块，并被分布到多个 DIMM 当中。当需要读取数据的时候，便通过分布到不同通道的每个 DIMM 当中读取小数据块，而不是从单个通道的单个 DIMM 中获取整个数据块。此外，同一个通道的 DIMM 还可以继续从不同层级（rank）中同时访问数据，来进一步通过交叉存取技术提高性能。

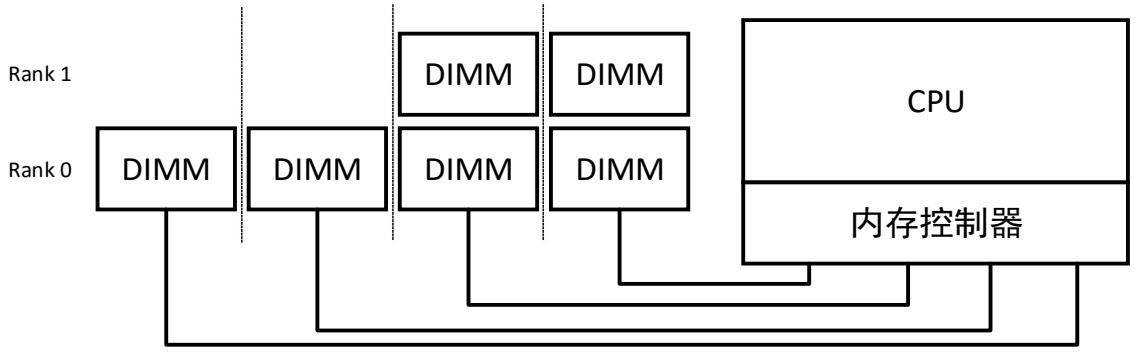

图 3.7 内存多通道技术描述图

对于 DPDK 来说，多通道技术也是可能影响到性能的一环。DPDK 可以把处理对象均衡分布到不同的通道和层级以提高性能。对于 DPDK 三层转发系统来说，由于只要读取包的前 64 个字节就可以确定包的转发方向，因此可以把包对象分配到不同的通道上，来加快包的处理。比如图 3.8 展示了一种双通道四层级的 DIMM 的包分布策略，图中第一行表示块号，第二行表示通道号，第三行表示层级，最后一行表示网络包，通过 DPDK 的对齐策略，并在包与包之间加入适当间隔（padding），使得相邻包能够被分布到不同层级的不同通道，达到加速包处理的目的。

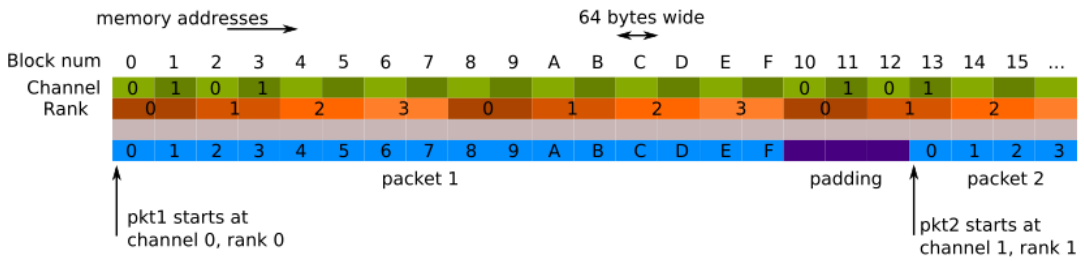

图 3.8 针对 DIMM 结构的包分布策略示例

3.5 本章小结

本章节实现了基于 DPDK 的三层转发系统 D-L3 Forwarding，阐述了系统的设计目标，解释了系统的架构及实现过程。并在其基础上分析了可能影响 D-L3 Forwarding 系统性能的关键因素，这些因素包括 NUMA 架构，巨页以及内存通道的设置。

4 测试结果及分析

围绕基于 DPDK 的三层转发系统 D-L3 Forwarding 进行测试。将 D-L3 Forwarding 与 Netmap 的收包性能进行对比测试，并完成 NUMA 架构、巨页、内存通道数对性能的的影响测试与分析。

4.1 测试环境

4.1.1 测试服务器配置

表 4.1 ca26 硬件配置

CPU	2x CPU(Intel Xeon E5-2620, 6x cores (each 2.0GHz))
内存	16GB DDR3 RAM
硬盘	8x SAS Disks (15000RPM, each 146GB)
操作系统	CentOS Linux release 7.0.1406
内核版本	Linux 3.13.0
网卡	Intel Corporation I350 Gigabit Network Connection
万兆网卡	Intel Corporation Ethernet 10G 2P X520 Adapter
DPDK 版本	2.2.0

