深入浅出 DPDK

朱河清　梁存铭　胡雪焜　曹水　等编著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

深入浅出 DPDK / 朱河清等编著 . —北京：机械工业出版社，2016.5

ISBN 978-7-111-53783-0

I. 深… II. 朱… III. 应用软件 – 软件包 IV. TP317

中国版本图书馆 CIP 数据核字（2016）第 097359 号

近年来，随着半导体和多核计算机体系结构技术的不断创新和市场的发展，越来越多的网络设备基础架构开始向基于通用处理器平台的架构方向融合，期望用更低的成本和更短的产品开发周期来提供多样的网络单元和丰富的功能，如应用处理、控制处理、包处理、信号处理等。为了适应这一新的产业趋势，英特尔公司十年磨一剑，联合第三方软件开发公司及时推出了基于 Intel® x86 的架构 DPDK（Data Plane Development Kit，数据平面开发套件），实现了高效灵活的包处理解决方案。经过近 3 年的开源与飞速发展，DPDK 已经发展成业界公认的高性能网卡和多通用处理器平台的开源软件工具包，并已成为通用处理器平台上影响力最大的数据平面解决方案。主流的 Linux 发行版都已经将 DPDK 纳入，DPDK 引发了基于 Linux 的高速网络技术的创新热潮，除了在传统的通信网络、安全设施领域应用之外，还被广泛应用于云计算、虚拟交换、存储网络甚至数据库、金融交易系统。

本书汇聚了最资深的 DPDK 技术专家的精辟见解和实战体验，详细介绍了 DPDK 技术的发展趋势、数据包处理、硬件加速技术、虚拟化以及 DPDK 技术在 SDN、NFV、网络存储等领域的实际应用。书中还使用大量的篇幅讲解各种核心软件算法、数据优化思想，并包括大量详尽的实战心得和使用指南。

作为国内第一本全面阐述网络数据面的核心技术的书籍，本书主要面向 IT、网络通信行业的从业人员，以及大专院校的师生，用通俗易懂的文字打开了一扇通向新一代网络处理架构的大门。DPDK 完全依赖软件，对 Linux 的报文处理能力做了重大革新，它的发展历程是一个不可多得的理论联系实际的教科书般的实例。

深入浅出 DPDK

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：姚 蕾　　责任校对：殷 虹

印　　刷：北京诚信伟业印刷有限公司　　版　　次：2016 年 5 月第 1 版第 1 次印刷

开　　本：186mm×240mm 1/16　　印　　张：18

书　　号：ISBN 978-7-111-53783-0　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

Prologue 序　言

2015 年的春天，在北京参加 DPDK 研讨大会时，有幸结识了本书的部分作者和众多 DPDK 研发的专业人士。这使我对这个专题的感召力深感诧异。DPDK 就像一块磁铁，可以把这么多不同行业的专业人士吸引在一起。同时，大家会上也相约来年的春天，国内的同仁们能在 DPDK 技术进步中展现出自己独到的贡献。

作为运营商研发队伍的一员，我们无时不刻都能感受到 NFV 这个话题的灼热度。作为网络演进的大趋势，NFV 将在未来为运营商实现网络重构扮演重要的角色。然而，大家都知道，NFV 技术的发展之路存在各种屏障，性能问题是一道迈不过去的坎。这个问题的复杂性在于，它涉及 I/O、操作系统内核、协议栈和虚拟化等多个层面对网络报文的优化处理技术。虽然 IT 界已发展出多类小众技术来应对，但这些技术对于普通应用技术人员而言比较陌生，即使对于传统网络的开发者而言，全面掌握这些技术也存在巨大的挑战。长久以来，用户更希望在这个领域有系统性的解决方案，能把相关的技术融会贯通，并系统性地组织在一起，同时也需要更为深入的细节技术支持工作。

DPDK 的到来正逢其时，它之所以能脱颖而出，并迅速发展为业界在 NFV 加速领域的一种标杆技术，在于它不仅是上述技术的集大成者，更重要的是它的开放性和持续迭代能力，这些都得益于 DPDK 背后这支强大的专业研发团队，而本书的专业功力也可见一斑。

作为运营商的网络研发队伍，我们已关注 DPDK 近 3 年，尽管学习过 DPDK 部分源码和大量社区文档，也组织通过大量的 DPDK 相关 NFV 测试验证，但我们仍然觉得迫切需要系统性地介绍现代服务器体系架构，以及虚拟化环境下 I/O 优化的最新技术。令人倍感欣慰的是，本书作者对 DPDK 的讲解游刃有余，系统全面的同时又不乏敏锐的产业视角。可以说，深入浅出是本书最大的特点。

形而上者谓之道，形而下者谓之器。书中一方面透彻地讲解了现代处理器体系架构、网络 I/O、内核优化和 I/O 虚拟化的原理与技术发展史，在这个“道用”的基础上，另一方面也

清晰地介绍了 DPDK 细节性的“器用”知识，包括并行处理、队列调度、I/O 调优、VNF 加速等大量方法与应用，两方面相得益彰。结合 DPDK 社区的开源代码和动手实践，相信读者仔细学习完本书，必能加快对 NFV 性能关键技术的领悟。本书的受益对象首先是那些立志跨界转型的 NFV 研发工程师，也面向高等院校计算机专业希望在体系架构方面有更深发展的在校生，更包括像我们这样关注 DPDK 应用场景、NFVI 集成和测试技术的最终用户。我们衷心感谢作者为业界带来的全新技术指引。这本书就像一粒种子，其中蕴含的知识未来定会在 NFV 这片沃土上枝繁叶茂，开花结果。

严格地讲，我们的团队只是 DPDK 用户的用户，我们研究 DPDK 的目的并非针对 DPDK 本身，而是为 NFV 的集成和开发提供一个准确的、可供评估的 NFVI 性能基准，减少各类网络功能组件在私有的优化过程中存在的不稳定风险。从对 DPDK 的初步评测来看，结果令人满意甚至超出预期，但我们仍应清醒地认识到，DPDK 作为 NFV 加速技术架构仍有很长的路要走，打造成熟、规范和完善的产业链是近期要解决的重要课题。我们呼吁也乐见有更多的朋友加入 DPDK 应用推广的行列，众志成城，汇聚成一股 SDN 时代的创新洪流。

欧亮博士　中国电信广州研究院

Introduction 引　言

动机

2015 年 4 月，第一届 DPDK 中国峰会在北京成功召开。来自中国移动、中国电信、阿里巴巴、IBM、Intel、华为以及中兴的专家朋友登台演讲，一起分享了以 DPDK 为中心的技术主题。表 1 列出了 2015 DPDK 中国峰会的主题及演讲者。

表 1　2015 DPDK 中国峰会主题及演讲者

主　　题	演讲者	公　　司
利用 DPDK 加速 NFV	邓辉	中国移动
利用 DPDK 优化云基础设施	孙成浩	阿里巴巴
构建 core 以及高能效应用的最佳实践	梁存铭	Intel
基于英特尔 ONP 构建虚拟化的 IP 接入方案	欧亮	中国电信
DPDK 加速无线数据核心网络	陈东华	中兴
电信业务场景下的数据面挑战	刘郡	华为
运行于 Power 架构下的 DPDK 和数据转发	祝超	IBM

这次会议吸引了来自各行业、科研单位与高校的 200 多名开发人员、专家和企业代表参会。会上问答交流非常热烈，会后我们就想，也许是时间写一本介绍 DPDK、探讨 NFV 数据面的技术书籍。现在，很多公司在招聘网络和系统软件人才时，甚至会将 DPDK 作为一项技能罗列在招聘要求中。DPDK 从一个最初的小众技术，经过 10 年的孕育，慢慢走来，直至今日已经逐渐被越来越多的通信、云基础架构厂商接受。同时，互联网上也出现不少介绍 DPDK 基础理论的文章和博客，从不同的角度对 DPDK 技术进行剖析和应用，其中很多观点非常新颖。作为 DPDK 的中国开发团队人员，我们意识到如果能够提供一本 DPDK 的书籍，进行一些系统性的梳理，将核心的原理进行深入分析，可以更好地加速 DPDK 技术的普及，触发更多的软件创新，促进行业的新技术发展。于是，就萌发了写这本书的初衷。当然，我

们心里既有创作的激动骄傲，也有些犹豫忐忑，写书不是一件简单的事情，但经过讨论和考量，我们逐渐变得坚定，这是一本集结团队智慧的尝试。我们希望能够把 DPDK 的技术深入浅出地解释清楚，让更多的从业人员和高校师生了解并使用 DPDK，促进 DPDK 发展日新月异，兴起百家争鸣的局面，这是我们最大的愿景。

多核

2005 年的夏天，刚加入 Intel 的我们畅想着 CPU 多核时代的到来给软件业带来的挑战与机会。如果要充分利用多核处理器，需要软件针对并行化做大量改进，传统软件的并行化程度不高，在多核以前，软件依靠 CPU 频率提升自动获得更高性能。并行化改进不是一件简单的工作，许多软件需要重新设计，基本很难在短期实现，整个计算机行业都对此纠结了很久。2005 年以前，整个 CPU 的发展历史，是不断提升芯片运算频率核心的做法，软件性能会随着处理器的频率升高，即使软件不做改动，性能也会跟着上一个台阶。但这样的逻辑进入多核时代已无法实现。首先我们来看看表 2 所示的 Intel® 多核处理器演进。

表 2　Intel® 多核处理器演进的历史图表

Xeon 处理器代码	制造工艺	最大核心数量	发布时间	超线程	双路服务器可使用核心数量
WoodCrest	65nm	2	2006 年 6 月	否	4
Nehalem-EP	45nm	4	2009 年 7 月	是	16
Westmere-EP	32nm	6	2010 年 2 月	是	24
SandyBridge-EP	32nm	8	2012 年 3 月	是	32
IvyBridge-EP	22nm	12	2013 年 9 月	是	48
Haswell-EP	22nm	18	2014 年 9 月	是	72

在过去 10 年里，服务器平台的处理器核心数目扩展了很多。表 2 参考了英特尔至强系列的处理器的核心技术演进历史，这个系列的处理器主要面向双通道（双路）服务器和相应的硬件平台。与此同时，基于 MIPS、Power、ARM 架构的处理器也经历着类似或者更加激进的并行化计算的路线图。在处理器飞速发展的同时，服务器平台在硬件技术上提供了支撑。基于 PCI Express 的高速 IO 设备、内存访问与带宽的上升相辅相成。此外，价格和经济性优势越发突出，今天一台双路服务器的价格可能和 10 年前一台高端笔记本电脑的价格类似，但计算能力达到甚至超越了当年的超级计算机。强大的硬件平台为软件优化技术创新蕴蓄了温床。

以太网接口技术也经历了飞速发展。从早期主流的 10Mbit/s 与 100Mbit/s，发展到千兆网（1Gbit/s）。到如今，万兆（10Gbit/s）网卡技术成为数据中心服务器的主流接口技术，近年来，Intel 等公司还推出了 40Gbit/s、100Gbit/s 的超高速网络接口技术。而 CPU 的运行频率基本停

留在 10 年前的水平，为了迎接超高速网络技术的挑战，软件也需要大幅度创新。

结合硬件技术的发展，DPDK（Data Plane Development Kit），一个以软件优化为主的数据面技术应时而生，它为今天 NFV 技术的发展提供了绝佳的平台可行性。

IXP

提到硬件平台和数据面技术，网络处理器是无法绕过的话题。电信行业通常使用网络处理器或类似芯片技术作为数据面开发平台首选。Intel 此前也曾专注此领域，2002 年收购了 DEC 下属的研究部门，在美国马萨诸塞州哈德逊开发了这一系列芯片，诞生了行业闻名的 Intel Exchange Architecture Network Processor（IXP4xx、IXP12xx、IXP24xx、IXP28xx）产品线，曾取得行业市场占有率第一的成绩。即使今日，相信很多通信业的朋友，还对这些处理器芯片有些熟悉或者非常了解。IXP 内部拥有大量的微引擎（MicroEngine），同时结合了 XSCALE 作为控制面处理器，众所周知，XSCALE 是以 ARM 芯片为核心技术的一种扩展。

2006 年，AMD 向 Intel 发起了一场大战，时至今日结局已然明了，Intel 依赖麾下的以色列团队，打出了新一代 Core 架构，迅速在能效比上完成超车。公司高层同时确立了 Tick-Tock 的研发节奏，每隔两年推出新一代体系结构，每隔两年推出基于新一代制造工艺的芯片。这一战略基本保证了每年都会推出新产品。当时 AMD 的处理器技术一度具有领先地位，并触发了 Intel 在内部研发架构城门失火的状况下不得不进行重组，就在那时 Intel 的网络处理器业务被进行重估，由于 IXP 芯片系列的市场容量不够大，Intel 的架构师也开始预测，通用处理器多核路线有取代 IXP 专用处理芯片的潜力。自此，IXP 的研发体系开始调整，逐步转向使用 Intel CPU 多核的硬件平台，客观上讲，这一转型为 DPDK 的产生创造了机会。时至今日，Intel 还保留并发展了基于硬件加速的 QuickAssist 技术，这和当日的 IXP 息息相关。由此看来，DPDK 算是生于乱世。

DPDK 的历史

网络处理器能够迅速将数据报文接收入系统，比如将 64 字节的报文以 10Gbit/s 的线速也就是 14.88Mp/s（百万报文每秒）收入系统，并且交由 CPU 处理，这在早期 Linux 和服务器平台上无法实现。以 Venky Venkastraen、Walter Gilmore、Mike Lynch 为核心的 Intel 团队开始了可行性研究，并希望借助软件技术来实现，很快他们取得了一定的技术突破，设计了运行在 Linux 用户态的网卡程序架构。传统上，网卡驱动程序运行在 Linux 的内核态，以中断方式来唤醒系统处理，这和历史形成有关。早期 CPU 运行速度远高于外设访问，所以中断处理方式十分有效，但随着芯片技术与高速网络接口技术的一日千里式发展，报文吞吐需要高达

10Gbit/s 的端口处理能力，市面上已经出现大量的 25Gbit/s、40Gbit/s 甚至 100Gbit/s 高速端口，主流处理器的主频仍停留在 3GHz 以下。高端游戏玩家可以将 CPU 超频到 5GHz，但网络和通信节点的设计基于能效比经济性的考量，网络设备需要日以继夜地运行，运行成本（包含耗电量）在总成本中需要重点考量，系统选型时大多选取 2.5GHz 以下的芯片，保证合适的性价比。I/O 超越 CPU 的运行速率，是横在行业面前的技术挑战。用轮询来处理高速端口开始成为必然，这构成了 DPDK 运行的基础。

在理论框架和核心技术取得一定突破后，Intel 与 6wind 进行了合作，交由在法国的软件公司进行部分软件开发和测试，6wind 向 Intel 交付了早期的 DPDK 软件开发包。2011 年开始，6wind、Windriver、Tieto、Radisys 先后宣布了对 Intel DPDK 的商业服务支持。Intel 起初只是将 DPDK 以源代码方式分享给少量客户，作为评估 IA 平台和硬件性能的软件服务模块，随着时间推移与行业的大幅度接受，2013 年 Intel 将 DPDK 这一软件以 BSD 开源方式分享在 Intel 的网站上，供开发者免费下载。2013 年 4 月，6wind 联合其他开发者成立 www.dpdk.org 的开源社区，DPDK 开始走上开源的大道。

开源

DPDK 在代码开源后，任何开发者都被允许通过 www.dpdk.org 提交代码。随着开发者社区进一步扩大，Intel 持续加大了在开源社区的投入，同时在 NFV 浪潮下，越来越多的公司和个人开发者加入这一社区，比如 Brocade、Cisco、RedHat、VMware、IBM，他们不再只是 DPDK 的消费者，角色向生产者转变，开始提供代码，对 DPDK 的代码进行优化和整理。起初 DPDK 完全专注于 Intel 的服务器平台技术，专注于利用处理器与芯片组高级特性，支持 Intel 的网卡产品线系列。

DPDK 2.1 版本在 2015 年 8 月发布，几乎所有行业主流的网卡设备商都已经加入 DPDK 社区，提供源代码级别支持。另外，除了支持通用网卡之外，能否将 DPDK 应用在特别的加速芯片上是一个有趣的话题，有很多工作在进行中，Intel 最新提交了用于 Crypto 设备的接口设计，可以利用类似 Intel 的 QuickAssit 的硬件加速单元，实现一个针对数据包加解密与压缩处理的软件接口。

在多架构支持方面，DPDK 社区也取得了很大的进展，IBM 中国研究院的祝超博士启动了将 DPDK 移植到 Power 体系架构的工作，Freescale 的中国开发者也参与修改，Tilera 与 Ezchip 的工程师也花了不少精力将 DPDK 运行在 Tile 架构下。很快，DPDK 从单一的基于 Intel 平台的软件，逐步演变成一个相对完整的生态系统，覆盖了多个处理器、以太网和硬件加速技术。

在 Linux 社区融合方面，DPDK 也开始和一些主流的 Linux 社区合作，并得到了越来越多的响应。作为 Linux 社区最主要的贡献者之一的 RedHat 尝试在 Fedora Linux 集成 DPDK；接着 RedHat Enterprise Linux 在安装库里也加入 DPDK 支持，用户可以自动下载安装 DPDK 扩展库。RedHat 工程师还尝试将 DPDK 与 Container 集成测试，并公开发布了运行结果。传统虚拟化的领导者 VMware 的工程师也加入 DPDK 社区，负责 VMXNET3-PMD 模块的维护。Canonical 在 Ubuntu 15 中加入了 DPDK 的支持。

延伸

由于 DPDK 主体运行在用户态，这种设计理念给 Linux 或者 FreeBSD 这类操作系统带来很多创新思路，也在 Linux 社区引发一些讨论。

DPDK 的出现使人们开始思考，Linux 的用户态和内核态，谁更适合进行高速网络数据报文处理。从简单数据对比来看，在 Intel 的通用服务器上，使用单核处理小包收发，纯粹的报文收发，理想模型下能达到大约 57Mp/s（每秒百万包）。尽管在真实应用中，不会只收发报文不处理，但这样的性能相对 Linux 的普通网卡驱动来说已经是遥不可及的高性能。OpenVSwitch 是一个很好的例子，作为主流的虚拟交换开源软件，也尝试用 DPDK 来构建和加速虚拟交换技术，DPDK 的支持在 OVS2.4 中被发布，开辟了在内核态数据通道之外一条新的用户态数据通道。目前，经过 20 多年的发展，Linux 已经累积大量的开源软件，具备丰富的协议和应用支持，无所不能，而数据报文进出 Linux 系统，基本都是在 Linux 内核态来完成处理。因为 Linux 系统丰富强大的功能，相当多的生产系统（现有软件）运行在 Linux 内核态，这样的好处是大量软件可以重用，研发成本低。但也正因为内核功能强大丰富，其处理效率和性能就必然要做出一些牺牲。

使用

在专业的通信网络系统中，高速数据进出速率是衡量系统性能的关键指标之一。大多通信系统是基于 Linux 的定制系统，在保证实时性的嵌入式开发环境中开发出用户态下的程序完成系统功能。利用 DPDK 的高速报文吞吐优势，对接运行在 Linux 用户态的程序，对成本降低和硬件通用化有很大的好处，使得以软件为主体的网络设备成为可能。对 Intel® x86 通用处理器而言，这是一个巨大的市场机会。

对于通信设备厂商，通用平台和软件驱动的开发方式具有易采购、易升级、稳定性、节约成本的优点。

- **易采购**：通用服务器作为主流的基础硬件，拥有丰富的采购渠道和供应商，供货量巨大。

- **易升级**：软件开发模式简单，工具丰富，最大程度上避免系统升级中对硬件的依赖和更新，实现低成本的及时升级。
- **稳定性**：通用服务器平台已经通过大量功能的验证，产品稳定性毋庸置疑。而且，对于专用的设计平台，系统稳定需要时间累积和大量测试，尤其是当采用新一代平台设计时可能需要硬件更新，这就会带来稳定性的风险。
- **节约研发成本和降低复杂性**：传统的网络设备因为功能复杂和高可靠性需求，系统切分为多个子系统，每个子系统需要单独设计和选型，独立开发，甚至选用单独的芯片。这样的系统需要构建复杂的开发团队、完善的系统规划、有效的项目管理和组织协调，来确保系统开发进度。而且，由于开发的范围大，各项目之间会产生路径依赖。而基于通用服务器搭建的网络设备可以很好地避免这些问题。

版权

DPDK 全称是 Data Plane Development Kit，从字面解释上看，这是专注于数据面软件开发的套件。本质上，它由一些底层的软件库组成。目前，DPDK 使用 BSD license，绝大多数软件代码都运行在用户态。少量代码运行在内核态，涉及 UIO、VFIO 以及 XenDom0，KNI 这类内核模块只能以 GPL 发布。BSD 给了 DPDK 的开发者和消费者很大的自由，大家可以自由地修改源代码，并且广泛应用于商业场景。这和 GPL 对商业应用的限制有很大区别。作为开发者，向 DPDK 社区提交贡献代码时，需要特别注意 license 的定义，开发者需要明确 license 并且申明来源的合法性。

社区

参与 DPDK 社区［ Ref1-1 ］（www.dpdk.org）就需要理解它的运行机制。目前，DPDK 的发布节奏大体上每年发布 3 次软件版本（announce@dpdk.org），发布计划与具体时间会提前公布在社区里。DPDK 的开发特性也会在路标中公布，一般是通过电子邮件列表讨论 dev@dpdk.org，任何参与者都可以自由提交新特性或者错误修正，具体规则可以参见 www.dpdk.org/dev，本书不做详细解读。对于使用 DPDK 的技术问题，可以参与 user@dpdk.org 进入讨论。

子模块的维护者名单也会发布到开源社区，便于查阅。在提交代码时，源代码以 patch 方式发送给 dev@dpdk.org。通常情况下，代码维护者会对提交的代码进行仔细检查，包括代码规范、兼容性等，并提供反馈。这个过程全部通过电子邮件组的方式来完成，由于邮件量可能巨大，如果作者没有得到及时回复，请尝试主动联系，提醒代码维护人员关注，这是参