表 4.2 de73 硬件配置

CPU	2x CPU(Intel Xeon E5620, 4x cores (each 2.4GHz))
内存	20GB DDR3 RAM
硬盘	2x SATA Disks (15000RPM,each 300GB)
操作系统	CentOS Linux release 7.0.1406
内核版本	Linux 3.13.0
网卡	Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network
DPDK 版本	2.2.0

测试所采用的服务器配置如表 4.1，表 4.2 所示。

测试系统采用了两台服务器。在两台服务器上均配置了 10Gbit 以太网卡，并编

译安装了版本号为 2.2.0 的 DPDK 套件。所有实验均使用 Linux 3.13.0 内核。将表 4.1 所示的浪潮服务器命名为 ca26，表 4.2 所示的戴尔服务器命名为 de73。

为了研究 NUMA 架构会如何对系统产生影响，需要给出系统的 NUMA 信息。可以通过 lscpu 或者第三方工具如 likwid-topology 获得该信息。两台机器的 NUMA 架构如图 4.1 和图 4.2 所示。比如图 4.2 当中，有两个 NUMA 节点，一共包含 16 个逻辑核。在同一 NUMA 节点中，同一列的两个逻辑核位于同一物理核，每个物理核有独自的一级缓存和二级缓存。每个 NUMA 节点中的逻辑核共享三级缓存、本地内存以及网卡。

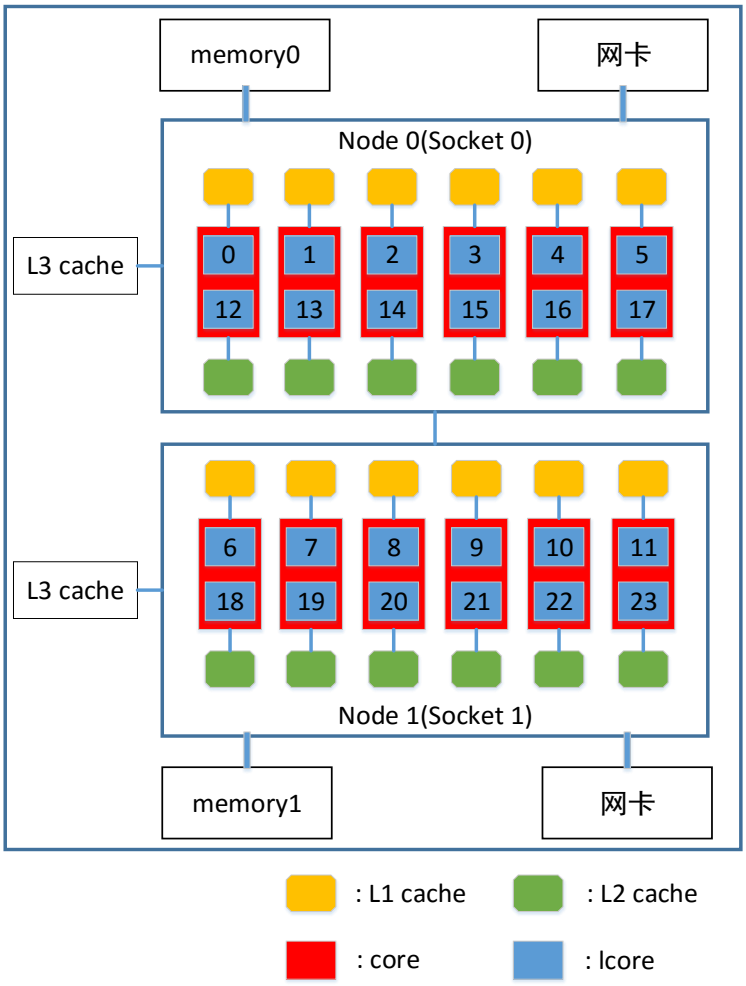

图 4.1 ca26 NUMA 架构图

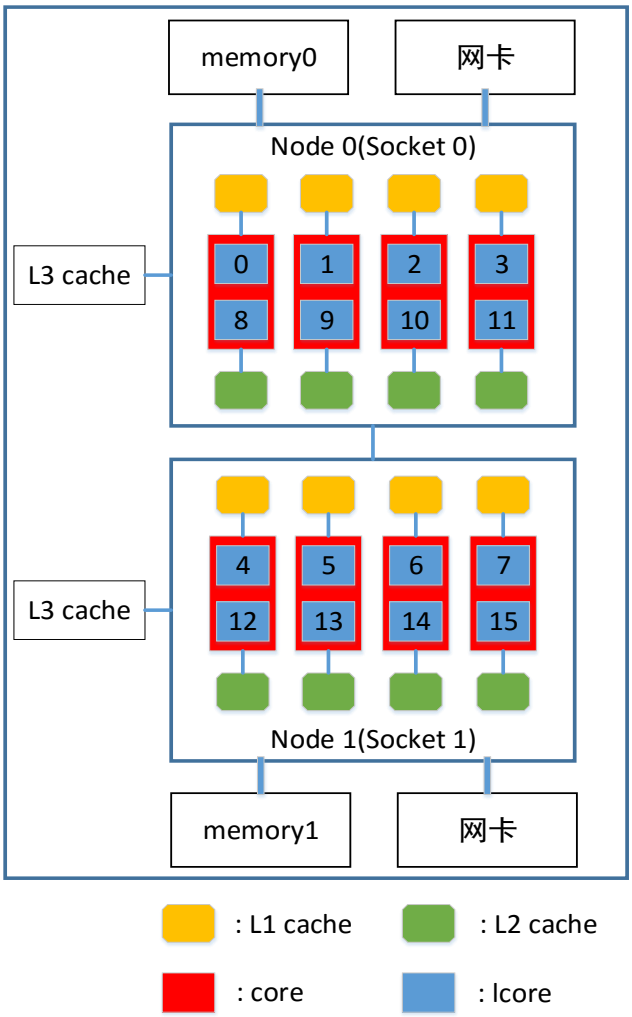

图 4.2 de73 架构图

4.1.2 测试拓扑

测试环境拓扑图如图 4.3 所示。测试在 ca26 服务器上面运行 pktgen-dpdk 这个包生成工具，在 de73 服务器面上运行三层转发系统 D-L3 Forwarding。在两台机器中，均把万兆网卡的两个端口绑定到 igb_uio 的用户态轮询驱动模块，而不是加载内核的 ixgbe 驱动。这样一来，pktgen-dpdk 以及三层转发系统便可以将这两个端口绑定到所选逻辑核当中，以执行收发和转发操作。根据拓扑图所描述的系统执行流程为：（1）根据双方端口的 MAC 地址，修改三层转发系统中每个端口的目的 MAC 地址并配置路由表；（2）编译并运行三层转发系统；（3）编译并运行 pktgen-dpdk 工具；（4）

根据双方端口的 MAC 地址及三层转发系统的路由表，配置 pktgen-dpdk 的运行时环境；（5）在 ca26 服务器中，通过 pktgen-dpdk 产生包，并通过万兆网卡的 0 号端口进行发送到 de73 服务器的 0 号端口；（6）de73 中的三层转发系统通过万兆网卡的 0 号端口接收到包之后，会进行包的处理，并把包通过万兆网卡的 1 号端口转发到 ca26 服务器的 1 号端口；（7）pktgen-dpdk 通过万兆网卡的 1 号端口接收到来自 ca26 服务器发送的包，并实时给出本地两个收发端口的统计信息。

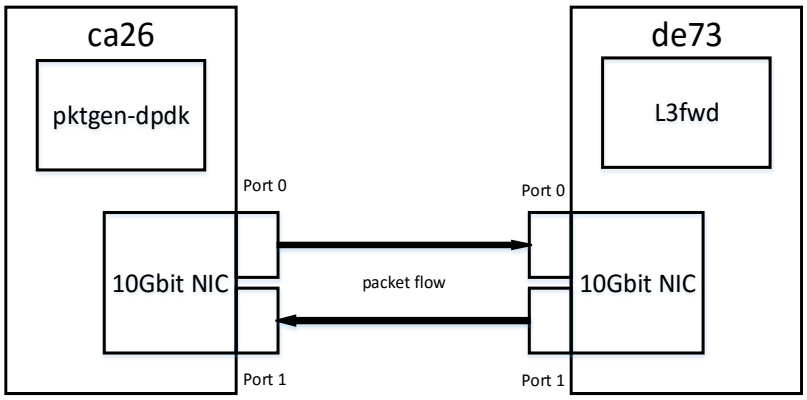

图 4.3 测试拓扑图

4.1.3 测试工具

Pktgen-dpdk 是一个基于 DPDK 进行开发的一款包生成工具，由于采用了 DPDK 的库，它相比于使用 Linux 自带的发包工具 Pktgen[44]具有更高的性能，特别是在发送小包的时候，可以充分发挥出万兆以太网卡的性能。Pktgen-dpdk 具有如下特点。