与社区非常有效的方式。开发者也可以在第一次提交代码时明确抄送相关的活跃成员和专家，以得到更加及时的反馈和关注。

目前，开源社区的大量工作由很多自愿开发者共同完成，因此需要耐心等待其他开发者来及时参与问答。通过提前研究社区运行方式，可以事半功倍。这对在校学生来说更是一个很好的锻炼机会。及早参与开源社区的软件开发，会在未来选择工作时使你具有更敏锐的产业视角和技术深度。作为长期浸润在通信行业的专业人士，我们强烈推荐那些对软件有强烈兴趣的同学积极参与开源社区的软件开发。

贡献

本书由目前DPDK社区中一些比较资深的开发者共同编写，很多作者是第一次将DPDK的核心思想付诸于书面文字。尽管大家已尽最大的努力，但由于水平和能力所限，难免存在一些瑕疵和不足，也由于时间和版面的要求，很多好的想法无法在本书中详细描述。但我们希望通过本书能够帮助读者理解DPDK的核心思想，引领大家进入一个丰富多彩的开源软件的世界。在本书编纂过程中我们得到很多朋友的帮助，必须感谢上海交通大学的金耀辉老师、中国科学技术大学的华蓓和张凯老师、清华大学的陈渝教授、中国电信的欧亮博士，他们给了我们很多中肯的建议；另外还要感谢李训和刘勇，他们提供了大量的资料和素材，帮助我们验证了大量的DPDK实例；还要感谢我们的同事杨涛、喻德、陈志辉、谢华伟、戴启华、常存银、刘长鹏、Jim St Leger、MJay、Patrick Lu，他们热心地帮助勘定稿件；最后还要特别感谢英特尔公司网络产品事业部的领导周晓梅和周林给整个写作团队提供的极大支持。正是这些热心朋友、同事和领导的支持，坚定了我们的信心，并帮助我们顺利完成此书。最后我们衷心希望本书读者能够有所收获。

作者介绍（按姓名排序）*About the Authors*

曹水：黑龙江省佳木斯人，2001 年毕业于复旦大学计算机系，硕士。现为英特尔软件经理，从事嵌入式开发和软件行业超过 15 年，现主要负责 DPDK 软件测试工作。

陈静：湖北省沙市人，2006 年毕业于华中科技大学，硕士。现为英特尔软件开发工程师，主要从事 DPDK 网卡驱动的开发和性能调优工作。

何少鹏：江西省萍乡人，毕业于上海交通大学，硕士。现为英特尔 DPDK 软件工程师，开发网络设备相关软件超过十年，也有数年从事互联网应用和 SDN 硬件设计工作。

胡雪焜：江西省南昌人，毕业于中国科学技术大学计算机系，硕士。现为英特尔网络通信平台部门应用工程师，主要研究底层虚拟化技术和基于 IA 架构的数据面性能优化，以及对网络演进的影响，具有丰富的 SDN/NFV 商业实践经验。

梁存铭：英特尔资深软件工程师，在计算机网络领域具有丰富的实践开发经验，提交过多项美国专利。作为 DPDK 早期贡献者之一，在 PCIe 高性能加速、I/O 虚拟化、IA 指令优化、改善闲时效率、协议栈优化等方面有较深入的研究。

刘继江：黑龙江省七台河人，毕业于青岛海洋大学自动化系，现主要从事 DPDK 网卡驱动程序和虚拟化研发，和 overlay 网络的性能优化工作。

陆文卓：安徽省淮南人，2004 年毕业于南京大学计算机系，硕士。现为英特尔中国研发中心软件工程师。在无线通信、有线网络方面均有超过十年的从业经验。

欧阳长春：2006 年毕业于华中科技大学计算机系，硕士。目前在阿里云任开发专家，从事网络虚拟化开发及优化，在数据报文加速、深度报文检测、网络虚拟化方面具有丰富开发经验。

仇大玉：江苏省南京人，2012 年毕业于东南大学，硕士。现为英特尔亚太研发有限公司软件工程师，主要从事 DPDK 软件开发和测试工作。

陶喆：上海交通大学学士，上海大学硕士。先后在思科和英特尔从事网络相关的设备、

协议栈和虚拟化的开发工作。曾获 CCIE R&S 认证。

万群：江西省南昌人，毕业于西安交通大学计算机系，硕士。现为英特尔上海研发中心研发工程师。从事测试领域的研究及实践近十年，对测试方法及项目管理有相当丰富的经验。

王志宏：四川省绵阳人，2011 年毕业于华东师范大学，硕士。现为英特尔亚太研发中心高级软件工程师，主要工作方向为 DPDK 虚拟化中的性能分析与优化。

吴菁菁：江苏省扬州人，2007 年毕业于西安交通大学电信系，硕士。现为英特尔软件工程师，主要从事 DPDK 软件开发工作。

许茜：浙江省杭州市人，毕业于浙江大学信电系，硕士，现为英特尔网络处理事业部软件测试人员，主要负责 DPDK 相关的虚拟化测试和性能测试。

杨子夜：2009 年毕业于复旦大学软件学院，硕士。现为英特尔高级软件工程师，从事存储软件开发和优化工作，在虚拟化、存储、云安全等领域拥有 5 个相关专利以及 20 项申请。

张合林：湖南省湘潭人，2004 年毕业于东华大学，工学硕士。现主要从事 DPDK 网卡驱动程序研发及性能优化工作。

张帆：湖南省长沙人，爱尔兰利莫里克大学计算机网络信息学博士。现为英特尔公司爱尔兰分部网络软件工程师，湖南省湘潭大学兼职教授。近年专著有《 Comparative Performance and Energy Consumption Analysis of Different AES Implementations on a Wireless Sensor Network Node 》等。发表 SCI/EI 检索国际期刊及会议论文 3 篇。目前主要从事英特尔 DPDK 在 SDN 应用方面的扩展研究工作。

朱河清：江苏省靖江人，毕业于电子科技大学数据通信与计算机网络专业，硕士，现为英特尔 DPDK 与 Hyperscan 软件经理，在英特尔、阿尔卡特、华为、朗讯有 15 年通信网络设备研发与开源软件开发经验。

Venky Venkatesan：毕业于印度孟买大学，现为英特尔网络产品集团高级主任工程师（Sr PE），DPDK 初始架构师，在美国 Oregon 负责报文处理与加速的系统架构与软件创新工作。

目　录 *Contents*

第三部分 DPDK 应用篇

第一部分 Part 1

DPDK 基础篇

软件正在统治整个世界。

——马克·安德森

本书的开始部分会重点介绍 DPDK 诞生的背景、基本概念、核心算法，并结合实例讲解各种基于 IA 平台的数据面优化技术，包括相关的网卡加速技术。希望可以帮助初次接触 DPDK 的读者全面了解 DPDK，为后面的阅读打下基础。

DPDK 基础篇共包括 9 章，其中前 5 章主要从软件优化的角度阐述如何利用 DPDK 来提升性能，包括 cache 优化、并行计算、同步互斥、转发算法等。后面 4 章则针对 PCIe 设备和高速网卡详细介绍如何优化网卡性能，提高网络带宽吞吐率。

第 1 章介绍了 DPDK 的技术演进历程，面临及需要解决的问题，以及如何从系统的角度看待 DPDK 的技术，最后结合几个编程实例帮助读者了解 DPDK 基本的编程模式。

第 2 章则系统地介绍内存和 cache 的相关基本知识，分析了各种 IA 平台上的 cache 技术的特点和优势，并介绍了一个 DPDK 的重要技术“大页”的使用。

第 3 章和第 4 章则围绕多核的使用，着重介绍如何使用多线程，最大限度地进行指令和数据的并行执行。为了解决多线程访问竞争的问题，还引入了几种常见的 DPDK 锁机制。

第 5 章详细讲述了 DPDK 的数据报文转发模型，帮助读者了解 DPDK 的工作模式。

从第 6 章开始，本书内容逐步从 CPU 转移到网卡 I/O。其中，第 6 章将会从 CPU 与 PCIe 总线架构的角度，带领读者领略 CPU 与网卡 DMA 协同工作的整个交互过程。

第 7 章则专注于网卡的性能优化，详细介绍了 DPDK 如何在软件设计、硬件平台选择和配置上实现高效的网络报文处理。

第 8 章介绍了目前高速网卡中一个非常通用的技术“多队列与流分类”，解释了 DPDK 如果利用这个技术实现更高效的 IO 处理。

第 9 章介绍了目前以网卡为主的硬件卸载与智能化发展趋势，帮助读者了解如何将 DPDK 与网卡的硬件卸载技术结合，减少 CPU 的开销，实现高协同化的软硬件设计。

第 1 章 Chapter 1

认识 DPDK

什么是 DPDK？对于用户来说，它可能是一个性能出色的包数据处理加速软件库；对于开发者来说，它可能是一个实践包处理新想法的创新工场；对于性能调优者来说，它可能又是一个绝佳的成果分享平台。当下火热的网络功能虚拟化，则将 DPDK 放在一个重要的基石位置。虽然很难用短短几语就勾勒出 DPDK 的完整轮廓，但随着认识的深入，我们相信你一定能够认可它传播的那些最佳实践方法，从而将这些理念带到更广泛的多核数据包处理的生产实践中去。

DPDK 最初的动机很简单，就是证明 IA 多核处理器能够支撑高性能数据包处理。随着早期目标的达成和更多通用处理器体系的加入，DPDK 逐渐成为通用多核处理器高性能数据包处理的业界标杆。

1.1 主流包处理硬件平台

DPDK 用软件的方式在通用多核处理器上演绎着数据包处理的新篇章，而对于数据包处理，多核处理器显然不是唯一的平台。支撑包处理的主流硬件平台大致可分为三个方向。

- ❑ 硬件加速器
- ❑ 网络处理器
- ❑ 多核处理器

根据处理内容、复杂度、成本、量产规模等因素的不同，这些平台在各自特定的领域都有一定的优势。硬件加速器对于本身规模化的固化功能具有高性能低成本的特点，网络处理器提供了包处理逻辑软件可编程的能力，在获得灵活性的同时兼顾了高性能的硬件包处理，多核处理器在更为复杂多变的高层包处理上拥有优势，随着包处理的开源生态系统逐渐丰富，以及

近年来性能的不断提升，其为软件定义的包处理提供了快速迭代的平台。参见［Ref1-2］。

随着现代处理器的创新与发展（如异构化），开始集成新的加速处理与高速 IO 单元，它们互相之间不断地融合。在一些多核处理器中，已能看到硬件加速单元的身影。从软件包处理的角度，可以卸载部分功能到那些硬件加速单元进一步提升性能瓶颈；从硬件包处理的流水线来看，多核上运行的软件完成了难以固化的上层多变逻辑的任务；二者相得益彰。

1.1.1 硬件加速器

硬件加速器被广泛应用于包处理领域，ASIC 和 FPGA 是其中最广为采用的器件。

ASIC（Application-Specific Integrated Circuit）是一种应特定用户要求和特定电子系统的需要而设计、制造的集成电路。ASIC 的优点是面向特定用户的需求，在批量生产时与通用集成电路相比体积更小、功耗更低、可靠性提高、性能提高、保密性增强、成本降低等。但 ASIC 的缺点也很明显，它的灵活性和扩展性不够、开发费用高、开发周期长。

为了弥补本身的一些缺点，ASIC 越来越多地按照加速引擎的思路来构建，结合通用处理器的特点，融合成片上系统（SoC）提供异构处理能力，使得 ASIC 带上了智能（Smart）的标签。

FPGA（Field-Programmable Gate Array）即现场可编程门阵列。它作为 ASIC 领域中的一种半定制电路而出现，与 ASIC 的区别是用户不需要介入芯片的布局布线和工艺问题，而且可以随时改变其逻辑功能，使用灵活。FPGA 以并行运算为主，其开发相对于传统 PC、单片机的开发有很大不同，以硬件描述语言（Verilog 或 VHDL）来实现。相比于 PC 或单片机（无论是冯・诺依曼结构还是哈佛结构）的顺序操作有很大区别。

全可编程 FPGA 概念的提出，使 FPGA 朝着进一步软化的方向持续发展，其并行化整数运算的能力将进一步在通用计算定制化领域得到挖掘，近年来在数据中心中取得了很大进展，比如应用于机器学习场合。我们预计 FPGA 在包处理的应用场景将会从通信领域（CT）越来越多地走向数据中心和云计算领域。

1.1.2 网络处理器

网络处理器（Network Processer Unit，NPU）是专门为处理数据包而设计的可编程通用处理器，采用多内核并行处理结构，其常被应用于通信领域的各种任务，比如包处理、协议分析、路由查找、声音 / 数据的汇聚、防火墙、QoS 等。其通用性表现在执行逻辑由运行时加载的软件决定，用户使用专用指令集即微码（microcode）进行开发。其硬件体系结构大多采用高速的接口技术和总线规范，具有较高的 I/O 能力，使得包处理能力得到很大提升。除了这些特点外，NPU 一般还包含多种不同性能的存储结构，对数据进行分类存储以适应不同的应用目的。NPU 中也越来越多地集成进了一些专用硬件协处理器，可进一步提高片内系统性能。

图 1-1 是 NP-5 处理器架构框图，以 EZCHIP 公司的 NP-5 处理器架构为例，TOP 部分为可编程部分，根据需要通过编写微码快速实现业务相关的包处理逻辑。NPU 拥有高性能和高可编程性等诸多优点，但其成本和特定领域的特性限制了它的市场规模（一般应用于专用通

信设备）。而不同厂商不同架构的 NPU 遵循的微码规范不尽相同，开发人员的成长以及生态系统的构建都比较困难。虽然一些 NPU 的微码也开始支持由高级语言（例如 C）编译生成，但由于结构化语言本身原语并未面向包处理，使得转换后的效率并不理想。

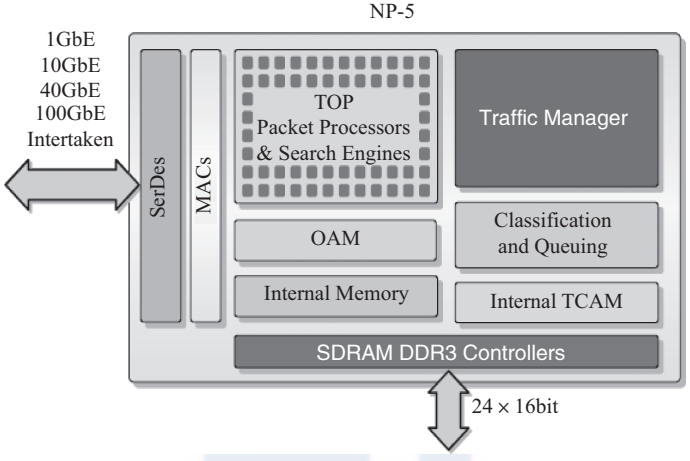


图 1-1　NP-5 处理器架构框图

随着 SDN 对于可编程网络，特别是可编程数据面的要求，网络处理器也可能会迎来新的发展机遇，但依然需要解决好不同架构的底层抽象以及上层业务的语义抽象。

1.1.3　多核处理器

现代 CPU 性能的扩展主要通过多核的方式进行演进。这样利用通用处理器同样可以在一定程度上并行地处理网络负载。由于多核处理器在逻辑负载复杂的协议及应用层面上的处理优势，以及越来越强劲的数据面的支持能力，它在多种业务领域得到广泛的采用。再加上多年来围绕 CPU 已经建立起的大量成熟软件生态，多核处理器发展的活力和热度也是其他形态很难比拟的。图 1-2 是 Intel 双路服务器平台框图，描述了一个典型的双路服务器平台的多个模块，CPU、芯片组 C612、内存和以太网控制器 XL710 构成了主要的数据处理通道。基于 PCIe 总线的 I/O 接口提供了大量的系统接口，为服务器平台引入了差异化的设计。

当前的多核处理器也正在走向 SoC 化，针对网络的 SoC 往往集成内存控制器、网络控制器，甚至是一些硬件加速处理引擎。

这里列出了一些主流厂商的多核处理器的 SoC 平台。

- ❑ IA multi-core Xeon
- ❑ Tilear-TILE-Gx
- ❑ Cavium Network-OCTEON & OCTEON II
- ❑ Freescale-QorIQ
- ❑ NetLogic Microsystem-XLP

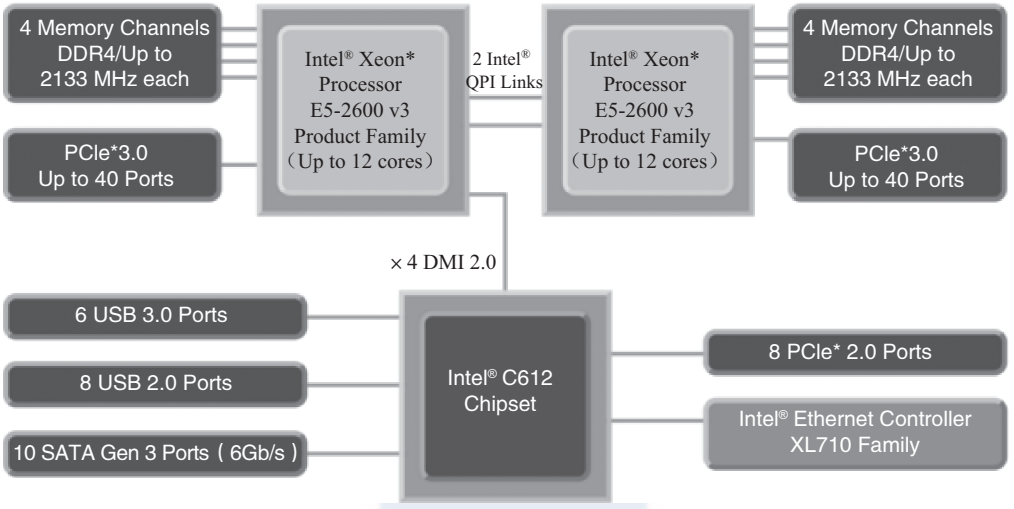


图 1-2 Intel 双路服务器平台框图

图 1-3 的 Cavium OCTEON 处理器框图以 Cavium OCTEON 多核处理器为例，它集成多个

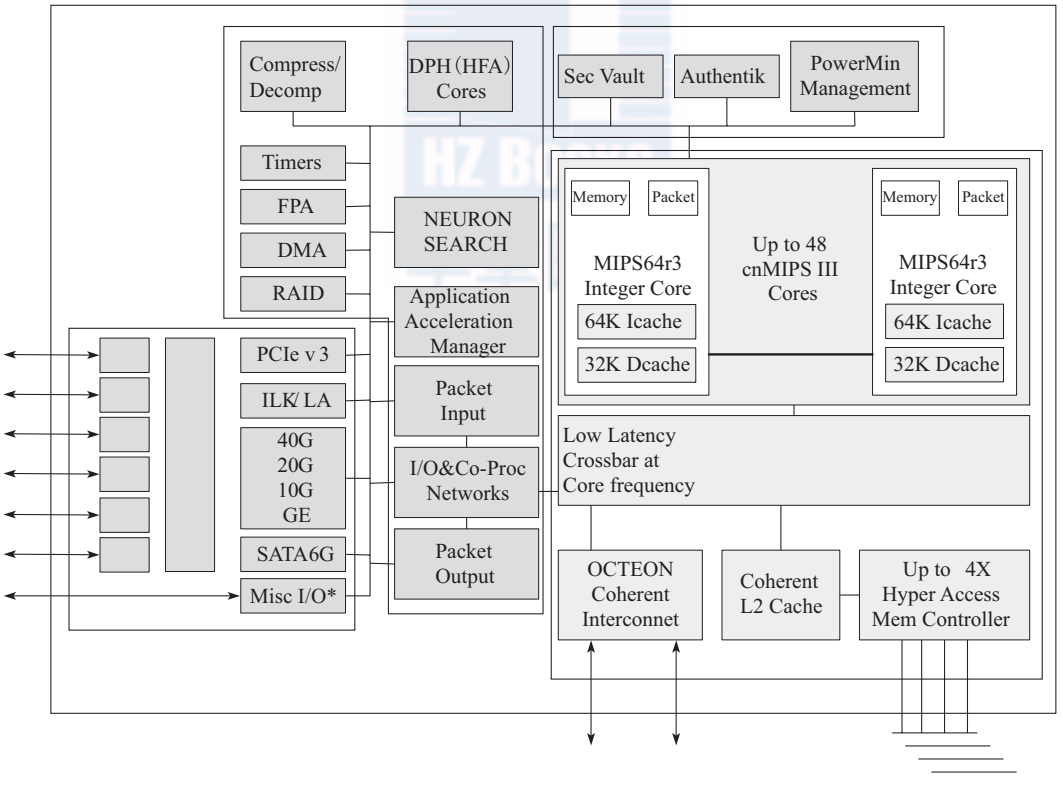


图 1-3 Cavium OCTEON 处理器框图

CPU 核以及众多加速单元和网络接口，组成了一个片上系统（SoC）。在这些 SoC 上，对于可固化的处理（例如，流分类，QoS）交由加速单元完成，而对于灵活的业务逻辑则由众多的通用处理器完成，这种方式有效地融合了软硬件各自的优势。随着软件（例如，DPDK）在 I/O 性能提升上的不断创新，将多核处理器的竞争力提升到一个前所未有的高度，网络负载与虚拟化的融合又催生了 NFV 的潮流。

更多内容请参考相关 Cavium 和 Ezchip 的信息（[Ref1-3] 和 [Ref1-4]）。

1.2　初识 DPDK

本书介绍 DPDK，主要以 IA（Intel Architecture）多核处理器为目标平台。在 IA 上，网络数据包处理远早于 DPDK 而存在。从商业版的 Windows 到开源的 Linux 操作系统，所有跨主机通信几乎都会涉及网络协议栈以及底层网卡驱动对于数据包的处理。然而，低速网络与高速网络处理对系统的要求完全不一样。

1.2.1　IA 不适合进行数据包处理吗

以 Linux 为例，传统网络设备驱动包处理的动作可以概括如下：

- ❑ 数据包到达网卡设备。
- ❑ 网卡设备依据配置进行 DMA 操作。
- ❑ 网卡发送中断，唤醒处理器。
- ❑ 驱动软件填充读写缓冲区数据结构。
- ❑ 数据报文达到内核协议栈，进行高层处理。
- ❑ 如果最终应用在用户态，数据从内核搬移到用户态。
- ❑ 如果最终应用在内核态，在内核继续进行。