1. 它能产生 14.88Mpps 速率(10Gb/s 网卡的极限吞吐率)的 64 字节大小的包。
2. 它能够作为包接收器或包发送器，并且以线速处理这些包。
3. 它能在运行时灵活配置环境信息，并随时可以开始或者停止包的传输。
4. 它能够实时显示端口的信息，比如端口配置或者通讯状况。
5. 它可以根据源和目的 MAC 地址或者 IP 地址，或者端口号来产生包。
6. 它能生成和处理 UDP，TCP，ARP，ICMP，GRE，MPLS 等类型的包。
7. 可以通过 TCP 连接来远程控制 pktgen-dpdk 的运行。
8. 它可以通过 Lua 脚本或者通过运行命令行脚本来设置测试环境。
9. 它满足 BSD 协议，用户可以获得其源代码在其基础上进行二次开发。

4.2 测试结果

4.2.1 D-L3forwarding 与 Netmap 对比测试

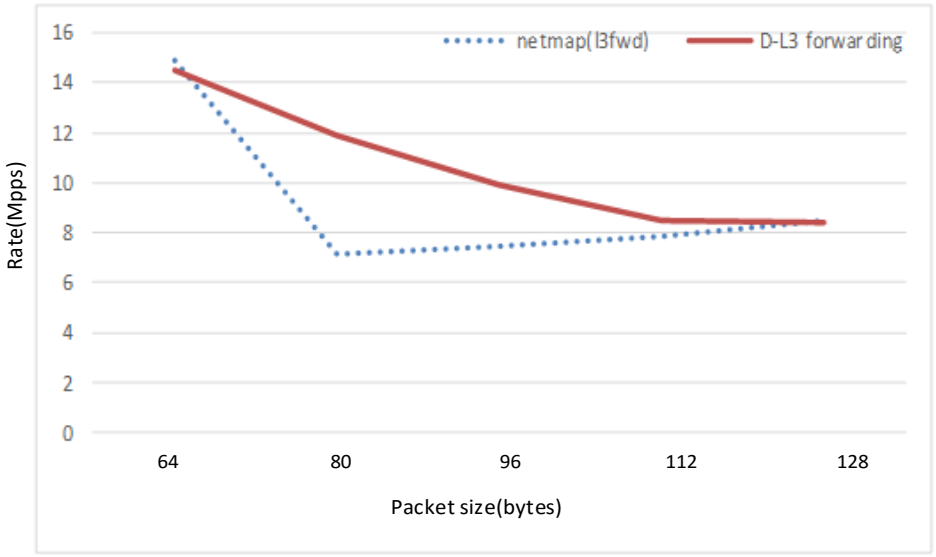

图 4.4 D-L3 Forwarding 与 Netmap 收包性能对比

D-L3 Forwarding 与 Netmap 的收包性能对比如图 4.4 所示。可以看出 Netmap 在包大小为 64 字节之后，包的传输速率会突然骤降，并在包大小到达 128 字节前逐步提高，在包大小为 128 字节情况下又重新回到网卡所能达到的最高收包速率。这是由于 Netmap 仍然使用传统 poll()/select()系统调用与内核驱动交互，进行包的接收，存在一定的系统调用开销。基于 DPDK 的 D-L3 Forwarding 系统直接在用户态以轮询的方式进行网络包的接收，有效的避免了系统调用开销。虽然 D-L3 Forwarding 在 64 字节情况下的收包速率相比 Netmap 略有不足，但是在测试范围内的性能表现很稳定，不会产生骤降的情况。综合比较，D-L3 Forwarding 相比 Netmap 在小包情况下能提供更稳定的性能表现。

4.2.2 NUMA 架构的影响测试

三层转发系统可以通过传入三元组参数（端口号，队列号，逻辑核号），表示将某端口的某一收包队列绑定在某个逻辑核当中进行处理。测试收集了不同绑定情况下的包性能（转发率）。收包端口与发包端口所绑定逻辑核在不同情况下的测试数据如表 4.3 到表 4.6 所示。统计信息均由 pktgen-dpdk 的实时数据得到。测试通过

另一个三元组（端口 0 所绑定逻辑核，端口 1 所绑定逻辑核，三层转发系统所在逻辑核）来表示每组测试数据，并以端口 0 所绑定逻辑核进行归类。为了测试极限转发率，测试中 pktgen-dpdk 生成的包大小均为 64 字节。

表 4.3 同一逻辑核的测试结果

测试组号	收包速率（pps 或 Mbit/s）	发包速率（pps 或 Mbit/s）
(1,1,2)	14467816/9259	14880209/9523
(1,1,4)	13543649/8667	14880100/9523
(1,1,9)	14593996/9340	14880105/9523

表 4.4 同一 NUMA 节点的测试结果

测试组号	收包速率（pps 或 Mbit/s）	发包速率（pps 或 Mbit/s）
(1,2,1)	14515368/9289	14880098/9523
(1,2,2)	7103306/4546	14880314/9523
(1,2,3)	14522364/9294	14880094/9523
(1,2,4)	13699600/8767	14880116/9523
(1,2,9)	14488304/9272	14880109/9523
(1,2,10)	10513348/6728	14880090/9523
(1,2,11)	14518355/9291	14880099/9523

表 4.5 不同 NUMA 节点的测试结果

测试组号	收包速率（pps 或 Mbit/s）	发包速率（pps 或 Mbit/s）
(1,5,1)	14600020/9344	14880214/9523
(1,5,2)	14597908/9342	148801179523
(1,5,5)	6846880/4382	14880092/9523
(1,5,6)	13988368/8952	14880099/9523
(1,5,9)	14598748/9343	14880103/9523
(1,5,13)	9149968/5855	14880106/9523
(1,5,14)	13964783/8937	14880078/9523

表 4.6 收包端口与发包端口位于同一物理核的测试结果

测试组号	收包速率 （pps 或 Mbit/s）	发包速率 （pps 或 Mbit/s）
(1,9,1)	10687519/6840	14880096/9523
(1,9,2)	14520192/9292	14880095/9523
(1,9,4)	13291388/8506	14888028/9523
(1,9,9)	7276288/4656	14880053/9523

由上述表格中所展示的测试结果，可以得到以下结论。

1. 三层转发系统绑定的逻辑核与收包端口所绑定逻辑核的关系

（1）通过诸如表 4.4 的（1,2,1）与（1,2,3）或者表 4.5 的（1,5,1）与（1,5,2）等对比测试三层转发系统绑定的逻辑核与收包端口所绑定逻辑核在同一逻辑核时，转发性能不受影响；

（2）通过诸如表 4.4（1,2,9）与（1,2,1）或者表 4.5（1,5,9）与（1,5,1）等对比测试，可以得出：三层转发系统绑定的逻辑核与收包端口所绑定逻辑核在同一 NUMA 节点同一个物理核的不同逻辑核时，转发性能不受影响；

（3）通过诸如表 4.3 的（1,1,2）与（1,1,4）或者表 4.6 的（1,9,2）与（1,9,4）等对比测试，可以看出三层转发系统绑定的逻辑核与收包端口所绑定逻辑核分处不同 NUMA 节点时，转发性能会受到一定影响。这是因为三层转发系统需要对所收到的包进行处理（如地址信息提取，分析查表等），然而当收包的队列与三层转发系统处于不同的 NUMA 节点时，对远端内存访问性能不如本地内存，因此带来了转发性能的下降。

2. 收包端口与发包端口所绑定逻辑核的关系

（1）对比表 4.3 的（1,1,2）与表 4.4 的（1,2,3）等其他类似组合可以得出，收包端口所绑定逻辑核与发包端口所绑定逻辑核在同一逻辑核时，转发性能不受影响；

（2）对比表 4.6 的（1,9,2）与表 4.4 的（1,2,3）等其他类似组合可以得出，收包端口所绑定逻辑核与发包端口所绑定逻辑核在同一 NUMA 节点同一个物理核的不同逻辑核时，转发性能不受影响；

（3）对比表 4.5 的（1,5,2）与表 4.4 的（1,2,3）等其他类似组合可以得出，收包端口所绑定逻辑核与发包端口所绑定逻辑核分处不同的 NUMA 节点时，转发性能

同样不受影响；

3. 三层转发系统绑定的逻辑核与发包端口所绑定逻辑核的关系

（1）对比表 4.5 的（1,5,13）与（1,5,2）或者表 4.4 的（1,2,10），（1,2,3）等测试可以得出，三层转发系统绑定的逻辑核与发包端口所绑定逻辑核在同一物理核的不同逻辑核时，会严重降低转发速度；并且当发包所绑定逻辑核与三层转发系统所绑定逻辑核是同一逻辑核的时候，比如表 4.5 的组合（1,5,5）或者表 4.4 的组合（1,2,2），性能达到最差。这是因为两个逻辑核在物理上是一个物理核心的两个线程，一个物理核心采用分片技术（超线程技术）来达到双线程的并发，并不是传统意义上的多核并行操作。因此当一个物理核心同时用于执行三层转发的处理以及包的发送时，会对 L1，L2 缓存进行抢占，使得缓存行不断被更新替换，性能便会受到影响；而当在同一逻辑核执行上述两个任务时，则两个任务会抢占该逻辑核，并且浪费了该物理核心的一个另一个逻辑核，因此性能会达到最差。