随着网络接口带宽从千兆向万兆迈进，原先每个报文就会触发一个中断，中断带来的开销变得突出，大量数据到来会触发频繁的中断开销，导致系统无法承受，因此有人在 Linux 内核中引入了 NAPI 机制，其策略是系统被中断唤醒后，尽量使用轮询的方式一次处理多个数据包，直到网络再次空闲重新转入中断等待。NAPI 策略用于高吞吐的场景，效率提升明显。

一个二层以太网包经过网络设备驱动的处理后，最终大多要交给用户态的应用，图 1-4 的典型网络协议层次 OSI 与 TCP/IP 模型，是一个基础的网络模型与层次，左侧是 OSI 定义的 7 层模型，右侧是 TCP/IP 的具体实现。网络包进入计算机大多需要经过协议处理，在 Linux 系统中 TCP/IP 由 Linux 内核处理。即使在不需要协议处理的场景下，大多数场景下也需要把包从内核的缓冲区复制到用户缓冲区，系统调用以及数据包复制的开销，会直接影响用户态应用从设备直接获得包的能力。而对于多样的网络功能节点来说，TCP/IP 协议栈并不是数据转发节点所必需的。

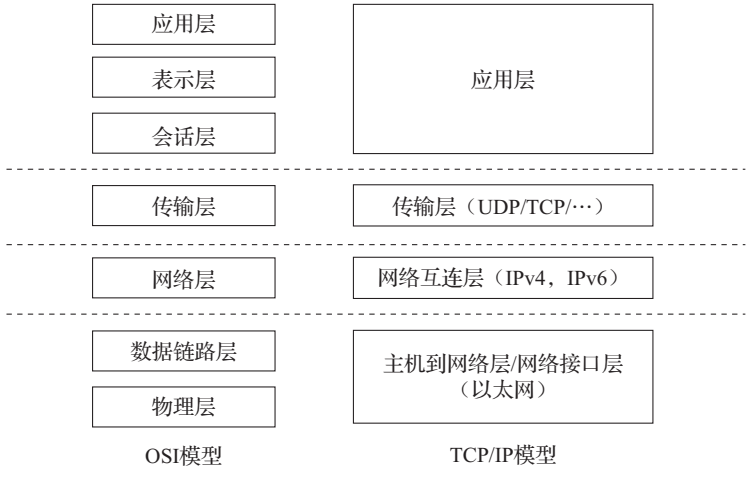


图 1-4 典型网络协议层次 OSI与 TCP/IP 模型

以无线网为例，图 1-5 的无线 4G/LTE 数据面网络协议展示了从基站、基站控制器到无线核心网关的协议层次，可以看到大量处理是在网络二、三、四层进行的。如何让 Linux 这样的面向控制面原生设计的操作系统在包处理上减少不必要的开销一直是一大热点。有个著名的高性能网络 I/O 框架 Netmap，它就是采用共享数据包池的方式，减少内核到用户空间的包复制。

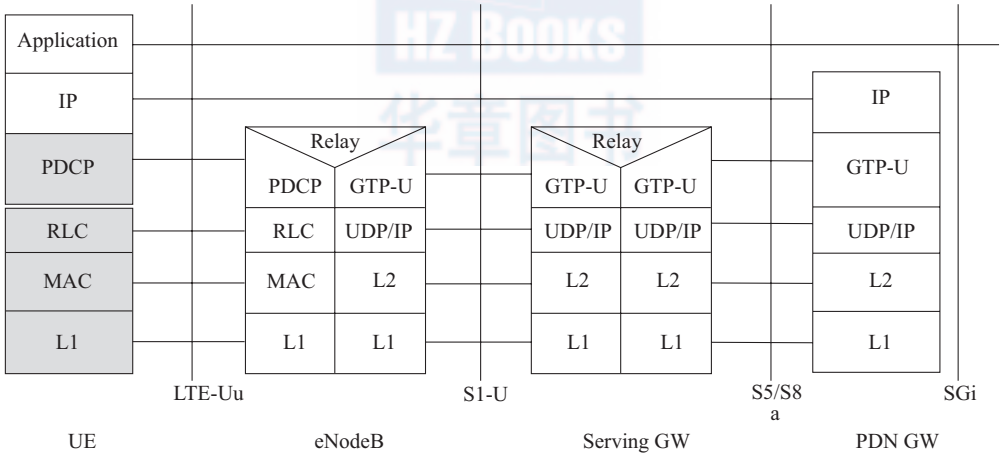


图 1-5 无线 4G/LTE 数据面网络协议

NAPI 与 Netmap 两方面的努力其实已经明显改善了传统 Linux 系统上的包处理能力，那是否还有空间去做得更好呢？作为分时操作系统，Linux 要将 CPU 的执行时间合理地调度给需要运行的任务。相对于公平分时，不可避免的就是适时调度。早些年 CPU 核数比较少，为了每个任务都得到响应处理，进行充分分时，用效率换响应，是一个理想的策略。现今

CPU 核数越来越多，性能越来越强，为了追求极端的高性能高效率，分时就不一定总是上佳的策略。以 Netmap 为例，即便其减少了内核到用户空间的内存复制，但内核驱动的收发包处理和用户态线程依旧由操作系统调度执行，除去任务切换本身的开销，由切换导致的后续 cache 替换（不同任务内存热点不同），对性能也会产生负面的影响。

如果再往实时性方面考虑，传统上，事件从中断发生到应用感知，也是要经过长长的软件处理路径。所以，在 2010 年前采用 IA 处理器的用户会得出这样一个结论，那就是 IA 不适合做包处理。

真的是这样么？在 IA 硬件基础上，包处理能力到底能做到多好，有没有更好的方法评估和优化包处理性能，怎样的软件设计方法能最充分地释放多核 IA 的包处理能力，这些问题都是在 DPDK 出现之前，实实在在地摆在 Intel 工程师面前的原始挑战。

1.2.2　DPDK 最佳实践

如今，DPDK 应该已经很好地回答了 IA 多核处理器是否可以应对高性能数据包处理这个问题。而解决好这样一个问题，也不是用了什么凭空产生的特殊技术，更多的是从工程优化角度的迭代和最佳实践的融合。如果要简单地盘点一下这些技术，大致可以归纳如下。

轮询，这一点很直接，可避免中断上下文切换的开销。之前提到 Linux 也采用该方法改进对大吞吐数据的处理，效果很好。在第 7 章，我们会详细讨论轮询与中断的权衡。

用户态驱动，在这种工作方式下，既规避了不必要的内存拷贝又避免了系统调用。一个间接的影响在于，用户态驱动不受限于内核现有的数据格式和行为定义。对 mbuf 头格式的重定义、对网卡 DMA 操作的重新优化可以获得更好的性能。而用户态驱动也便于快速地迭代优化，甚至对不同场景进行不同的优化组合。在第 6 章中，我们将探讨用户态网卡收发包优化。

亲和性与独占，DPDK 工作在用户态，线程的调度仍然依赖内核。利用线程的 CPU 亲和绑定的方式，特定任务可以被指定只在某个核上工作。好处是可避免线程在不同核间频繁切换，核间线程切换容易导致因 cache miss 和 cache write back 造成的大量性能损失。如果更进一步地限定某些核不参与 Linux 系统调度，就可能使线程独占该核，保证更多 cache hit 的同时，也避免了同一个核内的多任务切换开销。在第 3 章，我们会再展开讨论。

降低访存开销，网络数据包处理是一种典型的 I/O 密集型（I/O bound）工作负载。无论是 CPU 指令还是 DMA，对于内存子系统（Cache+DRAM）都会访问频繁。利用一些已知的高效方法来减少访存的开销能够有效地提升性能。比如利用内存大页能有效降低 TLB miss，比如利用内存多通道的交错访问能有效提高内存访问的有效带宽，再比如利用对于内存非对称性的感知可以避免额外的访存延迟。而 cache 更是几乎所有优化的核心地带，这些有意思而且对性能有直接影响的部分，将在第 2 章进行更细致的介绍。

软件调优，调优本身并不能说是最佳实践。这里其实指代的是一系列调优实践，比如结构的 cache line 对齐，比如数据在多核间访问避免跨 cache line 共享，比如适时地预取数据，

再如多元数据批量操作。这些具体的优化策略散布在 DPDK 各个角落。在第 2 章、第 6 章、第 7 章都会具体涉及。

利用 IA 新硬件技术，IA 的最新指令集以及其他新功能一直是 DPDK 致力挖掘数据包处理性能的源泉。拿 Intel® DDIO 技术来讲，这个 cache 子系统对 DMA 访存的硬件创新直接助推了性能跨越式的增长。有效利用 SIMD（Single Instruction Multiple Data）并结合超标量技术（Superscalar）对数据层面或者对指令层面进行深度并行化，在性能的进一步提升上也行之有效。另外一些指令（比如 cmpxchg），本身就是 lockless 数据结构的基石，而 crc32 指令对与 4 Byte Key 的哈希计算也是改善明显。这些内容，在第 2 章、第 4 章、第 5 章、第 6 章都会有涉及。

充分挖掘网卡的潜能，经过 DPDK I/O 加速的数据包通过 PCIe 网卡进入系统内存，PCIe 外设到系统内存之间的带宽利用效率、数据传送方式（coalesce 操作）等都是直接影响 I/O 性能的因素。在现代网卡中，往往还支持一些分流（如 RSS，FDIR 等）和卸载（如 Chksum，TSO 等）功能。DPDK 充分利用这些硬件加速特性，帮助应用更好地获得直接的性能提升。这些内容将从第 6 章 ~ 第 9 章一一展开。

除了这些基础的最佳实践，本书还会用比较多的篇幅带领大家进入 DPDK I/O 虚拟化的世界。在那里，我们依然从 I/O 的视角，介绍业界广泛使用的两种主流方式，SR-IOV 和 Virtio，帮助大家理解 I/O 硬件虚拟化的支撑技术以及 I/O 软件半虚拟化的技术演进和革新。从第 10 章到第 14 章，我们会围绕着这一主题逐步展开。

随着 DPDK 不断丰满成熟，也将自身逐步拓展到更多的平台和场景。从 Linux 到 FreeBSD，从物理机到虚拟机，从加速网络 I/O 到加速存储 I/O，DPDK 在不同纬度发芽生长。在 NFV 大潮下，无论是 NFVI（例如，virtual switch）还是 VNF，DPDK 都用坚实有力的性能来提供基础设施保障。这些内容将在第 10 章 ~ 第 15 章一一介绍。

当然，在开始后续所有章节之前，让我们概览一下 DPDK 的软件整体框架。

1.2.3 DPDK 框架简介

DPDK 为 IA 上的高速包处理而设计。图 1-6 所示的 DPDK 主要模块分解展示了以基础软件库的形式，为上层应用的开发提供一个高性能的基础 I/O 开发包。它大量利用了有助于包处理的软硬件特性，如大页、缓存行对齐、线程绑定、预取、NUMA、IA 最新指令的利用、Intel® DDIO、内存交叉访问等。

核心库 Core Libs，提供系统抽象、大页内存、缓存池、定时器及无锁环等基础组件。

PMD 库，提供全用户态的驱动，以便通过轮询和线程绑定得到极高的网络吞吐，支持各种本地和虚拟的网卡。

Classify 库，支持精确匹配（Exact Match）、最长匹配（LPM）和通配符匹配（ACL），提供常用包处理的查表操作。

QoS 库，提供网络服务质量相关组件，如限速（Meter）和调度（Sched）。

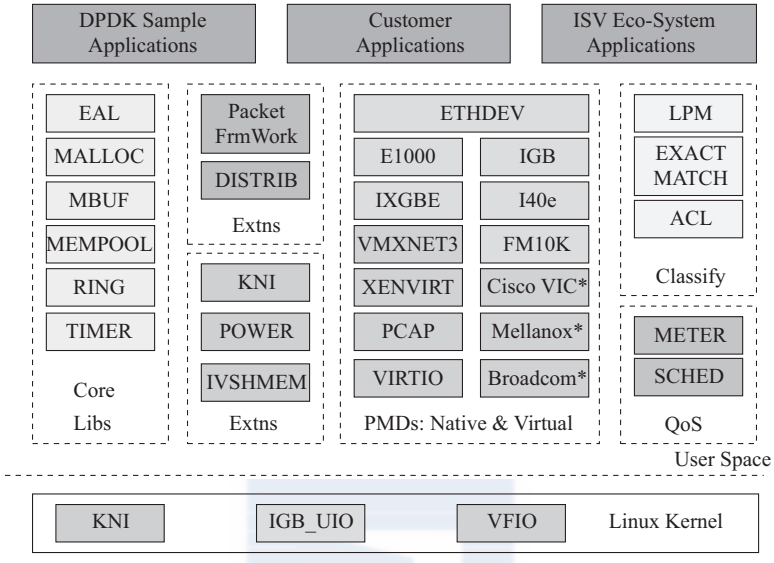


图 1-6　DPDK 主要模块分解

除了这些组件，DPDK 还提供了几个平台特性，比如节能考虑的运行时频率调整（POWER），与 Linux kernel stack 建立快速通道的 KNI（Kernel Network Interface）。而 Packet Framework 和 DISTRIB 为搭建更复杂的多核流水线处理模型提供了基础的组件。

1.2.4　寻找性能优化的天花板

性能优化不是无止境的，所谓天花板可以认为是理论极限，性能优化能做到的就是无限接近这个理论极限。而理论极限也不是单纬度的，当某个纬度接近极限时，可能在另一个纬度会有其他的发现。

我们讨论数据包处理，那首先就看看数据包转发速率是否有天花板。其实包转发的天花板就是理论物理线路上能够传送的最大速率，即线速。那数据包经过网络接口进入内存，会经过 I/O 总线（例如，PCIe bus），I/O 总线也有天花板，实际事务传输不可能超过总线最大带宽。CPU 从 cache 里加载 / 存储 cache line 有没有天花板呢，当然也有，比如 Haswell 处理器能在一个周期加载 64 字节和保存 32 字节。同样内存控制器也有内存读写带宽。这些不同纬度的边界把工作负载包裹起来，而优化就是在这个边界里吹皮球，不断地去接近甚至触碰这样的边界。

由于天花板是理论上的，因此对于前面介绍的一些可量化的天花板，总是能够指导并反映性能优化的优劣。而有些天花板可能很难量化，比如在某个特定频率的 CPU 下每个包所消耗的周期最小能做到多少。对于这样的天花板，可能只能用不断尝试实践的方式，当然不同的方法可能带来不同程度的突破，总的增益越来越少时，就可能是接近天花板的时候。

那 DPDK 在 IA 上提供网络处理能力有多优秀呢？它是否已经能触及一些系统的天花板？在这些天花板中，最难触碰的是哪一个呢？要真正理解这一点，首先要明白在 IA 上包

处理终极挑战的问题是什么，在这之前我们需要先来回顾一下衡量包处理能力的一些常见能力指标。

1.3 解读数据包处理能力

不管什么样的硬件平台，对于包处理都有最基本的性能诉求。一般常被提到的有吞吐、延迟、丢包率、抖动等。对于转发，常会以包转发率（pps，每秒包转发率）而不是比特率（bit/s，每秒比特转发率）来衡量转发能力，这跟包在网络中传输的方式有关。不同大小的包对存储转发的能力要求不尽相同。让我们先来温习一下有效带宽和包转发率概念。

线速（Wire Speed）是线缆中流过的帧理论上支持的最大帧数。

我们用以太网（Ethernet）为例，一般所说的接口带宽，1Gbit/s、10Gbit/s、25Gbit/s、40Gbit/s、100Gbit/s，代表以太接口线路上所能承载的最高传输比特率，其单位是 bit/s（bit per second，位 / 秒）。实际上，不可能每个比特都传输有效数据。以太网每个帧之间会有帧间距（Inter-Packet Gap，IPG），默认帧间距大小为 12 字节。每个帧还有 7 个字节的前导（Preamble），和 1 个字节的帧首定界符（Start Frame Delimiter，SFD）。具体帧格式如图 1-7 所示，有效内容主要是以太网的目的地址、源地址、以太网类型、负载。报文尾部是校验码。

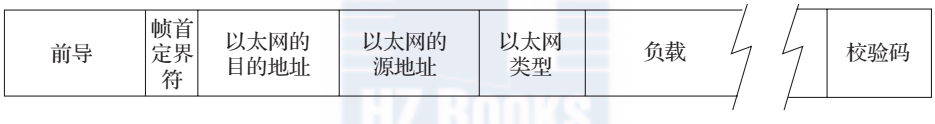


图 1-7 以太帧格式

所以，通常意义上的满速带宽能跑有效数据的吞吐可以由如下公式得到理论帧转发率：

$$\text{帧转发率} = \frac{\text{BitRate}/8}{\text{IPG+Preamble+SFD+PktSize}}$$

而这个最大理论帧转发率的倒数表示了线速情况下先后两个包到达的时间间隔。

按照这个公式，将不同包长按照特定的速率计算可得到一个以太帧转发率，如表 1-1 所示。如果仔细观察，可以发现在相同带宽速率下，包长越小的包，转发率越高，帧间延迟也越小。

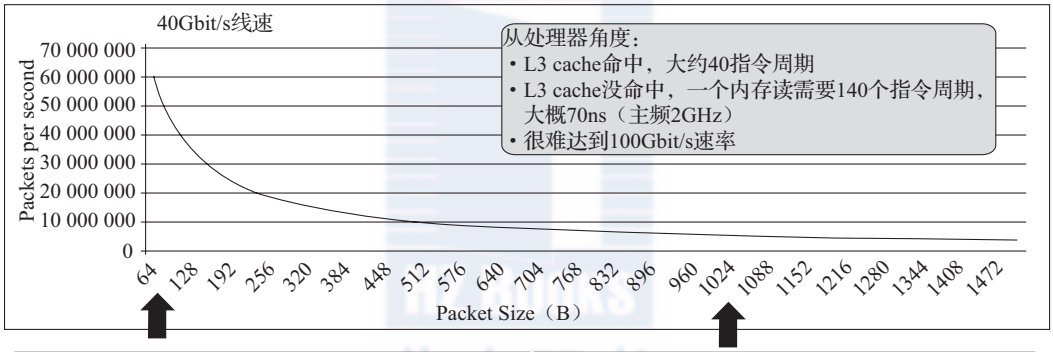
表 1-1 帧转发率

吞吐率	10Gbit/s		25Gbit/s		40Gbit/s	
包长	Mpps	arrival（ns）	Mpps	arrival（ns）	Mpps	arrival（ns）
64	14.88	67.20	37.20	26.88	59.52	16.80
128	8.45	118.40	21.11	47.36	33.78	29.60
256	4.53	220.80	11.32	88.32	18.12	55.20
512	2.35	425.60	5.87	170.24	9.40	106.40
1024	1.20	835.20	2.99	334.08	4.79	208.80

满足什么条件才能达到无阻塞转发的理论上限呢？如果我们把处理一个数据包的整个生命周期看做是工厂的生产流水线，那么就要保证在这个流水线上，不能有任何一级流水处理的延迟超过此时间间隔。理解了这一点，对照表 1-1，就很容易发现，对任何一个数据包处理流水线来说，越小的数据包，挑战总是越大。这样的红线对任何一个硬件平台，对任何一个在硬件平台上设计整体流水线的设计师来说都是无法逃避并需要积极面对的。

1.4　探索 IA 处理器上最艰巨的任务

在通用处理器上处理包的最大挑战是什么？为什么以往通用处理器很少在数据面中扮演重要的角色？如果我们带着这些问题来看数据面上的负载，就会有一个比较直观的理解。这里拿 40Gbit/s 的速率作为考察包转发能力的样本。如图 1-8 所示，曲线为不同大小的包的最大理论转发能力。



Packet Size	64 bytes
40G Packets/second	59.5 Million each way
Packet arrival rate	16.8ns
2GHz Clock cycles	33cycles

网络设备报文大小

Packet Size	1024 bytes
40G Packets/second	4.8 Million each way
Packet arrival rate	208.8ns
2GHz Clock cycles	417cycles

通用服务器报文大小

图 1-8　线速情况下的报文的指令成本

分别截取 64B 和 1024B 数据包长，图 1-8 所示的线速情况下的报文的指令成本能明显地说明不同报文大小给系统带来的巨大差异。就如我们在包转发率那一节中理解的，对于越小的包，相邻包到达的时间间隔就越小，16.8ns vs 208.8ns。假设 CPU 的主频率是 2GHz，要达到理论最大的转发能力，对于 64B 和 1024B 软件分别允许消耗 33 和 417 个时钟周期。在存储转发（store-forward）模型下，报文收发以及查表都需要访存。那就对比一下访存的时钟周期，一次 LLC 命中需要大约 40 个时钟周期，如果 LLC 未命中，一次内存的读就需要 70ns。换句话说，对于 64B 大小的包，即使每次都能命中 LLC，40 个时钟周期依然离 33 有距离。显然，小包处理时延对于通用 CPU 系统架构的挑战是巨大的。

那是否说明 IA 就完全不适合高性能的网络负载呢？答案是否定的。证明这样的结论我

们从两个方面入手，一个是 IA 平台实际能提供的最大能力，另一个是这个能力是否足以应对一定领域的高性能网络负载。

DPDK 的出现充分释放了 IA 平台对包处理的吞吐能力。我们知道，随着吞吐率的上升，中断触发的开销是不能忍受的，DPDK 通过一系列软件优化方法（大页利用，cache 对齐，线程绑定，NUMA 感知，内存通道交叉访问，无锁化数据结构，预取，SIMD 指令利用等）利用 IA 平台硬件特性，提供完整的底层开发支持库。使得单核三层转发可以轻松地突破小包 30Mpps，随着 CPU 封装的核数越来越多，支持的 PCIe 通道数越来越多，整系统的三层转发吞吐在 2 路 CPU 的 Xeon E5-2658 v3 上可以达到 300Mpps。这已经是一个相当可观的转发吞吐能力了。

虽然这个能力不足以覆盖网络中所有端到端的设备场景，但无论在核心网接入侧，还是在数据中心网络中，都已经可以覆盖相当多的场景。

随着数据面可软化的发生，数据面的设计、开发、验证乃至部署会发生一系列的变化。首先，可以采用通用服务器平台，降低专门硬件设计成本；其次，基于 C 语言的开发，就程序员数量以及整个生态都要比专门硬件开发更丰富；另外，灵活可编程的数据面部署也给网络功能虚拟化（NFV）带来了可能，更会进一步推进软件定义网络（SDN）的全面展开。