（2）对比表 4.5 的（1,5,2）与表 4.4 的（1,2,3）等实验对比可以得出，系统绑定的逻辑核与收包端口所绑定逻辑核分处不同 NUMA 节点时，并不会影响系统性能。

4.2.3 巨页的影响测试

巨页大小对三层转发系统 D-L3 Forwarding 的影响的测试结果如表 4.7 所示。由测试结果可以得出，增加巨页大小，对系统转发性能影响很小，可以忽略。但是需要注意的一点是，巨页大小必须大于（128,128）MB，否则系统将没有足够的内存空间处理接收到的包。

表 4.7 巨页大小的测试结果

巨页大小	收包速率 （pps 或 Mbit/s）	发包速率 （pps 或 Mbit/s）
256,256	14547496/9310	14880085/9523
512,512	14493152/9275	14880095/9523
1024,1024	14512952/9288	14880086/9523
2048,2048	14517532/9291	14880245/9523

4.2.4 内存通道数的影响测试

内存通道数对三层转发系统 D-L3 Forwarding 的测试结果如表 4.8 所示。可以看

到，增加系统运行时所采用的内存通道数量并不能有效影响到 D-L3 Forwarding 的转发性能。因为 DPDK 使用 mbuf 结构（详见本文第 3 章）表示发送/接收到的包，每个包的元数据所占内存较小（不超过一个 cache line，即 64 字节），同时由于本文测试均使用 64 字节小包（若使用大包测试，带宽很容易达到上限，测试没有意义），因此内存通道数量对 D-L3 Forwarding 系统转发性能影响较小，可以忽略。

表 4.8 内存通道数的测试结果

内存通道数	收包速率 （pps 或 Mb/s）	发包速率 （pps 或 Mb/s）
1	14519544/9292	14880091/9523
2	14506004/9283	14866384/9514
3	14525592/9296	14880141/9523
4	14515784/9290	14880109/9523

4.3 本章小结

本章设计了 4 类对比实验，分别测试了 D-L3 Forwarding 的性能以及 NUMA 架构、巨页及内存通道对 D-L3 Forwarding 系统性能的影响。介绍了相关测试环境及工具，如测试拓扑图及发包工具。最后对实验数据进行对比分析。测试结果表明，D-L3 Forwarding 系统能够在高速网络（10Gb/s）下，对包进行高效的转发。D-L3 Forwarding 系统对 NUMA 架构的使用会对性能产生较大影响，而巨页大小和内存通道数对 D-L3 Forwarding 系统性能的影响可以忽略。

5 全文总结

在大数据和分布式存储以及网络定义存储，网络功能虚拟化蓬勃发展的今天，对网络的传输要求越来越大。随着 CPU 架构的不断更新换代，多核 CPU 以及基于多核 CPU 的 NUMA 架构成为了主流，为服务器的计算性能带来了进一步的提高。在网络适配器方面，也从原来的百兆带宽提高到了万兆级别。Linux 内核的网络栈对于一般网络应用而言可以满足其性能需求，但是当采用的是 NUMA 架构的 CPU 及万兆网卡时，便不能充分发挥 NUMA 的优势和万兆网卡的性能，于是英特尔开发了数据平面开发套件 DPDK，通过使用该开发套件，可以绕过传统的 Linux 内核网络栈，帮助用户开发高性能的网络应用。

论文主要围绕 DPDK 进行展开，实现了基于 DPDK 的三层转发系统 D-L3 Forwarding，对影响系统性能的几个关键因素进行了研究。论文所做主要工作如下。

1. 详细阐述 Linux 内核网络栈，分析其存在的问题：包的数据结构过于庞大，包处理程序太复杂，系统调用导致的用户态与内核态切换存在开销，以及在收发队列上面存在锁开销。之后对比 Linux 网络栈，分析了 DPDK 的架构，论述了 DPDK 的优点：直接在用户态通过轮询取代中断对网卡进行收发包，能够将单个线程绑定到 CPU 核，提供了高效的内存管理及无锁环形队列。

2. 实现基于 DPDK 的三层转发系统 D-L3 Forwarding，使用高性能的 DPDK 接口绕过 Linux 网络栈，为网卡端口建立多个收发包队列，将这些队列绑定在固定的 CPU 核进行并行处理，采用 SSE 指令集对包进行分组及批量转发。

3. 搭建系统的测试环境，测试三层转发系统 D-L3 Forwarding 的性能。研究分析了可能影响系统性能的几个关键因素。测试结果表明，系统在小包时的转发速度可以接近 10Gbit/s 网卡的带宽。NUMA 架构的配置会对系统的性能产生较大的影响，然而系统在 10Gbit/s 网卡环境下，内存通道数与巨页大小的改动并不会影响小包的转发性能。

论文仍在存在一些问题。在实验测试方面，需要进一步完善，比如进一步优化系统，使得 64 字节包处理速度进一步提升，以达到 10Gbit 网卡的带宽；通过采用 40Gbit 的网卡进行测试，以进一步研究影响系统性能的因素。

致谢

在论文完成之际，我首先要感谢我的父母和家人，是我的家人和父母一直以来对我的支持和信任，给了我生活上的动力。

感谢我的导师，刘景宁老师，她总是待我如朋友，对我关照有加，在我的科研道路上给了我许多指导和建议。也要感谢童薇老师，在我迷茫的时候能够给我帮助。

感谢实验室的冯丹老师、王芳老师、李洁琼老师、谭志鹏老师、华宇老师、曾令仿老师、熊双莲老师、施展老师等各位老师，正是因为有各位老师对实验室的含辛茹苦的付出，共同引领实验室不断探索新的方向，实验室才会有现在蒸蒸日上的科研氛围。

感谢国光实验室的同学们的帮助，尤其是吴运翔师兄，他在我的研究生过程中同样一直指导着我，在许多方面上给我了很好的点子。感谢 F306 房间的同学们，在这里，可以很认真的科研，也可以很自在的开玩笑。

最后感谢我的同学，感谢我的舍友，这两年来一起度过的日子很高兴，我很庆幸能够拥有这两年，很庆幸能够认识你们，与你们一起学习，科研，打球，唱歌，一起共度欢乐与忧愁的日子是值得被铭记一生的。

感谢华中科技大学，六年以来在华中科技大学里面学到了很多东西，我很骄傲自己是华科的一份子。感谢研究生期间，国家光电实验室给我的科研生活提供了一系列优越条件。当我走出校园，迈向社会之时，我会很自豪地跟别人说，我是一名华科人，是一名国光人。

参考文献

[1] Intel 82599 10 GbE Controller Datasheet. Intel, .http://download.intel.com/design/network/datashts/82599 datasheet.pdf. 2012

[2] Lameter C. Numa (non-uniform memory access): An overview. Queue, 2013, 11(7): 40

[3] Nieplocha J, Harrison R J, Littlefield R J. Global arrays: A nonuniform memory access programming model for high-performance computers. The Journal of Supercomputing, 1996, 10(2): 169-189

[4] Foglia P, Prete C A, Solinas M, et al. Investigating design tradeoffs in S-NUCA based CMP systems. UCAS, 2009: 53-60

[5] Liu F, Jiang X, Solihin Y. Understanding How Off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. High Performance Computer Architecture, 2009: 1-12

[6] Boyd-Wickizer S, Clements A T, Mao Y, et al. An Analysis of Linux Scalability to Many Cores. OSDI. 2010, 10(13): 86-93

[7] Benvenuti C. Understanding Linux network internals. O'Reilly Media Inc, 2006

[8] Intel. DPDK: Data Plane Development Kit Project Page. http://www.dpdk.org. 2016

[9] Koch H J, Linutronix Gmb H. Userspace I/O drivers in a realtime context. The 13th Realtime Linux Workshop. 2011

[10] Yang J, Minturn D B, Hady F. When poll is better than interrupt. FAST. 2012, 12: 3-3

[11] Fusco F, Deri L. High speed network traffic analysis with commodity multi-core systems. 10th ACM SIGCOMM Conference on Internet Measurement. ACM, 2010:218-224

[12] Rusu S, Tam S, Muljono H, et al. A 45nm 8-core enterprise Xeon® processor. Solid-State Circuits Conference, 2009: 9-12

[13] Nunes B A A, Mendonca M, Nguyen X N, et al. A survey of software-defined networking: Past, present, and future of programmable networks. Communications Surveys & Tutorials, IEEE, 2014, 16(3): 1617-1634

[14] Han B, Gopalakrishnan V, Ji L, et al. Network function virtualization: Challenges and opportunities for innovations. Communications Magazine, IEEE, 2015, 53(2): 90-97.