1.5 软件包处理的潜力——再识 DPDK

DPDK 很好地将 IA 上包处理的性能提升到一个高度，这个高度已经达到很多网络应用场景的最低要求，使得满足要求的场景下对于网络业务软化产生积极的作用。

1.5.1 DPDK 加速网络节点

在理解了 IA 上包处理面临的根本性挑战后，我们会对 DPDK 所取得的性能提升感到异常兴奋。更令人兴奋的是，按照 DPDK 所倡导的方法，随着处理器的每一代更新，在 IA 上的性能提升以很高的斜率不断发酵。当千兆、万兆接口全速转发已不再是问题时，DPDK 已将目标伸向百万兆的接口。

DPDK 软件包内有一个最基本的三层转发实例（l3fwd），可用于测试双路服务器整系统的吞吐能力，实验表明可以达到 220Gbit/s 的数据报文吞吐能力。值得注意的是，除了通过硬件或者软件提升性能之外，如今 DPDK 整系统报文吞吐能力上限已经不再受限于 CPU 的核数，当前瓶颈在于 PCIe（IO 总线）的 LANE 数。换句话说，系统性能的整体 I/O 天花板不再是 CPU，而是系统所提供的所有 PCIe LANE 的带宽，能插入多少个高速以太网接口卡。

在这样的性能基础上，网络节点的软化就成为可能。对于网络节点上运转的不同形态的网络功能，一旦软化并适配到一个通用的硬件平台，随之一个自然的诉求可能就是软硬件解耦。解耦正是网络功能虚拟化（NFV）的一个核心思想，而硬件解耦的多个网络功能在单一通用节点上的隔离共生问题，是另一个核心思想虚拟化诠释的。当然这个虚拟化是广义的，在不同层面可以有不同的支撑技术。

NFV 有很多诉求，业务面高性能，控制面高可用、高可靠、易运维、易管理等。但没有业务面的高性能，后续的便无从谈起。DPDK 始终为高性能业务面提供坚实的支撑，除此以外，DPDK 立足 IA 的 CPU 虚拟化技术和 IO 的虚拟化技术，对各种通道做持续优化改进的同时，也对虚拟交换（vswitch）的转发面进化做出积极贡献。应对绝对高吞吐能力的要求，DPDK 支持各种 I/O 的 SR-IOV 接口；应对高性能虚拟主机网络的要求，DPDK 支持标准 virtio 接口；对虚拟化平台的支撑，DPDK 从 KVM、VMWARE、XEN 的 hypervisor 到容器技术，可谓全平台覆盖。

可以说，在如火如荼的网络变革的大背景下，DPDK 以强劲的驱动力加速各种虚拟化的网络功能部署到现实的网络节点上。

1.5.2　DPDK 加速计算节点

DPDK 之于网络节点，主要集中在数据面转发方面，这个很容易理解；对于计算节点，DPDK 也拥有很多潜在的机会。

C10K 是 IT 界的一个著名命题，甚至后续衍生出了关于 C1M 和 C10M 的讨论。其阐述的一个核心问题就是，随着互联网发展，随着数据中心接口带宽不断提升，计算节点上各种互联网服务对于高并发下的高吞吐有着越来越高的要求。详见［ Ref1-5 ］。

但是单一接口带宽的提高并不能直接导致高并发、高吞吐服务的发生，即使用到了一系列系统方法（异步非阻塞，线程等），但网络服务受限于内核协议栈多核水平扩展上的不足以及建立拆除连接的高开销，开始逐渐阻碍进一步高并发下高带宽的要求。另一方面，内核协议栈需要考虑更广泛的支持，并不能为特定的应用做特殊优化，一般只能使用系统参数进行调优。

当然，内核协议栈也在不断改进，而以应用为中心的趋势也会不断推动用户态协议栈的涌现。有基于 BSD 协议栈移植的，有基于多核模型重写的原型设计，也有将整个 Linux 内核包装成库的。它们大多支持以 DPDK 作为 I/O 引擎，有些也将 DPDK 的一些优化想法加入到协议栈的优化中，取得了比较好的效果。

可以说，由 DPDK 加速的用户态协议栈将会越来越多地支撑起计算节点上的网络服务。

1.5.3　DPDK 加速存储节点

除了在网络、计算节点的应用机会之外，DPDK 的足迹还渗透到存储领域。Intel® 最近开源了 SPDK（Storage Performance Development Kit），一款存储加速开发套件，其主要的应用场景是 iSCSI 性能加速。目前 iSCSI 系统包括前端和后端两个部分，在前端，DPDK 提供网络 I/O 加速，加上一套用户态 TCP/IP 协议栈（目前还不包含在开源包中），以流水线的工作方式支撑起基于 iSCSI 的应用；在后端，将 DPDK 用户态轮询驱动的方式实践在 NVMe 上，PMD 的 NVMe 驱动加速了后端存储访问。这样一个端到端的整体方案，用数据证明了卓有成效的 IOPS 性能提升。SPDK 的详细介绍见：https://01.org/spdk。

可以说，理解 DPDK 的核心方法，并加以恰当地实践，可以将 I/O 在 IA 多核的性能提

升有效地拓展到更多的应用领域，并产生积极的意义。

1.5.4 DPDK 的方法论

DPDK 采用了很多具体优化方法来达到性能的提升，有一些是利用 IA 软件优化的最佳实践方法，还有一些是利用了 IA 的处理器特性。这里希望脱离这一个个技术细节，尝试着去还原一些核心的指导思想，试图从方法论的角度去探寻 DPDK 成功背后的原因，但愿这样的方法论总结，可以在开拓未知领域的过程中对大家有所助益。

1. 专用负载下的针对性软件优化

专用处理器通过硬件架构专用优化来达到高性能，DPDK 则利用通用处理器，通过优化的专用化底层软件来达到期望的高性能。这要求 DPDK 尽可能利用一切平台（CPU，芯片组，PCIe 以及网卡）特性，并针对网络负载的特点，做针对性的优化，以发掘通用平台在某一专用领域的最大能力。

2. 追求可水平扩展的性能

利用多核并行计算技术，提高性能和水平扩展能力。对于产生的并发干扰，遵循临界区越薄越好、临界区碰撞越少越好的指导原则。数据尽可能本地化和无锁化，追求吞吐率随核数增加而线性增长。

3. 向 Cache 索求极致的实现优化性能

相比于系统优化和算法优化，实现优化往往较少被提及。实现优化对开发者的要求体现在需要对处理器体系结构有所了解。DPDK 可谓集大量的实现优化之大成，而这些方法多数围绕着 Cache 进行，可以说能娴熟地驾驭好 Cache，在追求极致性能的路上就已经成功了一半。

4. 理论分析结合实践推导

性能的天花板在哪，调优是否还有空间，是否值得花更多的功夫继续深入，这些问题有时很难直接找到答案。分析、推测、做原型、跑数据、再分析，通过这样的螺旋式上升，慢慢逼近最优解，往往是实践道路上的导航明灯。条件允许下，有依据的理论量化计算，可以更可靠地明确优化目标。

1.6 从融合的角度看 DPDK

这是一个最好的时代，也是一个最坏的时代。不可否认的是，这就是一个融合的时代。

随着云计算的推进，ICT 这个词逐渐在各类技术研讨会上被提及。云计算的定义虽然有各种版本，但大体都包含了对网络基础设施以及对大数据处理的基本要求，这也是 IT 与 CT 技术融合的推动力。

那这和 DPDK 有关系吗？还真有！我们知道云计算的对象是数据，数据在云上加工，可

还是要通过各种载体落到地上。在各种载体中最广泛使用的当属 IP，它是整个互联网蓬勃发展的基石。高效的数据处理总是离不开高效的数据承载网络。

教科书说到网络总会讲到那经典的 7 层模型，最低层是物理层，最高层是应用层。名副其实的是，纵观各类能联网的设备，从终端设备到网络设备再到数据中心服务器，还真是越靠近物理层的处理以硬件为主，越靠近应用层的处理以软件为主。这当然不是巧合，其中深谙了一个原则，越是能标准化的，越要追求极简极速，所以硬件当仁不让，一旦进入多样性可变性强的领域，软件往往能发挥作用。但没有绝对和一成不变，因为很多中间地带更多的是权衡。

DPDK 是一个软件优化库，目标是在通用处理器上发挥极致的包能力，以媲美硬件级的性能。当然软件是跑在硬件上的，如果看整个包处理的硬件平台，软硬件融合的趋势也相当明显。各类硬件加速引擎逐渐融入 CPU 构成异构 SoC（System On-Chip），随着 Intel® 对 Altera® 收购的完成，CPU+FPGA 这一对组合也给足了我们想象的空间，可以说包处理正处在一个快速变革的时代。

1.7　实例

在对 DPDK 的原理和代码展开进一步解析之前，先看一些小而简单的例子，建立一个形象上的认知。

1）helloworld，启动基础运行环境，DPDK 构建了一个基于操作系统的，但适合包处理的软件运行环境，你可以认为这是个 mini-OS。最早期 DPDK，可以完全运行在没有操作系统的物理核（bare-metal）上，这部分代码现在不在主流的开源包中。

2）skeleton，最精简的单核报文收发骨架，也许这是当前世界上运行最快的报文进出测试程序。

3）l3fwd，三层转发是 DPDK 用于发布性能测试指标的主要应用。

1.7.1　HelloWorld

DPDK 里的 HelloWorld 是最基础的入门程序，代码简短，功能也不复杂。它建立了一个多核（线程）运行的基础环境，每个线程会打印“ hello from core #”，core # 是由操作系统管理的。如无特别说明，本文里的 DPDK 线程与硬件线程是一一对应的关系。从代码角度，rte 是指 runtime environment，eal 是指 environment abstraction layer。DPDK 的主要对外函数接口都以 rte_ 作为前缀，抽象化函数接口是典型软件设计思路，可以帮助 DPDK 运行在多个操作系统上，DPDK 官方支持 Linux 与 FreeBSD。和多数并行处理系统类似，DPDK 也有主线程、从线程的差异。

```
int
main(int argc, char **argv)
```

```
{
    int ret;
    unsigned lcore_id;

    ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_panic( “Cannot init EAL\n” );

    /* call lcore_hello() on every slave lcore */
        RTE_LCORE_FOREACH_SLAVE(lcore_id) {
        rte_eal_remote_launch(lcore_hello, NULL, lcore_id);
    }

    /* call it on master lcore too */
    lcore_hello(NULL);

    rte_eal_mp_wait_lcore();
    return 0;
}
```

1. 初始化基础运行环境

主线程运行入口是 main 函数，调用了 rte_eal_init 入口函数，启动基础运行环境。

```
int rte_eal_init(int argc, char **argv);
```

入口参数是启动 DPDK 的命令行，可以是长长的一串很复杂的设置，需要深入了解的读者可以查看 DPDK 相关的文档与源代码 \lib\librte_eal\common\eal_common_options.c。对于 HelloWorld 这个实例，最需要的参数是“ -c <core mask>”，线程掩码（core mask）指定了需要参与运行的线程（核）集合。rte_eal_init 本身所完成的工作很复杂，它读取入口参数，解析并保存作为 DPDK 运行的系统信息，依赖这些信息，构建一个针对包处理设计的运行环境。主要动作分解如下

- ❑ 配置初始化
- ❑ 内存初始化
- ❑ 内存池初始化
- ❑ 队列初始化
- ❑ 告警初始化
- ❑ 中断初始化
- ❑ PCI 初始化
- ❑ 定时器初始化
- ❑ 检测内存本地化（NUMA）
- ❑ 插件初始化
- ❑ 主线程初始化
- ❑ 轮询设备初始化

❑ 建立主从线程通道
❑ 将从线程设置在等待模式
❑ PCI 设备的探测与初始化

对于 DPDK 库的使用者，这些操作已经被 EAL 封装起来，接口清晰。如果需要对 DPDK 进行深度定制，二次开发，需要仔细研究内部操作，这里不做详解。

2. 多核运行初始化

DPDK 面向多核设计，程序会试图独占运行在逻辑核（lcore）上。main 函数里重要的是启动多核运行环境，RTE_LCORE_FOREACH_SLAVE（lcore_id）如名所示，遍历所有 EAL 指定可以使用的 lcore，然后通过 rte_eal_remote_launch 在每个 lcore 上，启动被指定的线程。

```
int rte_eal_remote_launch(int (*f)(void *),
    void *arg, unsigned slave_id);
```

第一个参数是从线程，是被征召的线程；
第二个参数是传给从线程的参数；
第三个参数是指定的逻辑核，从线程会执行在这个 core 上。

具体来说，int rte_eal_remote_launch(lcore_hello, NULL, lcore_id);
参数 lcore_id 指定了从线程 ID，运行入口函数 lcore_hello。
运行函数 lcore_hello，它读取自己的逻辑核编号（lcore_id），打印出“hello from core #”

```
static int
lcore_hello(__attribute__((unused)) void *arg)
{
    unsigned lcore_id;
    lcore_id = rte_lcore_id();
    printf("hello from core %u\n", lcore_id);
    return 0;
}
```

这是个简单示例，从线程很快就完成了指定工作，在更真实的场景里，这个从线程会是一个循环运行的处理过程。

1.7.2　Skeleton

DPDK 为多核设计，但这是单核实例，设计初衷是实现一个最简单的报文收发示例，对收入报文不做任何处理直接发送。整个代码非常精简，可以用于平台的单核报文出入性能测试。

主要处理函数 main 的处理逻辑如下（伪码），调用 rte_eal_init 初始化运行环境，检查网络接口数，据此分配内存池 rte_pktmbuf_pool_create，入口参数是指定 rte_socket_id()，考虑了本地内存使用的范例。调用 port_init(portid, mbuf_pool) 初始化网口的配置，最后调用 lcore_main() 进行主处理流程。

```
int main(int argc, char *argv[])
{
    struct rte_mempool *mbuf_pool;
    unsigned nb_ports;
    uint8_t portid;

    /* Initialize the Environment Abstraction Layer (EAL). */
    int ret = rte_eal_init(argc, argv);

    /* Check that there is an even number of ports t send/receive on. */
    nb_ports = rte_eth_dev_count();
    if (nb_ports < 2 || (nb_ports & 1))
        rte_exit(EXIT_FAILURE, "Error: number of ports must be even\n");

    /* Creates a new mempool in memory to hold the mbufs. */
    mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", NUM_MBUFS * nb_ports,
        MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());

    /* Initialize all ports. */
    for (portid = 0; portid < nb_ports; portid++)
        if (port_init(portid, mbuf_pool) != 0)
            rte_exit(EXIT_FAILURE, "Cannot init port %"PRIu8 "\n",
                    portid);

    /* Call lcore_main on the master core only. */
    lcore_main();
    return 0;
}
```

网口初始化流程：

```
port_init(uint8_t port, struct rte_mempool *mbuf_pool)
```

首先对指定端口设置队列数，基于简单原则，本例只指定单队列。在收发两个方向上，基于端口与队列进行配置设置，缓冲区进行关联设置。如不指定配置信息，则使用默认配置。

网口设置：对指定端口设置接收、发送方向的队列数目，依据配置信息来指定端口功能

```
int rte_eth_dev_configure(uint8_t port_id, uint16_t nb_rx_q,
            uint16_t nb_tx_q, const struct rte_eth_conf *dev_conf)
```

队列初始化：对指定端口的某个队列，指定内存、描述符数量、报文缓冲区，并且对队列进行配置

```
int rte_eth_rx_queue_setup(uint8_t port_id, uint16_t rx_queue_id,
              uint16_t nb_rx_desc, unsigned int socket_id,
              const struct rte_eth_rxconf *rx_conf,
              struct rte_mempool *mp)
int rte_eth_tx_queue_setup(uint8_t port_id, uint16_t tx_queue_id,
```

```
                uint16_t nb_tx_desc, unsigned int socket_id,
                const struct rte_eth_txconf *tx_conf)
```

网口设置：初始化配置结束后，启动端口 int rte_eth_dev_start(uint8_t port_id)；完成后，读取 MAC 地址，打开网卡的混杂模式设置，允许所有报文进入。

```
static inline int
port_init(uint8_t port, struct rte_mempool *mbuf_pool)
{
    struct rte_eth_conf port_conf = port_conf_default;
    const uint16_t rx_rings = 1, tx_rings = 1;

    /* Configure the Ethernet device. */
    retval = rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);

    /* Allocate and set up 1 RX queue per Ethernet port. */
    for (q = 0; q < rx_rings; q++) {
        retval = rte_eth_rx_queue_setup(port, q, RX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL, mbuf_pool);
    }

    /* Allocate and set up 1 TX queue per Ethernet port. */
    for (q = 0; q < tx_rings; q++) {
        retval = rte_eth_tx_queue_setup(port, q, TX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL);
    }

    /* Start the Ethernet port. */
    retval = rte_eth_dev_start(port);

    /* Display the port MAC address. */
    struct ether_addr addr;
    rte_eth_macaddr_get(port, &addr);

    /* Enable RX in promiscuous mode for the Ethernet device. */
    rte_eth_promiscuous_enable(port);
    return 0;
}
```

网口收发报文循环收发在 lcore_main 中有个简单实现，因为是示例，为保证性能，首先检测 CPU 与网卡的 Socket 是否最优适配，建议使用本地 CPU 就近操作网卡，后续章节有详细说明。数据收发循环非常简单，为高速报文进出定义了 burst 的收发函数如下，4 个参数意义非常直观：端口，队列，报文缓冲区以及收发包数。

基于端口队列的报文收发函数：

```
static inline uint16_t rte_eth_rx_burst(uint8_t port_id, uint16_t queue_id,
struct rte_mbuf **rx_pkts, const uint16_t nb_pkts)
static inline uint16_t rte_eth_tx_burst(uint8_t port_id, uint16_t queue_id,
struct rte_mbuf **tx_pkts, uint16_t nb_pkts)
```

这就构成了最基本的 DPDK 报文收发展示。可以看到，此处不涉及任何具体网卡形态，软件接口对硬件没有依赖。

```
static __attribute__((noreturn)) void lcore_main(void)
{
    const uint8_t nb_ports = rte_eth_dev_count();
    uint8_t port;
    for (port = 0; port < nb_ports; port++)
        if (rte_eth_dev_socket_id(port) > 0 &&
                rte_eth_dev_socket_id(port) !=
                        (int)rte_socket_id())
            printf("WARNING, port %u is on remote NUMA node to "
                    "polling thread.\n\tPerformance will "
                    "not be optimal.\n", port);

    /* Run until the application is quit or killed. */
    for (;;) {
        /*
         * Receive packets on a port and forward them on the paired
         * port. The mapping is 0 -> 1, 1 -> 0, 2 -> 3, 3 -> 2, etc.
         */
        for (port = 0; port < nb_ports; port++) {

            /* Get burst of RX packes, from first port of pair. */
            struct rte_mbuf *bufs[BURST_SIZE];
            const uint16_t nb_rx = rte_eth_rx_burst(port, 0,
                    bufs, BURST_SIZE);

            if (unlikely(nb_rx == 0))
                continue;

            /* Send burst of TX packets, to second port of pair. */
            const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
                    bufs, nb_rx);

            /* Free any unsent packets. */
            if (unlikely(nb_tx < nb_rx)) {
                uint16_t buf;
                for (buf = nb_tx; buf < nb_rx; buf++)
                    rte_pktmbuf_free(bufs[buf]);
            }
        }
    }
}
}
```

1.7.3 L3fwd

这是 DPDK 中最流行的例子，也是发布 DPDK 性能测试的例子。如果将 PCIE 插槽上填满高速网卡，将网口与大流量测试仪表连接，它能展示在双路服务器平台具备 200Gbit/s 的

转发能力。数据包被收入系统后，会查询 IP 报文头部，依据目标地址进行路由查找，发现目的端口，修改 IP 头部后，将报文从目的端口送出。路由查找有两种方式，一种方式是基于目标 IP 地址的完全匹配（exact match），另一种方式是基于路由表的最长掩码匹配（Longest Prefix Match，LPM）。三层转发的实例代码文件有 2700 多行（含空行与注释行），整体逻辑其实很简单，是前续 HelloWorld 与 Skeleton 的结合体。

启动这个例子，指定命令参数格式如下：

```
./build/l3fwd [EAL options] -- -p PORTMASK [-P]
--config(port,queue,lcore)[,(port,queue,lcore)]
```

命令参数分为两个部分，以“ --”为分界线，分界线右边的参数是三层转发的私有命令选项。左边则是 DPDK 的 EAL Options。

- [EAL Options] 是 DPDK 运行环境的输入配置选项，输入命令会交给 rte_eal_init 处理；
- PORTMASK 依据掩码选择端口，DPDK 启动时会搜索系统认识的 PCIe 设备，依据黑白名单原则来决定是否接管，早期版本可能会接管所有端口，断开网络连接。现在可以通过脚本绑定端口，具体可以参见 http://www.dpdk.org/browse/dpdk/tree/tools/dpdk_nic_bind.py
- config 选项指定 (port,queue,lcore)，用指定线程处理对应的端口的队列。要实现 200Gbit/s 的转发，需要大量线程（核）参与，并行转发。

端口	队列	线程	描述
0	0	0	线程 0 处理端口 0 的队列 0
0	1	2	线程 2 处理端口 0 的队列 1
1	0	1	线程 1 处理端口 1 的队列 0
1	1	3	线程 3 处理端口 1 的队列 1

先来看主线程流程 main 的处理流程，因为和 HelloWorld 与 Skeleton 类似，不详细叙述。

```
初始化运行环境：rte_eal_init(argc, argv);
分析入参：parse_args(argc, argv)
初始化 lcore 与 port 配置
端口与队列初始化，类似 Skeleton 处理
端口启动，使能混杂模式
启动从线程，令其运行 main_loop()
```

从线程执行 main_loop() 的主要步骤如下：

```
读取自己的 lcore 信息完成配置；
读取关联的接收与发送队列信息；
进入循环处理：
{
    向指定队列批量发送报文；
    从指定队列批量接收报文；
```

```
    批量转发接收到报文;
}
```