[15] Sinharoy B, Van Norstrand J A, Eickemeyer R J, et al. IBM POWER8 processor core

microarchitecture. IBM Journal of Research and Development, 2015, 59(1): 1-2

[16] Mattina M. Architecture and Performance of the Tile-GX Processor Family. white paper, 2014: 1-18

[17] Seal D. ARM architecture reference manual. Pearson Education, 2001

[18] Pfaff B, Pettit J, Koponen T, et al. The design and implementation of open vswitch. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). 2015: 117-130

[19] McKeown N, Anderson T, Balakrishnan H, et al. OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 2008, 38(2): 69-74

[20] Pongrácz G, Molnar L, Kis Z L. Removing roadblocks from SDN: OpenFlow software switch performance on Intel DPDK.Software Defined Networks (EWSDN), 2013 Second European Workshop on. IEEE, 2013: 62-67

[21] 陈明达. 固态硬盘(SSD)产品现状与展望. 移动通信, 2009, 33(11):29-31

[22] Simpson R E, Fons P, Kolobov A V, et al. Interfacial phase-change memory. Nature Nanotechnology, 2011, 6(8):501-505

[23] Hosomi M, Yamagishi H, Yamamoto T, et al. A Novel Nonvolatile Memory with Spin Torque Transfer Magnetization Switching: Spin-RAM. International Electron Devices Meeting, 2005: 459-462

[24] Strukov D B, Snider G S, Stewart D R, et al. The missing memristor found. nature, 2008, 453(7191): 80-83

[25] Caulfield A M, Mollov T I, Eisner L A, et al. Providing safe, user space access to fast, solid state disks. ACM SIGARCH Computer Architecture News, 2012, 40(1): 387-400

[26] Bjørling M, Axboe J, Nellans D, et al. Linux block IO: introducing multi-queue SSD access on multi-core systems. SYSTOR. 2013: 1-10

[27] Thai Le. Introduction to the Storage Performance Development Kit (SPDK).Intel Corp. 2015

[28] Intel Corporation. Intel QuickData Technology Software Guide for Linux. Intel Corp, 2008

[29] Weil S A, Brandt S A, Miller E L, et al. Ceph: A scalable, high-performance distributed file system. Operating systems design and implementation. USENIX Association, 2006: 307-320

[30] Weil S A, Brandt S A, Miller E L, et al. CRUSH: Controlled, scalable, decentralized placement of replicated data. 2006 ACM/IEEE conference on Supercomputing. ACM, 2006: 122

[31] Weil S A, Leung A W, Brandt S A, et al. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. International workshop on Petascale data storage: held in conjunction with Supercomputing'07. ACM, 2007: 35-44

[32] Sébastien Han. Ceph Jewel Preview: a new store is coming, BlueStore. https://www.sebastien-han.fr/blog/2016/03/21/ceph-a-new-store-is-coming/. 2016

[33] Rizzo L. Netmap: a novel framework for fast packet I/O. 21st USENIX Security Symposium (USENIX Security 12). 2012: 101-112

[34] Deri L, PF RING M. URL http://www. ntop. org. PF_RING. html, 2011

[35] Nvidia C. Compute unified device architecture programming guide. 2007

[36] Han S, Jang K, Park K S, et al. PacketShader: a GPU-accelerated software router. ACM SIGCOMM Computer Communication Review, 2011, 41(4): 195-206

[37] Dominik Scholz. A look at intel's dataplane development kit. Network , 2014:115

[38] Rosen R. Linux Kernel Networking: Implementation and Theory. Apress, 2013

[39] Rio M, Goutelle M, Kelly T, et al. A map of the networking code in Linux kernel 2.4. 20. DataTAG, 2004

[40] Wu W, Crawford M, Bowden M. The performance analysis of Linux networking–packet receiving. Computer Communications, 2007, 30(5): 1044-1057

[41] García-Dorado J L, Mata F, Ramos J, et al. High-performance network traffic processing systems using commodity hardware. Data Traffic Monitoring and Analysis, 2013: 3-27

[42] Intel. Intel DPDK: Getting Started Guide, Intel Corp. 2014

[43] Intel. Intel DPDK: Programmers Guide, Intel Corp. 2015

[44] Olsson R. Pktgen the linux packet generator. Linux Symposium, Ottawa, Canada, 2005: 11-24