向指定队列批量发送报文，从指定队列批量接收报文，此前已经介绍了 DPDK 的收发函数。批量转发接收到的报文是处理的主体，提供了基于 Hash 的完全匹配转发，也可以基于最长匹配原则（LPM）进行转发。转发路由查找方式可以由编译配置选择。除了路由转发算法的差异，下面的例子还包括基于 multi buffer 原理的代码实现。在 #if (ENABLE_MULTI_BUFFER_OPTIMIZE == 1) 的路径下，一次处理 8 个报文。和普通的软件编程不同，初次见到的程序员会觉得奇怪。它的实现有效利用了处理器内部的乱序执行和并行处理能力，能显著提高转发性能。

```
for (j = 0; j < n; j += 8) {
    uint32_t pkt_type =
        pkts_burst[j]->packet_type &
        pkts_burst[j+1]->packet_type &
        pkts_burst[j+2]->packet_type &
        pkts_burst[j+3]->packet_type &
        pkts_burst[j+4]->packet_type &
        pkts_burst[j+5]->packet_type &
        pkts_burst[j+6]->packet_type &
        pkts_burst[j+7]->packet_type;
    if (pkt_type & RTE_PTYPE_L3_IPV4) {
        simple_ipv4_fwd_8pkts(&pkts_burst[j], portid, qconf);
    } else if (pkt_type & RTE_PTYPE_L3_IPV6) {
        simple_ipv6_fwd_8pkts(&pkts_burst[j], portid, qconf);
    } else {
        l3fwd_simple_forward(pkts_burst[j],portid, qconf);
        l3fwd_simple_forward(pkts_burst[j+1],portid, qconf);
        l3fwd_simple_forward(pkts_burst[j+2],portid, qconf);
        l3fwd_simple_forward(pkts_burst[j+3],portid, qconf);
        l3fwd_simple_forward(pkts_burst[j+4],portid, qconf);
        l3fwd_simple_forward(pkts_burst[j+5],portid, qconf);
        l3fwd_simple_forward(pkts_burst[j+6],portid, qconf);
        l3fwd_simple_forward(pkts_burst[j+7],portid, qconf);
        }
    }
    for (; j < nb_rx ; j++) {
        l3fwd_simple_forward(pkts_burst[j],portid, qconf);
    }
```

依据 IP 头部的五元组信息，利用 rte_hash_lookup 来查询目标端口。

```
mask0 = _mm_set_epi32(ALL_32_BITS, ALL_32_BITS, ALL_32_BITS, BIT_8_TO_15);
ipv4_hdr = (uint8_t *)ipv4_hdr + offsetof(struct ipv4_hdr, time_to_live);
__m128i data = _mm_loadu_si128((__m128i*)(ipv4_hdr));
/* Get 5 tuple: dst port, src port, dst IP address, src IP address and protocol */
key.xmm = _mm_and_si128(data, mask0);
/* Find destination port */
```

```
ret = rte_hash_lookup(ipv4_l3fwd_lookup_struct, (const void *)&key);
return (uint8_t)((ret < 0)? portid : ipv4_l3fwd_out_if[ret]);
```

这段代码在读取报文头部信息时，将整个头部导入了基于 SSE 的矢量寄存器（128 位宽），并对内部进行了掩码 mask0 运算，得到 key，然后把 key 作为入口参数送入 rte_hash_lookup 运算。同样的操作运算还展示在对 IPv6 的处理上，可以在代码中参考。

我们并不计划在本节将读者带入代码陷阱中，实际上本书总体上也没有偏重代码讲解，而是在原理上进行解析。如果读者希望了解详细完整的编程指南，可以参考 DPDK 的网站。

1.8　小结

什么是 DPDK？相信读完本章，读者应该对它有了一个整体的认识。DPDK 立足通用多核处理器，经过软件优化的不断摸索，实践出一套行之有效的方法，在 IA 数据包处理上取得重大性能突破。随着软硬件解耦的趋势，DPDK 已经成为 NFV 事实上的数据面基石。着眼未来，无论是网络节点，还是计算节点，或是存储节点，这些云服务的基础设施都有机会因 DPDK 而得到加速。在 IT 和 CT 不断融合的过程中，在运营商网络和数据中心网络持续 SDN 化的过程中，在云基础设施对数据网络性能孜孜不倦的追求中，DPDK 将扮演越来越重要的作用。

Chapter 2 第 2 章

Cache 和内存

2.1 存储系统简介

一般而言，存储系统不仅仅指用于存储数据的磁盘、磁带和光盘存储器等，还包括内存和 CPU 内部的 Cache。当处理完毕之后，系统还要提供数据存储的服务。存储系统的性能和系统的处理能力息息相关，如果 CPU 性能很好，处理速度很快，但是配备的存储系统吞吐率不够或者性能不够好，那 CPU 也只能处于忙等待，从而导致处理数据的能力下降。接下来本章会讨论 Cache 和内存，对于磁盘和磁带等永久性存储系统，在此不作讨论。

2.1.1 系统架构的演进

在当今时代，一个处理器通常包含多个核心（Core），集成 Cache 子系统，内存子系统通过内部或外部总线与其通信。

在经典计算机系统中一般都有两个标准化的部分：北桥（North Bridge）和南桥（South Bridge）。它们是处理器和内存以及其他外设沟通的渠道。处理器和内存系统通过前端总线（Front Side Bus，FSB）相连，当处理器需要读取或者写回数据时，就通过前端总线和内存控制器通信。图 2-1 给出了处理器、内存、南北桥以及其他总线之间的关系。

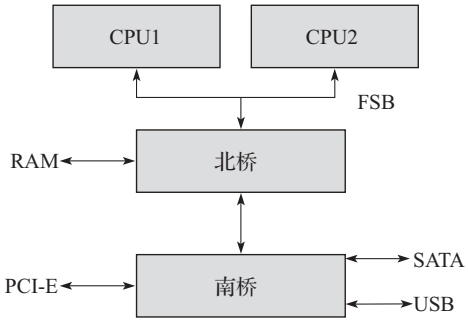


图 2-1 计算机系统中的南北桥示意图

我们可以看到，该架构所有的处理器共用一

条前端总线与北桥相连。北桥也称为主桥（Host Bridge），主要用来处理高速信号，通常负责与处理器的联系，并控制内存 AGP、PCI 数据在北桥内部传输。而北桥中往往集成了一个内存控制器（最近几年英特尔的处理器已经把内存控制器集成到了处理器内部），根据不同的内存，比如 SRAM、DRAM、SDRAM，集成的内存控制器也不一样。南桥也称为 IO 桥（IO bridge），负责 I/O 总线之间的通信，比如 PCI 总线、SATA、USB 等，可以连接光驱、硬盘、键盘灯设备交换数据。在这种系统中，所有的数据交换都需要通过北桥：

1）处理器访问内存需要通过北桥。

2）处理器访问所有的外设都需要通过北桥。

3）处理器之间的数据交换也需要通过北桥。

4）挂在南桥的所有设备访问内存也需要通过北桥。

可以看出，这种系统的瓶颈就在北桥中。当北桥出现拥塞时，所有的设备和处理器都要瘫痪。这种系统设计的另外一个瓶颈体现在对内存的访问上。不管是处理器或者显卡，还是南桥的硬盘、网卡或者光驱，都需要频繁访问内存，当这些设备都争相访问内存时，增大了对北桥带宽的竞争，而且北桥到内存之间也只有一条总线。

为了改善对内存的访问瓶颈，出现了另外一种系统设计，内存控制器并没有被集成在北桥中，而是被单独隔离出来以协调北桥与某个相应的内存之间的交互，如图 2-2 所示。这样的话，北桥可以和多个内存相连。

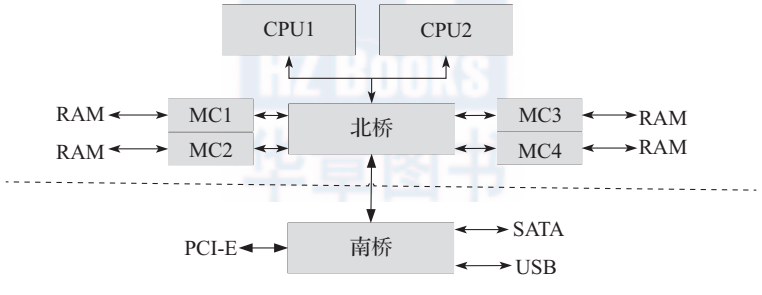


图 2-2　更为复杂的南北桥示意图

图 2-2 所示的这种架构增加了内存的访问带宽，缓解了不同设备对同一内存访问的拥塞问题，但是却没有改进单一北桥芯片的瓶颈的问题。

为了解决这个瓶颈，产生了如图 2-3 所示的 NUMA（Non-Uniform Memory Architecture，非一致性内存架构）系统。

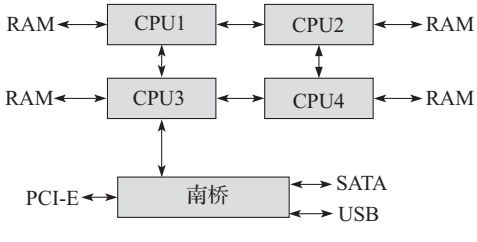


图 2-3　NUMA 系统

在这种架构下，在一个配有四核的机器中，不需要一个复杂的北桥就能将内存带宽增加到以前的四倍。当然，这样的架构也存在缺点。该系统中，访问内存所花的时间和处理器相关。之所

以和处理器相关是因为该系统每个处理器都有本地内存（Local memory），访问本地内存的时间很短，而访问远程内存（remote memory），即其他处理器的本地内存，需要通过额外的总线！对于某个处理器来说，当其要访问其他的内存时，轻者要经过另外一个处理器，重者要经过 2 个处理器，才能达到访问非本地内存的目的，因此内存与处理器的“距离”不同，访问的时间也有所差异，对于 NUMA，后续章节会给出更详细的介绍。

2.1.2 内存子系统

为了了解内存子系统，首先需要解释一下和内存相关的常用用语。

1）RAM（Random Access Memory）：随机访问存储器

2）SRAM（Static RAM）：静态随机访问存储器

3）DRAM（Dynamic RAM）：动态随机访问存储器。

4）SDRAM（Synchronous DRAM）：同步动态随机访问存储器。

5）DDR（Double Data Rate SDRAM）：双数据速率 SDRAM。

6）DDR2：第二代 DDR。

7）DDR3：第三代 DDR。

8）DDR4：第四代 DDR。

1. SRAM

SRAM 内部有一块芯片结构维持信息，通常非常快，但是成本相对 DRAM 很高，应用时容量不会很大，因而不能作用系统的主要内存。一般处理器内部的 Cache 就是采用 SRAM。

2. DRAM

DRAM 通常是系统的主要内存，动态表示信息是存储在集成电路的电容器内的，由于电容器会自动放电，为了避免数据丢失，需要定期充电。通常，内存控制器会负责定期充电的操作。不过随着更好技术的提出，该技术已经被淘汰。

3. SDRAM

一般 DRAM 都是采用异步时钟进行同步，而 SDRAM 则是采用同步时钟进行同步。通常，采用 SDRAM 结构的系统会使处理器和内存通过一个相同的时钟锁在一起，从而使处理器和内存能够共享一个时钟周期，以相同的速度同步工作。该时钟会驱动一个内部的有限状态机，能够采用流水线的方式处理多个读写请求。

SDRAM 采用分布式架构，内含多个存储块（Bank），在一个时钟周期内，它能够独立地访问每个存储块，从而可以多次进行读写操作，增加了内存系统的吞吐率。

SDRAM 技术广泛用在计算机行业中，随着该技术的提出，又出现了 DDR（也称为 DDR1），DDR2，DDR3。最新的 DDR4 技术标准也在 2014 年下半年发布。

2.2　Cache 系统简介

随着计算机行业的飞速发展，CPU 的速度和内存的大小都发生了翻天覆地的变化。英特尔公司在 1982 年推出 80286 芯片的时候，处理器内部含有 13.4 万个晶体管，时钟频率只有 6MHz，内部和外部数据总线只有 16 位，地址总线 24 位，可寻址内存大小 16MB。

而英特尔公司在 2014 年推出的 Haswell 处理器的时候，处理器内部仅处理器本身就包含了 17 亿个晶体管，还不包括 Cache 和 GPU 这种复杂部件。时钟频率达到 3.8GHz，数据总线和地址总线也都扩展到了 64 位，可以寻址的内存大小也已经开始以 TB（1T=1024GB）计算。

在处理器速度不断增加的形势下，处理器处理数据的能力也得到大大提升。但是，数据是存储在内存中的，虽然随着 DDR2、DDR3、DDR4 的新技术不断推出，内存的吞吐率得到了大大提升，但是相对于处理器来讲，仍然非常慢。一般来讲，处理器要从内存中直接读取数据都要花大概几百个时钟周期，在这几百个时钟周期内，处理器除了等待什么也不能做。在这种环境下，才提出了 Cache 的概念，其目的就是为了匹配处理器和内存之间存在的巨大的速度鸿沟。

2.2.1　Cache 的种类

一般来讲，Cache 由三级组成，之所以对 Cache 进行分级，也是从成本和生产工艺的角度考虑的。一级（L1）最快，但是容量最小；三级（LLC，Last Level Cache）最慢，但是容量最大，在早期计算机系统中，这一级 Cache 也可以省略。不过在当今时代，大多数处理器都会包含这一级 Cache。

Cache 是一种 SRAM，在早期计算机系统中，一般一级和二级 Cache 集成在处理器内部，三级 Cache 集成在主板上，这样做的主要原因是生产工艺的问题，处理器内部能够集成的晶体管数目有限，而三级 Cache 又比较大，从而占用的晶体管数量较多。以英特尔最新的 Haswell i7-5960X 为例，一级 Cache 有 32K，二级有 512K，但是三级却有 20M，在早期计算机系统中集成如此大的 SRAM 实在是很难做到。不过随着 90nm、45nm、32nm 以及 22nm 工艺的推出，处理器内部能够容纳更多的晶体管，所以三级 Cache 也慢慢集成到处理器内部了。

图 2-4 是一个简单的 Cache 系统逻辑示意图。

一级 Cache，一般分为数据 Cache 和指令 Cache，数据 Cache 用来存储数据，而指令 Cache 用于存放指令。这种 Cache 速度最快，一般处理器只需要 3 ~ 5 个指令周期就能访问到数据，因此成本高，容量小，一般都只有几十 KB。在多核处理器内部，每个处理器核心都拥有仅属于自己的一级 Cache。

二级 Cache，和一级 Cache 分为数据 Cache 和指令 Cache 不同，数据和指令都无差别地存放在一起。速度相比一级 Cache 慢一些，处理器大约需要十几个处理器周期才能访问到数

据，容量也相对来说大一些，一般有几百 KB 到几 MB 不等。在多核处理器内部，每个处理器核心都拥有仅属于自己的二级 Cache。

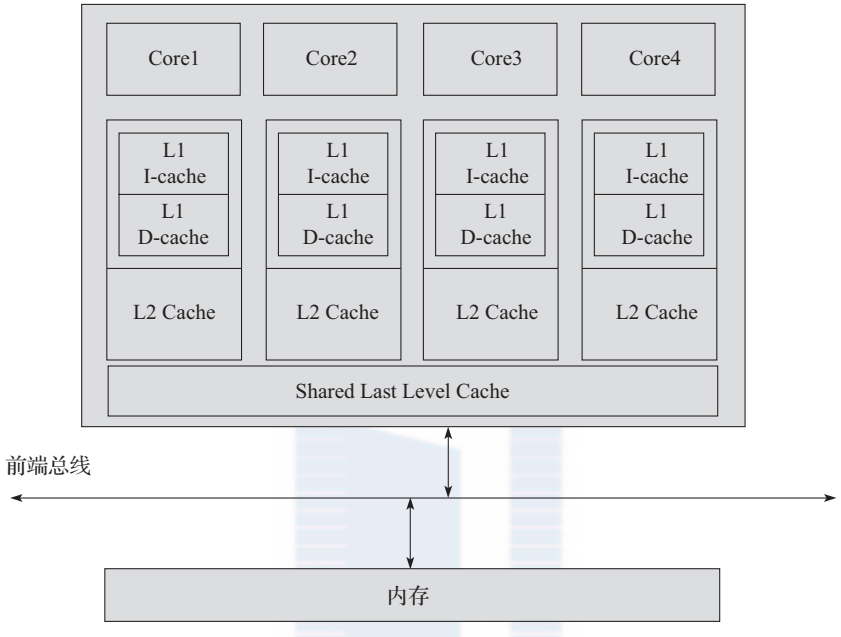


图 2-4 Cache 系统示意图

三级 Cache，速度更慢，处理器需要几十个处理器周期才能访问到数据，容量更大，一般都有几 MB 到几十个 MB。在多核处理器内部，三级 Cache 由所有的核心所共有。这样的共享方式，其实也带来一个问题，有的处理器可能会极大地占用三级 Cache，导致其他处理器只能占用极小的容量，从而导致 Cache 不命中，性能下降。因此，英特尔公司推出了 Intel® CAT 技术，确保有一个公平，或者说软件可配置的算法来控制每个核心可以用到的 Cache 大小。在此，本书就不再赘述。

对于各级 Cache 的访问时间，在英特尔的处理器上一直都保持着非常稳定，一级 Cache 访问是 4 个指令周期，二级 Cache 是 12 个指令周期，三级 Cache 则是 26 ~ 31 个指令周期。这里所谓的稳定，是指在不同频率、不同型号的英特尔处理器上，处理器访问这三级 Cache 所花费的指令周期数是相同的。请参照［Ref2-2］。

除了上述的 Cache 种类之外，还包含一些其他类型，接下来的章节会接着介绍。

2.2.2 TLB Cache

在早期计算机系统中，程序员都是直接访问物理地址进行编程，当程序出现错误时，整个系统都瘫痪掉；或者在多进程系统中，当一个进程出现问题，对属于另外一个进程的数据或者指令区域进行写操作，会导致另外一个进程崩溃。因此，随着计算机技术的进步，虚拟

地址和分段分页技术被提出来用来保护脆弱的软件系统。软件使用虚拟地址访问内存，而处理器负责虚拟地址到物理地址的映射工作。为了完成映射工作，处理器采用多级页表来进行多次查找最终找到真正的物理地址。当处理器发现页表中找不到真正对应的物理地址时，就会发出一个异常，挂起寻址错误的进程，但是其他进程仍然可以正常工作。

页表也存储在内存中，处理器虽然可以利用三级 Cache 系统来缓存页表内容，但是基于两点原因不能这样做。一种原因下面的段落会讲到，我们先讲另外一个原因。处理器每当进行寻址操作都要进行一次映射工作，这使得处理器访问页表的频率非常得高，有可能一秒钟需要访问几万次。因此，即使 Cache 的命中率能够达到 99% 以上，也就是说不命中率有 1%，那么不命中的概率每秒也有几百次，这会导致处理器在单位时间内访问内存（因为 Cache 没有命中，只能访问内存）的次数增多，降低了系统的性能。

因此，TLB（Translation Look-aside Buffer）Cache 应运而生，专门用于缓存内存中的页表项。TLB 一般都采用相连存储器或者按内容访问存储器（CAM，Content Addressable Memory）。相连存储器使用虚拟地址进行搜索，直接返回对应的物理地址，相对于内存中的多级页表需要多次访问才能得到最终的物理地址，TLB 查找无疑大大减少了处理器的开销，这也是上文提到的第二个原因。如果需要的地址在 TLB Cache 中，相连存储器迅速返回结果，然后处理器用该物理地址访问内存，这样的查找操作也称为 TLB 命中；如果需要的地址不在 TLB Cache 中，也就是不命中，处理器就需要到内存中访问多级页表，才能最终得到物理地址。

2.3　Cache 地址映射和变换

Cache 的容量一般都很小，即使是最大的三级 Cache（L3）也只有 20MB ~ 30MB。而当今内存的容量都是以 GB 作为单位，在一些服务器平台上，则都是以 TB（1TB=1024GB）作为单位。在这种情况下，如何把内存中的内容存放到 Cache 中去呢？这就需要一个映射算法和一个分块机制。

分块机制就是说，Cache 和内存以块为单位进行数据交换，块的大小通常以在内存的一个存储周期中能够访问到的数据长度为限。当今主流块的大小都是 64 字节，因此一个 Cache line 就是指 64 个字节大小的数据块。

而映射算法是指把内存地址空间映射到 Cache 地址空间。具体来说，就是把存放在内存中的内容按照某种规则装入到 Cache 中，并建立内存地址与 Cache 地址之间的对应关系。当内容已经装入到 Cache 之后，在实际运行过程中，当处理器需要访问这个数据块内容时，则需要把内存地址转换成 Cache 地址，从而在 Cache 中找到该数据块，最终返回给处理器。

根据 Cache 和内存之间的映射关系的不同，Cache 可以分为三类：第一类是全关联型 Cache（full associative cache），第二类是直接关联型 Cache（direct mapped cache），第三类是组关联型 Cache（N-ways associative cache）。

2.3.1 全关联型 Cache

全关联型 Cache 是指主存中的任何一块内存都可以映射到 Cache 中的任意一块位置上。在 Cache 中，需要建立一个目录表，目录表的每个表项都有三部分组成：内存地址、Cache 块号和一个有效位。当处理器需要访问某个内存地址时，首先通过该目录表查询是否该内容缓存在 Cache 中，具体过程如图 2-5 所示。

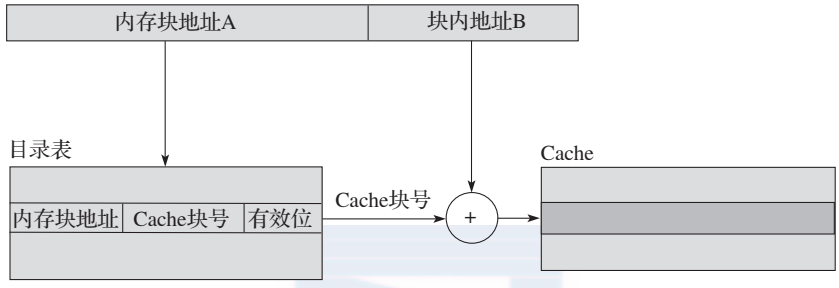


图 2-5 全关联 Cache 查找过程

首先，用内存的块地址 A 在 Cache 的目录表中进行查询，如果找到等值的内存块地址，检查有效位是否有效，只有有效的情况下，才能通过 Cache 块号在 Cache 中找到缓存的内存，并且加上块内地址 B，找到相应数据，这时则称为 Cache 命中，处理器拿到数据返回；否则称为不命中，处理器则需要在内存中读取相应的数据。

可以看出，使用全关联型 Cache，块的冲突最小（没有冲突），Cache 的利用率也高，但是需要一个访问速度很快的相联存储器。随着 Cache 容量的增加，其电路设计变得十分复杂，因此只有容量很小的 Cache 才会设计成全关联型的（如一些英特尔处理器中的 TLB Cache）。

2.3.2 直接关联型 Cache

直接关联型 Cache 是指主存中的一块内存只能映射到 Cache 的一个特定的块中。假设一个 Cache 中总共存在 N 个 Cache line，那么内存被分成 N 等分，其中每一等分对应一个 Cache line。举个简单的例子，假设 Cache 的大小是 2K，而一个 Cache line 的大小是 64B，那么就一共有 2K/64B=32 个 Cache line，那么对应我们的内存，第 1 块（地址 0 ~ 63），第 33 块（地址 64*32 ~ 64*33–1），以及第（N*32+1）块（地址 64*（N–1）~ 64*N–1）都被映射到 Cache 第一块中；同理，第 2 块，第 34 块，以及第（N*32+2）块都被映射到 Cache 第二块中；可以依次类推其他内存块。

直接关联型 Cache 的目录表只有两部分组成：区号和有效位。其查找过程如图 2-6 所示。首先，内存地址被分成三部分：区号 A、块号 B 和块内地址 C。根据区号 A 在目录表中找到完全相等的区号，并且在有效位有效的情况下，说明该数据在 Cache 中，然后通过内存

地址的块号 B 获得在 Cache 中的块地址，加上块内地址 C，最终找到数据。如果在目录表中找不到相等的区号，或者有效位无效的情况下，则说明该内容不在 Cache 中，需要到内存中读取。

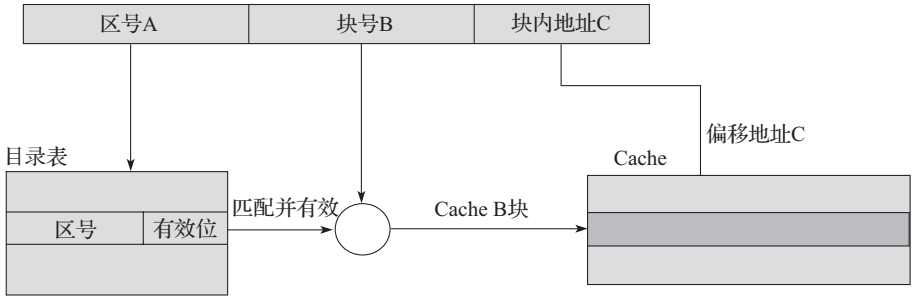


图 2-6　直接相联 Cache 查找过程

可以看出，直接关联是一种很“死”的映射方法，当映射到同一个 Cache 块的多个内存块同时需要缓存在 Cache 中时，只有一个内存块能够缓存，其他块需要被“淘汰”掉。因此，直接关联型命中率是最低的，但是其实现方式最为简单，匹配速度也最快。

2.3.3　组关联型 Cache

组关联型 Cache 是目前 Cache 中用的比较广泛的一种方式，是前两种 Cache 的折中形式。在这种方式下，内存被分为很多组，一个组的大小为多个 Cache line 的大小，一个组映射到对应的多个连续的 Cache line，也就是一个 Cache 组，并且该组内的任意一块可以映射到对应 Cache 组的任意一个。可以看出，在组外，其采用直接关联型 Cache 的映射方式，而在组内，则采用全关联型 Cache 的映射方式。

假设有一个 4 路组关联型 Cache，其大小为 1M，一个 Cache line 的大小为 64B，那么总共有 16K 个 Cache line，但是在 4 路组关联的情况下，我们并不是简简单单拥有 16K 个 Cache line，而是拥有了 4K 个组，每个组有 4 个 Cache line。一个内存单元可以缓存到它所对应的组中的任意一个 Cache line 中去。

图 2-7 以 4 路组关联型 Cache 为例介绍其在 Cache 中的查找过程。目录表由三部分组成，分别是“区号 + 块号”、Cache 块号和有效位。当收到一个内存地址时，该地址被分成四部分：区号 A、组号 B、块号 C 和块内地址 D。首先，根据组号 B 按地址查找到一组目录表项，在 4 路组关联中，则有四个表项，每个表项都有可能存放该内存块；然后，根据区号 A 和块号 C 在该组表项中进行关联查找（即并行查找，为了提高效率），如果匹配且有效位有效，则表明该数据块缓存在 Cache 中，得到 Cache 块号，加上块内地址 D，可以得到该内存地址在 Cache 中映射的地址，得到数据；如果没有找到匹配项或者有效位无效，则表示该内存块不在 Cache 中，需要处理器到内存中读取。

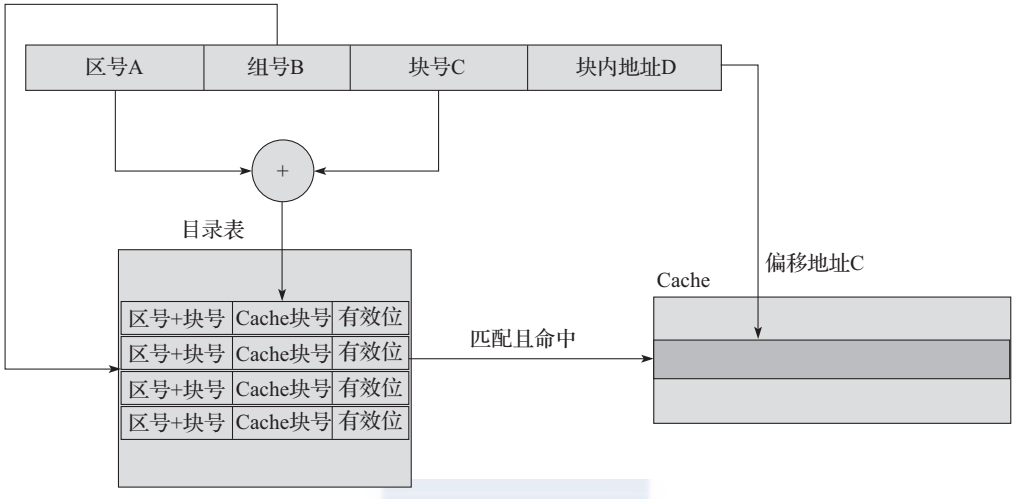


图 2-7 4 路组关联型 Cache 查找过程

实际上，直接关联型 Cache 和全关联型 Cache 只是组关联型 Cache 的特殊情况，当组内 Cache Line 数目为 1 时，即为直接关联型 Cache。而当组内 Cache Line 数目和 Cache 大小相等时，即整个 Cache 只有一个组，这成为全关联型 Cache。

2.4 Cache 的写策略

内存的数据被加载到 Cache 后，在某个时刻其要被写回内存，对于这个时刻的选取，有如下几个不同的策略。

直写（write-through）：所谓直写，就是指在处理器对 Cache 写入的同时，将数据写入到内存中。这种策略保证了在任何时刻，内存的数据和 Cache 中的数据都是同步的，这种方式简单、可靠。但由于处理器每次对 Cache 更新时都要对内存进行写操作，因此总线工作繁忙，内存的带宽被大大占用，因此运行速度会受到影响。假设一段程序在频繁地修改一个局部变量，尽管这个局部变量的生命周期很短，而且其他进程 / 线程也用不到它，CPU 依然会频繁地在 Cache 和内存之间交换数据，造成不必要的带宽损失。

回写（write-back）：回写相对于直写而言是一种高效的方法。直写不仅浪费时间，而且有时是不必要的，比如上文提到的局部变量的例子。回写系统通过将 Cache line 的标志位字段添加一个 Dirty 标志位，当处理器在改写了某个 Cache line 后，并不是马上把其写回内存，而是将该 Cache line 的 Dirty 标志设置为 1。当处理器再次修改该 Cache line 并且写回到 Cache 中，查表发现该 Dirty 位已经为 1，则先将 Cache line 内容写回到内存中相应的位置，再将新数据写到 Cache 中。其实，回写策略在多核系统中会引起 Cache 一致性的问题。设想有两个处理器核心都需要对某个内存块进行读写，其中一个核心已经修改了该数据块，并且写回到 Cache 中，设置了 Dirty 位；这时另外一个核心也完成了该内存块的修改，并且准备

写入到 Cache 中，这时才发现该 Cache line 是“脏”的，在这种情况下，Cache 如何处理呢？之后的章节我们会继续这个话题。

除了上述这两种写策略，还有 WC（write-combining）和 UC（uncacheable）。这两种策略都是针对特殊的地址空间来使用的。

write-combining 策略是针对于具体设备内存（如显卡的 RAM）的一种优化处理策略。对于这些设备来说，数据从 Cache 到内存转移的开销比直接访问相应的内存的开销还要高得多，所以应该尽量避免过多的数据转移。试想，如果一个 Cache line 里的字被改写了，处理器将其写回内存，紧接着又一个字被改写了，处理器又将该 Cache line 写回内存，这样就显得低效，符合这种情况的一个例子就是显示屏上水平相连的像素点数据。write-combining 策略的引入就是为了解决这种问题，顾名思义，这种策略就是当一个 Cache line 里的数据一个字一个字地都被改写完了之后，才将该 Cache line 写回到内存中。

uncacheable 内存是一部分特殊的内存，比如 PCI 设备的 I/O 空间通过 MMIO 方式被映射成内存来访问。这种内存是不能缓存在 Cache 中的，因为设备驱动在修改这种内存时，总是期望这种改变能够尽快通过总线写回到设备内部，从而驱动设备做出相应的动作。如果放在 Cache 中，硬件就无法收到指令。

2.5　Cache 预取

以上章节讲到了多种和 Cache 相关的技术，但是事实上，Cache 对于绝大多数程序员来说都是透明不可见的。程序员在编写程序时不需要关心是否有 Cache 的存在，有几级 Cache，每级 Cache 的大小是多少；不需要关心 Cache 采取何种策略将指令和数据从内存中加载到 Cache 中；也不需要关心 Cache 何时将处理完毕的数据写回到内存中。这一切，都是硬件自动完成的。但是，硬件也不是完全智能的，能够完美无缺地处理各种各样的情况，保证程序能够以最优的效率执行。因此，一些体系架构引入了能够对 Cache 进行预取的指令，从而使一些对程序执行效率有很高要求的程序员能够一定程度上控制 Cache，加快程序的执行。

接下来，将简单介绍一下硬件预取的原理，通过英特尔 NetBurst 架构具体介绍其预取的原则，最后介绍软件可以使用的 Cache 预取指令。

2.5.1　Cache 的预取原理

Cache 之所以能够提高系统性能，主要是程序执行存在局部性现象，即时间局部性和空间局部性。

1）时间局部性：是指程序即将用到的指令 / 数据可能就是目前正在使用的指令 / 数据。因此，当前用到的指令 / 数据在使用完毕之后可以暂时存放在 Cache 中，可以在将来的时候再被处理器用到。一个简单的例子就是一个循环语句的指令，当循环终止的条件满足之前，处理器需要反复执行循环语句中的指令。

2）空间局部性：是指程序即将用到的指令 / 数据可能与目前正在使用的指令 / 数据在空间上相邻或者相近。因此，在处理器处理当前指令 / 数据时，可以从内存中把相邻区域的指令 / 数据读取到 Cache 中，这样，当处理器需要处理相邻内存区域的指令 / 数据时，可以直接从 Cache 中读取，节省访问内存的时间。一个简单的例子就是一个需要顺序处理的数组。

所谓的 Cache 预取，也就是预测数据并取入到 Cache 中，是根据空间局部性和时间局部性，以及当前执行状态、历史执行过程、软件提示等信息，然后以一定的合理方法，在数据 / 指令被使用前取入 Cache。这样，当数据 / 指令需要被使用时，就能快速从 Cache 中加载到处理器内部进行运算和执行。

以上介绍的只是基本的预取原理，在不同体系架构，甚至不同处理器上，具体采取的预取方法都可能是不同的。以下以英特尔 NetBurst 架构的处理器为例介绍其预取的原则。详细内容请参见［ Ref2-1 ］。

2.5.2 NetBurst 架构处理器上的预取

在 NetBurst 架构上，每一级 Cache 都有相应的硬件预取单元，根据相应原则来预取数据 / 指令。由于篇幅原因，仅以一级数据 Cache 进行介绍。

1. 一级数据 Cache 的预取单元

NetBurst 架构的处理器上有两个硬件预取单元，用来加快程序，这样可以更快速地将所需要的数据送到一级数据 Cache 中。

1）数据 Cache 预取单元：也叫基于流的预取单元（Streaming prefetcher）。当程序以地址递增的方式访问数据时，该单元会被激活，自动预取下一个 Cache 行的数据。

2）基于指令寄存器（Instruction Pointer，IP）的预取单元：该单元会监测指令寄存器的读取（Load）指令，当该单元发现读取数据块的大小总是相对固定的情况下，会自动预取下一块数据。假设当前读取地址是 0xA000，读取数据块大小为 256 个字节，那地址是 0xA100-0xA200 的数据就会自动被预取到一级数据 Cache 中。该预取单元能够追踪的最大数据块大小是 2K 字节。

不过需要指出的是，只有以下的条件全部满足的情况下，数据预取的机制才会被激活。

1）读取的数据是回写（Writeback）的内存类型。

2）预取的请求必须在一个 4K 物理页的内部。这是因为对于程序员来说，虽然指令和数据的虚拟地址都是连续的，但是分配的物理页很有可能是不连续的。而预取是根据物理地址进行判断的，因此跨界预取的指令和数据很有可能是属于其他进程的，或者没有被分配的物理页。

3）处理器的流水线作业中没有 fence 或者 lock 这样的指令。

4）当前读取（Load）指令没有出现很多 Cache 不命中。

5）前端总线不是很繁忙。

6）没有连续的存储（Store）指令。

在该硬件预取单元激活的情况下，也不一定能够提高程序的执行效率。这取决于程序是如何执行的。

当程序需要多次访问某种大的数据结构，并且访问的顺序是有规律的，硬件单元能够捕捉到这种规律，进而能够提前预取需要处理的数据，那么就能提高程序的执行效率；当访问的顺序没有规律，或者硬件不能捕捉这种规律，这种预取不但会降低程序的性能，而且会占用更多的带宽，浪费一级 Cache 有限的空间；甚至在某些极端情况下，程序本身就占用了很多一级数据 Cache 的空间，而预取单元为了预取它认为程序需要的数据，不适当地淘汰了程序本身存放在一级 Cache 的数据，从而导致程序的性能严重下降。

2. 硬件预取所遵循的原则

在 Netburst 架构的处理器中，硬件遵循以下原则来决定是否开启自动预取。

1）只有连续两次 Cache 不命中才能激活预取机制。并且，这两次不命中的内存地址的位置偏差不能超过 256 或者 512 字节（NetBurst 架构的不同处理器定义的阈值不一样），否则也不会激活预取。这样做的目的是因为预取也会有开销，会占用内部总线的带宽，当程序执行没有规律时，盲目预取只会带来更多的开销，并且并不一定能够提高程序执行的效率。

2）一个 4K 字节的页（Page）内，只定义一条流（Stream，可以是指令，也可以是数据）。因为处理器同时能够追踪的流是有限的。

3）能够同时、独立地追踪 8 条流。每条流必须在一个 4K 字节的页内。

4）对 4K 字节的边界之外不进行预取。也就是说，预取只会在一个物理页（4K 字节）内发生。这和一级数据 Cache 预取遵循相同的原则。

5）预取的数据存放在二级或者三级 Cache 中。

6）对于 UC（Strong Uncacheable）和 WC（Write Combining）内存类型不进行预取。

2.5.3　两个执行效率迥异的程序

虽然绝大多数 Cache 预取对程序员来说都是透明的，但是了解预取的基本原理还是很有必要的，这样可以帮助我们编写高效的程序。以下就是两个相似的程序片段，但是执行效率却相差极大。这两个程序片段都定义了一个二维数组 arr[1024][1024]，对数组中每个元素都进行赋值操作。在内循环内，程序 1 是依次对 a[i][0], a[i][1], a[i][2]… a[i][1023] 进行赋值；程序 2 是依次对 a[0][i], a[1][i], a[2][i] … a[1023] [i] 进行赋值。

```
程序 1：
for(int i = 0; i < 1024; i++) {
    for(int j = 0; j < 1024; j++) {
        arr[i][j] = num++;
    }
}

程序 2：
for(int i = 0; i < 1024; i++) {
```

```
        for(int j = 0; j < 1024; j++) {
            arr[j][i] = num++;
        }
    }
```

通过图 2-8 可以清晰地看到程序 1 和程序 2 的执行顺序。程序 1 是按照数组在内存中的保存方式顺序访问，而程序 2 则是跳跃式访问。对于程序 1，硬件预取单元能够自动预取接下来需要访问的数据到 Cache，节省访问内存的时间，从而提高程序 1 的执行效率；对于程序 2，硬件不能够识别数据访问的规律，因而不会预取，从而使程序 2 总是需要在内存中读取数据，降低了执行的效率。

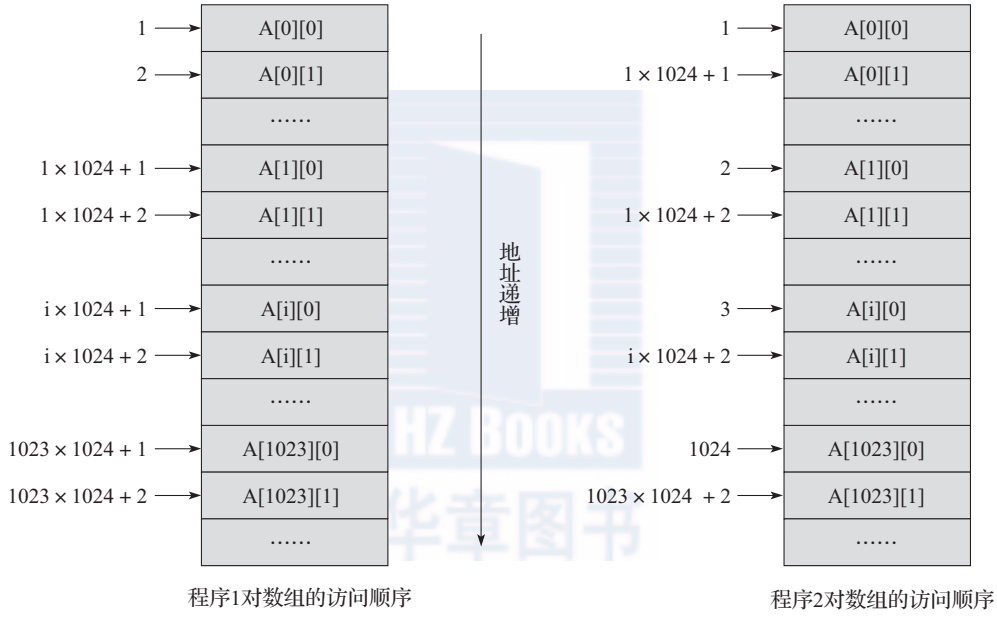


图 2-8 两组程序执行过程示意图

2.5.4 软件预取

从上面的介绍可以看出，硬件预取单元并不一定能够提高程序执行的效率，有些时候可能会极大地降低执行的效率。因此，一些体系架构的处理器增加了一些指令，使得软件开发者和编译器能够部分控制 Cache。能够影响 Cache 的指令很多，本书仅介绍预取相关的指令。

❑ 软件预取指令

预取指令使软件开发者在性能相关区域，把即将用到的数据从内存中加载到 Cache，这样当前数据处理完毕后，即将用到的数据已经在 Cache 中，大大减小了从内存直接读取的开销，也减少了处理器等待的时间，从而提高了性能。增加预取指令并不是让软件开发者需要时时考虑到 Cache 的存在，让软件自己来管理 Cache，而是在某些热点区域，或者性能相关

区域能够通过显示地加载数据到 Cache，提高程序执行的效率。不过，不正确地使用预取指令，造成 Cache 中负载过重或者无用数据的比例增加，反而还会造成程序性能下降，也有可能造成其他程序执行效率降低（比如某程序大量加载数据到三级 Cache，影响到其他程序）。因此，软件开发者需要仔细衡量利弊，充分进行测试，才能够正确地优化程序。需要指出的是，预取指令只对数据有效，对指令预取是无效的。表 2-1 给出了预取的指令列表。

表 2-1　预取指令列表

指　令	解　释
PREFETCH0	将数据存放在每一级 Cache。假设有三级 Cache，则 L1、L2、L3 Cache 都包含该数据的一个备份
PREFETCH1	将数据存放在除了 L1Cache 之外的每一级 Cache。假设有三级 Cache，则 L2、L3 Cache 都包含该数据的一个备份
PREFETCH2	将数据存放在除了 L1 和 L2 Cache 之外的每一级 Cache。假设有三级 Cache，则 L3 Cache 包含该数据的一个备份
PREFETCHNTA	和 PREFETCH0 功能类似，区别是数据是作为非临时（non-temporal）数据存放，在使用完一次之后，Cache 认为该数据是可以被淘汰出去的

预取指令是汇编指令，对于很多软件开发者来说，直接插入汇编指令不是很方便，一些程序库也提供了相应的软件版本。比如“mmintrin.h”提供了如下的函数原型：

```
void _mm_prefetch(char *p, int i);
```

p 是需要预取的内存地址，i 对应相应的预取指令，如表 2-2 所示。

表 2-2　软件库中的预取函数

i	对应预取指令
_MM_HINT_T0	PREFETCH0
_MM_HINT_T1	PREFETCH1
_MM_HINT_T2	PREFETCH2
_MM_HINT_NTA	PREFETCHNTA

接下来，我们将以 DPDK 中 PMD（Polling Mode Driver）驱动中的一个程序片段看看 DPDK 是如何利用预取指令的。

❑ DPDK 中的预取

在讨论之前，我们需要了解另外一个和性能相关的话题。DPDK 一个处理器核每秒钟大概能够处理 33M 个报文，大概每 30 纳秒需要处理一个报文，假设处理器的主频是 2.7GHz，那么大概每 80 个处理器时钟周期就需要处理一个报文。那么，处理报文需要做一些什么事情呢？以下是一个基本过程。

1）写接收描述符到内存，填充数据缓冲区指针，网卡收到报文后就会根据这个地址把报文内容填充进去。

2）从内存中读取接收描述符（当收到报文时，网卡会更新该结构）（内存读），从而确认是否收到报文。

3）从接收描述符确认收到报文时，从内存中读取控制结构体的指针（内存读），再从内存中读取控制结构体（内存读），把从接收描述符读取的信息填充到该控制结构体。

4）更新接收队列寄存器，表示软件接收到了新的报文。

5）内存中读取报文头部（内存读），决定转发端口。

6）从控制结构体把报文信息填入到发送队列发送描述符，更新发送队列寄存器。

7）从内存中读取发送描述符（内存读），检查是否有包被硬件传送出去。

8）如果有的话，从内存中读取相应控制结构体（内存读），释放数据缓冲区。

可以看出，处理一个报文的过程，需要 6 次读取内存（见上“内存读”）。而之前我们讨论过，处理器从一级 Cache 读取数据需要 3 ~ 5 个时钟周期，二级是十几个时钟周期，三级是几十个时钟周期，而内存则需要几百个时钟周期。从性能数据来说，每 80 个时钟周期就要处理一个报文。

因此，DPDK 必须保证所有需要读取的数据都在 Cache 中，否则一旦出现 Cache 不命中，性能将会严重下降。为了保证这点，DPDK 采用了多种技术来进行优化，预取只是其中的一种。

而从上面的介绍可以看出，控制结构体和数据缓冲区的读取都没有遵循硬件预取的原则，因此 DPDK 必须用一些预取指令来提前加载相应数据。以下就是部分接收报文的代码。

```
while (nb_rx < nb_pkts) {
    rxdp = &rx_ring[rx_id]; //读取接收描述符
    staterr = rxdp->wb.upper.status_error;
    //检查是否有报文收到
    if (!(staterr & rte_cpu_to_le_32(IXGBE_RXDADV_STAT_DD)))
        break;
    rxd = *rxdp;
    //分配数据缓冲区
    nmb = rte_rxmbuf_alloc(rxq->mb_pool);
    nb_hold++;
    //读取控制结构体
    rxe = &sw_ring[rx_id];
    ……
    rx_id++;
    if (rx_id == rxq->nb_rx_desc)
        rx_id = 0;
    //预取下一个控制结构体 mbuf
    rte_ixgbe_prefetch(sw_ring[rx_id].mbuf);
    //预取接收描述符和控制结构体指针
    if ((rx_id & 0x3) == 0) {
        rte_ixgbe_prefetch(&rx_ring[rx_id]);
        rte_ixgbe_prefetch(&sw_ring[rx_id]);
    }
    ……
```

```
    // 预取报文
    rte_packet_prefetch((char *)rxm->buf_addr + rxm->data_off);
    // 把接收描述符读取的信息存储在控制结构体 mbuf 中
    rxm->nb_segs = 1;
    rxm->next = NULL;
    rxm->pkt_len = pkt_len;
    rxm->data_len = pkt_len;
    rxm->port = rxq->port_id;
    ......
    rx_pkts[nb_rx++] = rxm;
}
```

2.6　Cache 一致性

我们知道，Cache 是按照 Cache Line 作为基本单位来组织内容的，其大小是 32（较早的 ARM、1990 年 ~ 2000 年早期的 x86 和 PowerPC）、64（较新的 ARM 和 x86）或 128（较新的 Power ISA 机器）字节。当我们定义了一个数据结构或者分配了一段数据缓冲区之后，在内存中就有一个地址和其相对应，然后程序就可以对它进行读写。对于读，首先是从内存加载到 Cache，最后送到处理器内部的寄存器；对于写，则是从寄存器送到 Cache，最后通过内部总线写回到内存。这两个过程其实引出了两个问题：

1）该数据结构或者数据缓冲区的起始地址是 Cache Line 对齐的吗？如果不是，即使该数据区域的大小小于 Cache Line，那么也需要占用两个 Cache entry；并且，假设第一个 Cache Line 前半部属于另外一个数据结构并且另外一个处理器核正在处理它，那么当两个核都修改了该 Cache Line 从而写回各自的一级 Cache，准备送到内存时，如何同步数据？毕竟每个核都只修改了该 Cache Line 的一部分。

2）假设该数据结构或者数据缓冲区的起始地址是 Cache Line 对齐的，但是有多个核同时对该段内存进行读写，当同时对内存进行写回操作时，如何解决冲突？

接下来，我们先回答第一个问题，然后再回答第二个问题。

2.6.1　Cache Line 对齐

对于第一个问题，其实有多种方法来解决。比如，用解决第二个问题的方法去解决它，从本质来讲，第一个问题和第二个问题都是因为多个核同时操作一个 Cache Line 进行写操作造成的。

另外一个简单的方法就是定义该数据结构或者数据缓冲区时就申明对齐，DPDK 对很多结构体定义的时候就是如此操作的。见下例：

```
struct rte_ring_debug_stats {
uint64_t enq_success_bulk;
uint64_t enq_success_objs;
uint64_t enq_quota_bulk;
```

```
uint64_t enq_quota_objs;
uint64_t enq_fail_bulk;
uint64_t enq_fail_objs;
uint64_t deq_success_bulk;
uint64_t deq_success_objs;
uint64_t deq_fail_bulk;
uint64_t deq_fail_objs;
} __rte_cache_aligned;
```

__rte_cache_aligned 的定义如下所示：

```
#define RTE_CACHE_LINE_SIZE 64
#define __rte_cache_aligned
__attribute__((__aligned__(RTE_CACHE_LINE_SIZE)))
```

其实现在编译器很多时候也比较智能，会在编译的时候尽量做到 Cache Line 对齐。

2.6.2 Cache 一致性问题的由来

上文提到的第二个问题，即多个处理器对某个内存块同时读写，会引起冲突的问题，这也被称为 Cache 一致性问题。

Cache 一致性问题出现的原因是在一个多处理器系统中，每个处理器核心都有独占的 Cache 系统（比如我们之前提到的一级 Cache 和二级 Cache），而多个处理器核心都能够独立地执行计算机指令，从而有可能同时对某个内存块进行读写操作，并且由于我们之前提到的回写和直写的 Cache 策略，导致一个内存块同时可能有多个备份，有的已经写回到内存中，有的在不同的处理器核心的一级、二级 Cache 中。由于 Cache 缓存的原因，我们不知道数据写入的时序性，因而也不知道哪个备份是最新的。还有另外一个一种可能，假设有两个线程 A 和 B 共享一个变量，当线程 A 处理完一个数据之后，通过这个变量通知线程 B，然后线程 B 对这个数据接着进行处理，如果两个线程运行在不同的处理器核心上，那么运行线程 B 的处理器就会不停地检查这个变量，而这个变量存储在本地的 Cache 中，因此就会发现这个值总也不会发生变化。

其实，关于一致性问题的阐述，我们附加了很多限制条件，比如多核，独占 Cache，Cache 写策略。如果当中有一个或者多个条件不成立时可能就不会引发一致性的问题了。

1）假设只是单核处理器，那么只有一个处理器会对内存进行读写，Cache 也是只有一份，因而不会出现一致性的问题。

2）假设是多核处理器系统，但是 Cache 是所有处理器共享的，那么当一个处理器对内存进行修改并且缓存在 Cache 中时，其他处理器都能看到这个变化，因而也不会产生一致性的问题。

3）假设是多核处理器系统，每个核心也有独占的 Cache，但是 Cache 只会采用直写，那么当一个处理器对内存进行修改之后，Cache 会马上将数据写入到内存中，也不会有问题吗？考虑之前我们介绍的一个例子，线程 A 把结果写回到内存中，但是线程 B 只会从独占的

Cache 中读取这个变量（因为没人通知它内存的数据产生了变化），因此在这种条件下还是会有 Cache 一致性的问题。

因而，Cache 一致性问题的根源是因为存在多个处理器独占的 Cache，而不是多个处理器。如果多个处理器共享 Cache，也就是说只有一级 Cache，所有处理器都共享它，在每个指令周期内，只有一个处理器核心能够通过这个 Cache 做内存读写操作，那么就不会存在 Cache 一致性问题。

讲到这里，似乎我们找到了一劳永逸解决 Cache 一致性问题的办法，只要所有的处理器共享 Cache，那么就不会有任何问题。但是，这种解决办法的问题就是太慢了。首先，既然是共享的 Cache，势必容量不能小，那么就是说访问速度相比之前提到的一级、二级 Cache，速度肯定几倍或者十倍以上；其次，每个处理器每个时钟周期内只有一个处理器才能访问 Cache，那么处理器把时间都花在排队上了，这样效率太低了。

因而，我们还是需要针对出现的 Cache 一致性问题，找出一个解决办法。

2.6.3　一致性协议

解决 Cache 一致性问题的机制有两种：基于目录的协议（Directory-based protocol）和总线窥探协议（Bus snooping protocol）。其实还有另外一个 Snarfing 协议，在此不作讨论。

基于目录协议的系统中，需要缓存在 Cache 的内存块被统一存储在一个目录表中，目录表统一管理所有的数据，协调一致性问题。该目录表类似于一个仲裁者，当处理器需要把一个数据从内存中加载到自己独占的 Cache 中时，需要向目录表提出申请；当一个内存块被某个处理器改变之后，目录表负责改变其状态，更新其他处理器的 Cache 中的备份，或者使其他处理器的 Cache 的备份无效。

总线窥探协议是在 1983 年被首先提出来，这个协议提出了一个窥探（snooping）的动作，即对于被处理器独占的 Cache 中的缓存的内容，该处理器负责监听总线，如果该内容被本处理器改变，则需要通过总线广播；反之，如果该内容状态被其他处理器改变，本处理器的 Cache 从总线收到了通知，则需要相应改变本地备份的状态。

可以看出，这两类协议的主要区别在于基于目录的协议采用全局统一管理不同 Cache 的状态，而总线窥探协议则使用类似于分布式的系统，每个处理器负责管理自己的 Cache 的状态，通过共享的总线，同步不同 Cache 备份的状态。

通过之前的描述可以发现，在上面两种协议中，每个 Cache Block 都必须有自己的一个状态字段。而维护 Cache 一致性问题的关键在于维护每个 Cache Block 的状态域。Cache 控制器通常使用一个状态机来维护这些状态域。

基于目录的协议的延迟性较大，但是在拥有很多个处理器的系统中，它有更好的可扩展性。而总线窥探协议适用于具有广播能力的总线结构，允许每个处理器能够监听其他处理器对内存的访问，适合小规模的多核系统。

接下来，我们将主要介绍总线窥探协议。最经典的总线窥探协议 Write-Once 由 C.V.

Ravishankar 和 James R. Goodman 于 1983 年提出，继而被 x86、ARM 和 Power 等架构广泛采用，衍生出著名的 MESI 协议，或者称为 Illinois Protocol。之所以有这个名字，是因为该协议是由伊利诺伊州立大学研发出来的。

2.6.4 MESI 协议

MESI 协议是 Cache line 四种状态的首字母的缩写，分别是修改（Modified）态、独占（Exclusive）态、共享（Shared）态和失效（Invalid）态。Cache 中缓存的每个 Cache Line 都必须是这四种状态中的一种。详见［Ref2-2］。

- 修改态，如果该 Cache Line 在多个 Cache 中都有备份，那么只有一个备份能处于这种状态，并且“dirty”标志位被置上。拥有修改态 Cache Line 的 Cache 需要在某个合适的时候把该 Cache Line 写回到内存中。但是在写回之前，任何处理器对该 Cache Line 在内存中相对应的内存块都不能进行读操作。Cache Line 被写回到内存中之后，其状态就由修改态变为共享态。
- 独占态，和修改状态一样，如果该 Cache Line 在多个 Cache 中都有备份，那么只有一个备份能处于这种状态，但是“dirty”标志位没有置上，因为它是和主内存内容保持一致的一份拷贝。如果产生一个读请求，它就可以在任何时候变成共享态。相应地，如果产生了一个写请求，它就可以在任何时候变成修改态。
- 共享态，意味着该 Cache Line 可能在多个 Cache 中都有备份，并且是相同的状态，它是和内存内容保持一致的一份拷贝，而且可以在任何时候都变成其他三种状态。
- 失效态，该 Cache Line 要么已经不在 Cache 中，要么它的内容已经过时。一旦某个 Cache Line 被标记为失效，那它就被当作从来没被加载到 Cache 中。

对于某个内存块，当其在两个（或多个）Cache 中都保留了一个备份时，只有部分状态是允许的。如表 2-3 所示，横轴和竖轴分别表示了两个 Cache 中某个 Cache Line 的状态，两个 Cache Line 都映射到相同的内存块。如果一个 Cache Line 设置成 M 态或者 E 态，那么另外一个 Cache Line 只能设置成 I 态；如果一个 Cache Line 设置成 S 态，那么另外一个 Cache Line 可以设置成 S 态或者 I 态；如果一个 Cache Line 设置成 I 态，那么另外一个 Cache Line 可以设置成任何状态。

表 2-3 MESI 中两个 Cache 备份的状态矩阵

	M	E	S	I
M	×	×	×	√
E	×	×	×	√
S	×	×	√	√
I	√	√	√	√

那么，究竟怎样的操作才会引起 Cache Line 的状态迁移，从而保持 Cache 的一致性呢？

以下所示表 2-4 是根据不同读写操作触发的状态迁移表。

表 2-4　MESI 状态迁移表

当前状态	触发事件	解　　释	迁移状态
修改态（M）	总线读	侦测到总线上有其他处理器在请求读该行，刷新该行至内存，以便其他处理器能用到最新的数据，并且状态更新为 S 态	S
	总线写	侦测到总线上有其他处理器请求“意图”写该行，即请求独占态，刷新该行至内存，并且设置本地副本为 I 态	I
	处理器读	本地处理器对该行进行读操作，不改变状态	M
	处理器写	本地处理器对该行进行写操作，不改变状态	M
独占态（E）	总线读	侦测到总线上有其他处理器请求读该行，因为本地处理器还没有对该行进行写操作，因此缓存内容与内存中内容一致，仅仅改变成 S 状态	S
	总线写	侦测到总线上有其他处理器请求“意图”写该行，即另外有处理器请求独占该行，并且有写的意图，因此设置成 I 态	I
	处理器读	本地处理器对该行进行读操作，不改变状态	E
	处理器写	本地处理器对该行进行写操作，进入到 M 态	M
共享态（S）	总线读	侦测到总线上有其他处理器请求读该行，不改变状态	S
	总线写	侦测到总线上有其他处理器请求“意图”写该行，进入 I 态	I
	处理器读	本地处理器对该行进行读操作，不改变状态	S
	处理器写	产生一个请求“意图”写该行的信号到总线，进入到 M 态	M
无效态（I）	总线读	侦测到总线上有其他处理器请求读该行，不改变状态	I
	总线写	侦测到总线上有其他处理器请求“意图”写该行，不改变状态	I
	处理器读	Cache 不命中，产生一个读请求，送到总线上，内存数据到达 Cache 后，进入 S 态	S
	处理器写	Cache 不命中，产生一个“意图”写该行的信号到总线，然后进入 M 态	M

2.6.5　DPDK 如何保证 Cache 一致性

从上面的介绍我们知道，Cache 一致性这个问题的最根本原因是处理器内部不止一个核，当两个或多个核访问内存中同一个 Cache 行的内容时，就会因为多个 Cache 同时缓存了该内容引起同步的问题。

DPDK 与生俱来就是为了网络平台的高性能和高吞吐，并且总是需要部署在多核的环境下。因此，DPDK 必须提出好的解决方案，避免由于不必要的 Cache 一致性开销而造成额外的性能损失。

其实，DPDK 的解决方案很简单，首先就是避免多个核访问同一个内存地址或者数据结构。这样，每个核尽量都避免与其他核共享数据，从而减少因为错误的数据共享（cache line false sharing）导致的 Cache 一致性的开销。

以下是两个 DPDK 为了避免 Cache 一致性的例子。

例子 1：数据结构定义。DPDK 的应用程序很多情况下都需要多个核同时来处理事务，因而，对于某些数据结构，我们给每个核都单独定义一份，这样每个核都只访问属于自己核的备份。如下例所示：

```
struct lcore_conf {
uint16_t n_rx_queue;
struct lcore_rx_queue rx_queue_list[MAX_RX_QUEUE_PER_LCORE];
uint16_t tx_queue_id[RTE_MAX_ETHPORTS];
struct mbuf_table tx_mbufs[RTE_MAX_ETHPORTS];
lookup_struct_t * ipv4_lookup_struct;
lookup_struct_t * ipv6_lookup_struct;
} __rte_cache_aligned;    //Cache 行对齐
struct lcore_conf lcore[RTE_MAX_LCORE] __rte_cache_aligned;
```

以上的数据结构“struct lcore_conf”总是以 Cache 行对齐，这样就不会出现该数据结构横跨两个 Cache 行的问题。而定义的数组“lcore[RTE_MAX_LCORE]”中 RTE_MAX_LCORE 指一个系统中最大核的数量。DPDK 中对每个核都进行编号，这样核 n 就只需要访问 lcore[n]，核 m 只需要访问 lcore[m]，这样就避免了多个核访问同一个结构体。

例子 2：对网络端口的访问。在网络平台中，少不了访问网络设备，比如网卡。多核情况下，有可能多个核访问同一个网卡的接收队列 / 发送队列，也就是在内存中的一段内存结构。这样，也会引起 Cache 一致性的问题。那么 DPDK 是如何解决这个问题的呢？

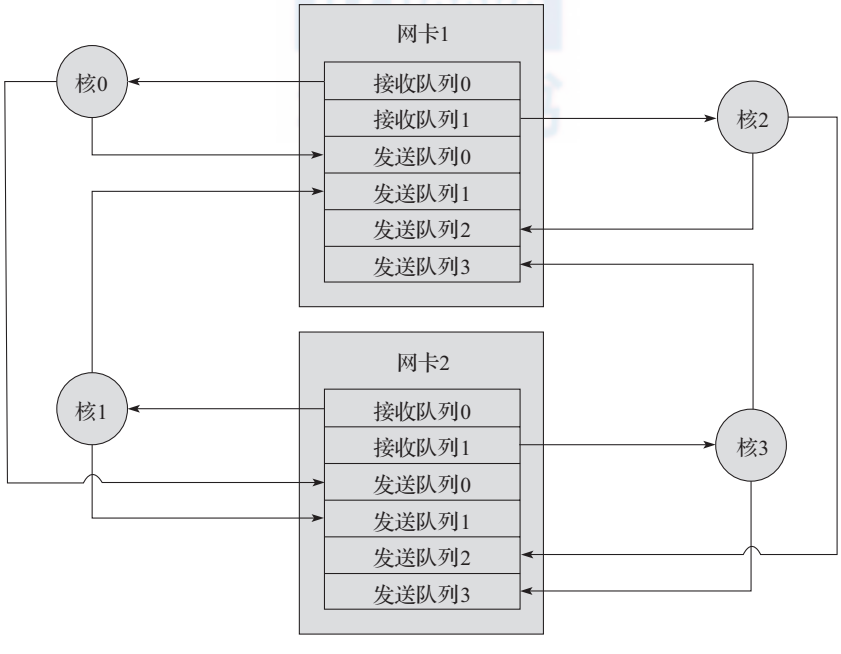


图 2-9　多核多队列收发示意图

需要指出的是，网卡设备一般都具有多队列的能力，也就是说，一个网卡有多个接收队列和多个访问队列，其他章节会很详细讲到，本节不再赘述。

DPDK 中，如果有多个核可能需要同时访问同一个网卡，那么 DPDK 就会为每个核都准备一个单独的接收队列 / 发送队列。这样，就避免了竞争，也避免了 Cache 一致性问题。

图 2-9 是四个核可能同时访问两个网络端口的图示。其中，网卡 1 和网卡 2 都有两个接收队列和四个发送队列；核 0 到核 3 每个都有自己的一个接收队列和一个发送队列。核 0 从网卡 1 的接收队列 0 接收数据，可以发送到网卡 1 的发送队列 0 或者网卡 2 的发送队列 0；同理，核 3 从网卡 2 的接收队列 1 接收数据，可以发送到网卡 1 的发送队列 3 或者网卡 2 的发送队列 3。

2.7　TLB 和大页

在之前的章节我们提到了 TLB，TLB 和 Cache 本质上是一样的，都是一种高速的 SRAM，存放了内存中内容的一份快照或者备份，以便处理器能够快速地访问，减少等待的时间。有所不同的是，Cache 存放的是内存中的数据或者代码，或者说是任何内容，而 TLB 存放的是页表项。

提到页表项，有必要简短介绍一下处理器的发展历史。最初的程序员直接对物理地址编程，自己去管理内存，这样不仅对程序员要求高，编程效率低，而且一旦程序出现问题也不方便进行调试。特别还出现了恶意程序，这对计算机系统危害实在太大，因而后来不同的体系架构推出了虚拟地址和分页的概念。

分页是指把物理内存分成固定大小的块，按照页来进行分配和释放。一般常规页大小为 4K（2^{12}）个字节，之后又因为一些需要，出现了大页，比如 2M（2^{20}）个字节和 1G（2^{30}）个字节的大小，我们后面会讲到为什么使用大页。

虚拟地址是指程序员使用虚拟地址进行编程，不用关心物理内存的大小，即使自己的程序出现了问题也不会影响其他程序的运行和系统的稳定。而处理器在寄存器收到虚拟地址之后，根据页表负责把虚拟地址转换成真正的物理地址。

接下来，我们以一个例子来简单介绍地址转换过程。

2.7.1　逻辑地址到物理地址的转换

图 2-10 是 x86 在 32 位处理器上进行一次逻辑地址（或线性地址）转换物理地址的示意图。

处理器把一个 32 位的逻辑地址分成 3 段，每段都对应一个偏移地址。查表的顺序如下：

1）根据位 bit[31:22] 加上寄存器 CR3 存放的页目录表的基址，获得页目录表中对应表项的物理地址，读内存，从内存中获得该表项内容，从而获得下一级页表的基址。

2）根据位 bit[21:12] 页表加上上一步获得的页表基址，获得页表中对应表项的物理地址，读内存，从内存中获得该表项内容，从而获得内容页的基址。

3）根据为 bit[11:0] 加上上一步获得的内容页的基址得到准确的物理地址，读内容获得真正的内容。

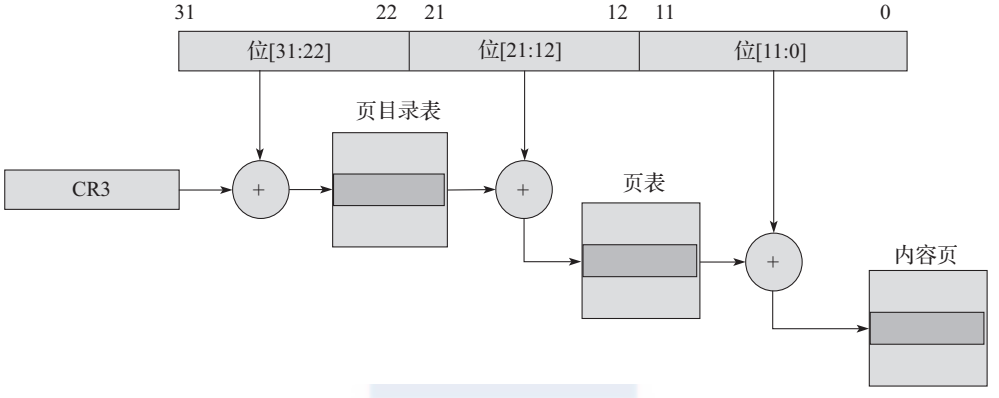


图 2-10 页表查找过程

从上面的描述可以看出，为了完成逻辑地址到物理地址的转换，需要三次内存访问，这实在是太浪费时间了。有的读者可能会问，为什么要分成三段进行查找呢？如果改成两段的话，那不是可以减少一级页表，也可以减少一次内存访问，从而可以提高访问速度。为了回答这个问题，我们举一个例子来看。

假设有一个程序，代码段加数据段可以放在两个 4KB 的页内。如果使用三段的方式，那么需要一个页存放页目录表（里面只有一个目录项有效），一个页存放页表（里面有两个目录项有效），因此需要总共两个页 8192 个字节就可以了；如果使用两段的方式，那使用 bit[31:12] 共 20 位来查页表，根据其范围，那么需要有 2^{20} 个表项，因此需要 4MB 来建立页表，也就是 1024 个物理页，而其中只有两个表项是有效的，这实在是太浪费了。特别是当程序变多时，系统内存会不堪使用。这样的改进代价实在太大。

通过之前的介绍我们知道有 Cache 的存在，我们也可以把页表缓存在 Cache 中，但是由于页表项的快速访问性（每次程序内存寻址都需要访问页表）和 Cache 的“淘汰”机制，有必要提供专门的 Cache 来保存，也就是 TLB。

2.7.2 TLB

相比之前提到的三段查表方式，引入 TLB 之后，查找过程发生了一些变化。TLB 中保存着逻辑地址前 20 位 [31:12] 和页框号的对应关系，如果匹配到逻辑地址就可以迅速找到页框号（页框号可以理解为页表项），通过页框号与逻辑地址后 12 位的偏移组合得到最终的物理地址。

如果没在 TLB 中匹配到逻辑地址，就出现 TLB 不命中，从而像我们刚才讨论的那样，进行常规的查找过程。如果 TLB 足够大，那么这个转换过程就会变得很快速。但是事实是，TLB 是非常小的，一般都是几十项到几百项不等，并且为了提高命中率，很多处理器还采用全相连方式。另外，为了减少内存访问的次数，很多都采用回写的策略。

在有些处理器架构中，为了提高效率，还将 TLB 进行分组，以 x86 架构为例，一般都分成以下四组 TLB：

第一组：缓存一般页表（4KB 页面）的指令页表缓存（Instruction-TLB）。

第二组：缓存一般页表（4KB 页面）的数据页表缓存（Data-TLB）。

第三组：缓存大尺寸页表（2MB/4MB 页面）的指令页表缓存（Instruction-TLB）。

第四组：缓存大尺寸页表（2MB/4MB 页面）的数据页表缓存（Data-TLB）。

2.7.3　使用大页

从上面的逻辑地址到物理地址的转换我们知道，如果采用常规页（4KB）并且使 TLB 总能命中，那么至少需要在 TLB 表中存放两个表项，在这种情况下，只要寻址的内容都在该内容页内，那么两个表项就足够了。如果一个程序使用了 512 个内容页也就是 2MB 大小，那么需要 512 个页表表项才能保证不会出现 TLB 不命中的情况。通过上面的介绍，我们知道 TLB 大小是很有限的，随着程序的变大或者程序使用内存的增加，那么势必会增加 TLB 的使用项，最后导致 TLB 出现不命中的情况。那么，在这种情况下，大页的优势就显现出来了。如果采用 2MB 作为分页的基本单位，那么只需要一个表项就可以保证不出现 TLB 不命中的情况；对于消耗内存以 GB（2^{30}）为单位的大型程序，可以采用 1GB 为单位作为分页的基本单位，减少 TLB 不命中的情况。

2.7.4　如何激活大页

我们以 Linux 系统为例来说明如何激活大页的使用。

首先，Linux 操作系统采用了基于 hugetlbfs 的特殊文件系统来加入对 2MB 或者 1GB 的大页面支持。这种采用特殊文件系统形式支持大页面的方式，使得应用程序可以根据需要灵活地选择虚存页面大小，而不会被强制使用 2MB 大页面。

为了使用大页，必须在编译内核的时候激活 hugetlbfs。

在激活 hugetlbfs 之后，还必须在 Linux 启动之后保留一定数量的内存作为大页来使用。现在有两种方式来预留内存。

第一种是在 Linux 命令行指定，这样 Linux 启动之后内存就已经预留；第二种方式是在 Linux 启动之后，可以动态地预留内存作为大页使用。以下是 2MB 大页命令行的参数。

```
Huagepage=1024
```

对于其他大小的大页，比如 1GB，其大小必须显示地在命令行指定，并且命令行还可以指定默认的大页大小。比如，我们想预留 4GB 内存作为大页使用，大页的大小为 1GB，那么可以用以下的命令行：

```
default_hugepagesz=1G hugepagesz=1G hugepages=4
```

需要指出的是，系统能否支持大页，支持大页的大小为多少是由其使用的处理器决定的。以 Intel® 的处理器为例，如果处理器的功能列表有 PSE，那么它就支持 2MB 大小的大

页；如果处理器的功能列表有 PDPE1GB，那么就支持 1GB 大小的大页。当然，不同体系架构支持的大页的大小都不尽相同，比如 x86 处理器架构的 2MB 和 1GB 大页，而在 IBM Power 架构中，大页的大小则为 16MB 和 16GB。

在我们之后会讲到的 NUMA 系统中，因为存在本地内存的问题，系统会均分地预留大页。假设在有两个处理器的 NUMA 系统中，以上例预留 4GB 内存为例，在 NODE0 和 NODE1 上会各预留 2GB 内存。

在 Linux 启动之后，如果想预留大页，则可以使用以下的方法来预留内存。在非 NUMA 系统中，可以使用以下方法预留 2MB 大小的大页。

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

该命令预留 1024 个大小为 2MB 的大页，也就是预留了 2GB 内存。

如果是在 NUMA 系统中，假设有两个 NODE 的系统中，则可以用以下的命令：

```
echo 1024 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
echo 1024 > /sys/devices/system/node/node1/hugepages/hugepages-2048kB/nr_hugepages
```

该命令在 NODE0 和 NODE1 上各预留 1024 个大小为 2MB 的大页，总共预留了 4GB 大小。

而对于大小为 1GB 的大页，则必须在 Linux 命令行的时候就指定，不能动态预留。

在大页预留之后，接下来则涉及使用的问题。我们以 DPDK 为例来说明如何使用大页。

DPDK 也是使用 HUGETLBFS 来使用大页。首先，它需要把大页 mount 到某个路径，比如 /mnt/huge，以下是命令：

```
mkdir /mnt/huge
mount -t hugetlbfs nodev /mnt/huge
```

需要指出的是，在 mount 之前，要确保之前已经成功预留内存，否则之上命令会失败。该命令只是临时的 mount 了文件系统，如果想每次开机时省略该步骤，可以修改 /etc/fstab 文件，加上一行：

```
nodev /mnt/huge hugetlbfs defaults 0 0
```

对于 1GB 大小的大页，则必须用如下的命令：

```
nodev /mnt/huge_1GB hugetlbfs pagesize=1GB 0 0
```

接下来，在 DPDK 运行的时候，会使用 mmap() 系统调用把大页映射到用户态的虚拟地址空间，然后就可以正常使用了。

2.8 DDIO

2.8.1 时代背景

当今时代，随着大数据和云计算的爆炸式增长，宽带的普及以及个人终端网络数据的日

益提高，对电信服务节点和数据中心的数据交换能力和网络带宽提出了更高的要求。并且，数据中心本身对虚拟化功能的需求也增加了更多的网络带宽需求。电信服务节点和数据中心为了应付这种需求，需要对内部的各种服务器资源进行升级。在这种环境下，英特尔公司提出了 Intel® DDIO（Data Direct I/O）的技术。该技术的主要目的就是让服务器能更快处理网络接口的数据，提高系统整体的吞吐率，降低延迟，同时减少能源的消耗。但是，DDIO 是如何做到这种优化和改进的呢？为了回答这个问题，有必要回顾一下 DDIO 技术出现之前，服务器是如何处理从网络上来的数据的。

当一个网络报文送到服务器的网卡时，网卡通过外部总线（比如 PCI 总线）把数据和报文描述符送到内存。接着，CPU 从内存读取数据到 Cache 进而到寄存器。进行处理之后，再写回到 Cache，并最终送到内存中。最后，网卡读取内存数据，经过外部总线送到网卡内部，最终通过网络接口发送出去。

可以看出，对于一个数据报文，CPU 和网卡需要多次访问内存。而内存相对 CPU 来讲是一个非常慢速的部件。CPU 需要等待数百个周期才能拿到数据，在这过程中，CPU 什么也做不了。

DDIO 技术是如何改进的呢？这种技术使外部网卡和 CPU 通过 LLC Cache 直接交换数据，绕过了内存这个相对慢速的部件。这样，就增加了 CPU 处理网络报文的速度（减少了 CPU 和网卡等待内存的时间），减小了网络报文在服务器端的处理延迟。这样做也带来了一个问题，因为网络报文直接存储在 LLC Cache 中，这大大增加了对其容量的需求，因而在英特尔的 E5 处理器系列产品中，把 LLC Cache 的容量提高到了 20MB。

图 2-11 是 DDIO 技术对网络报文的处理流程示意图。

DDIO 功能模块会学习来自 I/O 设备的读写请求，也就是 I/O 对内存的读或者写的请求。例如，当网卡需要从服务器端传送一个数据报文到网络上时，它会发起一个 I/O 读请求（读数据操作），请求把内存中的某个数据块通过外部总线送到网卡上；当网卡从网络中收到一个数据报文时，它会发起一个 I/O 写请求（写数据操作），请求把某个数据块通过外部总线送到内存中某个地址上。

接下来的章节会详细介绍在没有 DDIO 技术和有 DDIO 技术条件下，服务器是如何处理这些 I/O 读写请求的。

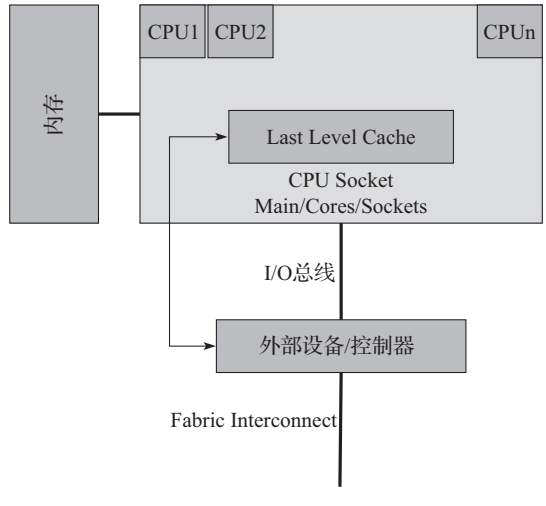


图 2-11　DDIO 中报文的处理流程

2.8.2　网卡的读数据操作

通常来说，为了发送一个数据报文到网络上去，首先是运行在 CPU 上的软件分配了一段内存，然后把这段内存读取到 CPU

内部，更新数据，并且填充相应的报文描述符（网卡会通过读取描述符了解报文的相应信息），然后写回到内存中，通知网卡，最终网卡把数据读回到内部，并且发送到网络上去。但是，没有 DDIO 技术和有 DDIO 技术条件的处理方式是不同的，图 2-12 是两种环境下的处理流程图。

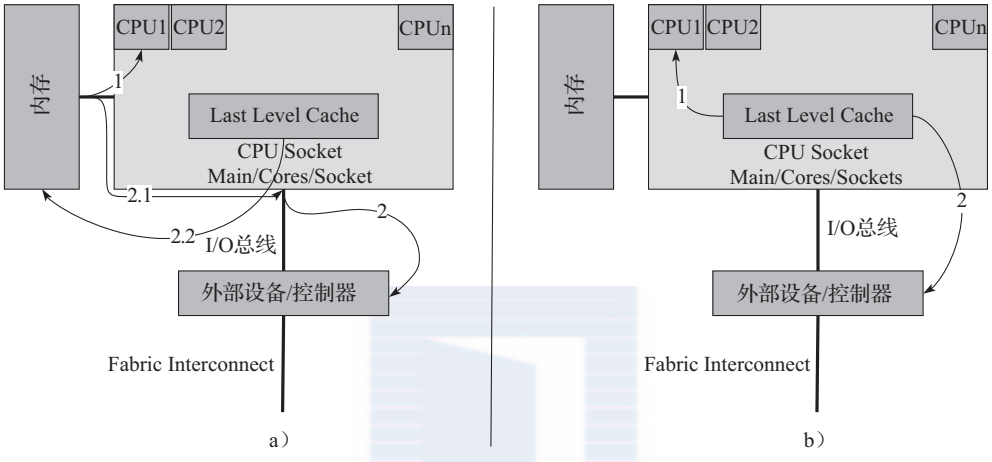


图 2-12 网卡读数据的处理流程

图 2-12a 是没有 DDIO 技术的处理流程。

1）处理器更新报文和控制结构体。由于分配的缓冲区在内存中，因此会触发一次 Cache 不命中，处理器把内存读取到 Cache 中，然后更新控制结构体和报文信息。之后通知 NIC 来读取报文。

2）NIC 收到有报文需要传递到网络上的通知后，它首先需要读取控制结构体进而知道从哪里获取报文。由于之前处理器刚把该缓冲区从内存读到 Cache 中并且做了更新，很有可能 Cache 还没有来得及把更新的内容写回到内存中。因此，当 NIC 发起一个对内存的读请求时，很有可能这个请求会发送到 Cache 系统中，Cache 系统会把数据写回到内存中，然后内存控制器再把数据写到 PCI 总线上去。因此，一个读内存的操作会产生多次内存的读写。

图 2-12b 是有 DDIO 技术的处理流程。

1）处理器更新报文和控制结构体。这个步骤和没有 DDIO 的技术类似，但是由于 DDIO 的引入，处理器会开始就把内存中的缓冲区和控制结构体预取到 Cache，因此减少了内存读的时间。

2）NIC 收到有报文需要传递到网络上的通知后，通过 PCI 总线把控制结构体和报文送到 NIC 内部。利用 DDIO 技术，I/O 访问可以直接将 Cache 的内容送到 PCI 总线上。这样，就减少了 Cache 写回时等待的时间。

由此可以看出，由于 DDIO 技术的引入，网卡的读操作减少了访问内存的次数，因而

提高了访问效率，减少了报文转发的延迟。在理想状况下，NIC 和处理器无需访问内存，直接通过访问 Cache 就可以完成更新数据，把数据送到 NIC 内部，进而送到网络上的所有操作。

2.8.3　网卡的写数据操作

网卡的写数据操作和上节讲到的网卡的读数据操作是完全相反的操作，通俗意义上来讲就是有网络报文需要送到系统内部进行处理，运行的软件可以对收到的报文进行协议分析，如果有问题可以丢弃，也可以转发出去。其过程一般是 NIC 从网络上收到报文后，通过 PCI 总线把报文和相应的控制结构体送到预先分配的内存，然后通知相应的驱动程序或者软件来处理。和之前讲到的网卡的读数据操作类似，有 DDIO 技术和没有 DDIO 技术的处理也是不一样的，以下是具体处理过程。

首先还是没有 DDIO 技术的处理流程，如图 2-13a 所示。

1）报文和控制结构体通过 PCI 总线送到指定的内存中。如果该内存恰好缓存在 Cache 中（有可能之前处理器有对该内存进行过读写操作），则需要等待 Cache 把内容先写回到内存中，然后才能把报文和控制结构体写到内存中。

2）运行在处理器上的驱动程序或者软件得到通知收到新报文，去内存中读取控制结构体和相应的报文，Cache 不命中。之所以 Cache 一定不会命中，是因为即使该内存地址在 Cache 中，在步骤 1 中也被强制写回到内存中。因此，只能从内存中读取控制结构体和报文。

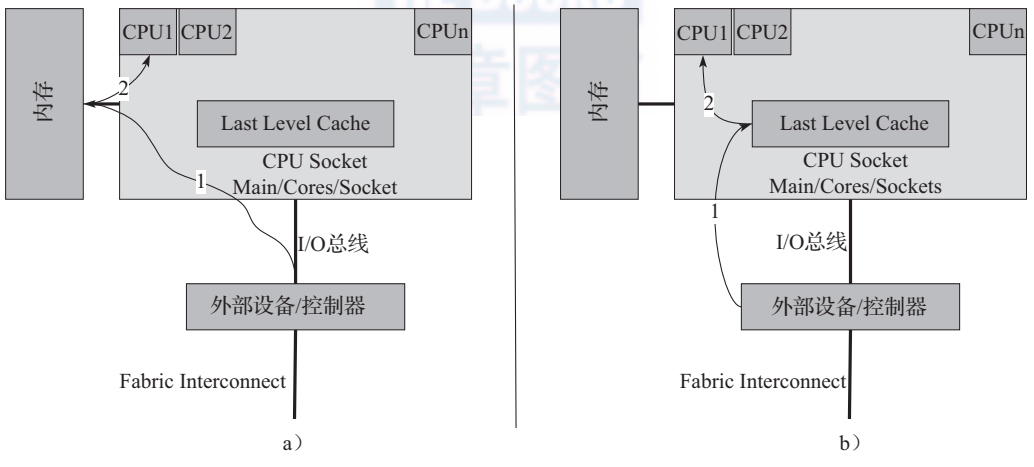


图 2-13　网卡写数据的处理流程

有 DDIO 技术的处理流程，如图 2-13b 所示。

1）这时，报文和控制结构体通过 PCI 总线直接送到 Cache 中。这时有两种情形：

a）如果该内存恰好缓存在 Cache 中（有可能之前处理器有对该内存进行过读写操

作），则直接在 Cache 中更新内容，覆盖原有内容。

b）如果该内存没有缓存在 Cache 中，则在最后一级 Cache 中分配一块区域，并相应更新 Cache 表，表明该内容是对应于内存中的某个地址的。

2）运行在处理器上的驱动或者软件被通知到有报文到达，其产生一个内存读操作，由于该内容已经在 Cache 中，因此直接从 Cache 中读。

由此可以看出，DDIO 技术在处理器和外设之间交换数据时，减少了处理器和外设访问内存的次数，也减少了 Cache 写回的等待，提高了系统的吞吐率和数据的交换延迟。

2.9 NUMA 系统

之前的章节已经简要介绍过 NUMA 系统，它是一种多处理器环境下设计的计算机内存结构。NUMA 系统是从 SMP（Symmetric Multiple Processing，对称多处理器）系统演化而来。

SMP 系统最初是在 20 世纪 90 年代由 Unisys、Convex Computer（后来的 HP）、Honeywell、IBM 等公司开发的一款商用系统，该系统被广泛应用于 Unix 类的操作系统，后来又扩展到 Windows NT 中，该系统有如下特点：

1）所有的硬件资源都是共享的。即每个处理器都能访问到任何内存、外设等。

2）所有的处理器都是平等的，没有主从关系。

3）内存是统一结构、统一寻址的（UMA，Uniform Memory Architecture）。

4）处理器和内存，处理器和处理器都通过一条总线连接起来。

其结构如图 2-14 所示：

SMP 的问题也很明显，因为所有的处理器都通过一条总线连接起来，因此随着处理器的增加，系统总线成为了系统瓶颈，另外，处理器和内存之间的通信延迟也较大。为了克服以上的缺点，才应运而生了 NUMA 架构，如图 2-15 所示。

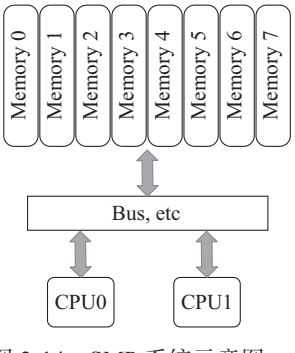


图 2-14 SMP 系统示意图

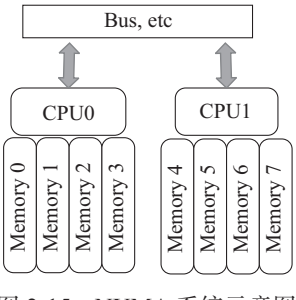


图 2-15 NUMA 系统示意图

NUMA 是起源于 AMD Opteron 的微架构，同时被英特尔 Nehalem 架构采用。在这个架构中，处理器和本地内存之间拥有更小的延迟和更大的带宽，而整个内存仍然可作为一个整

体，任何处理器都能够访问，只不过跨处理器的内存访问的速度相对较慢一点。同时，每个处理器都可以拥有本地的总线，如 PCIE、SATA、USB 等。和内存一样，处理器访问本地的总线延迟低，吞吐率高；访问远程资源，则延迟高，并且要和其他处理器共享一条总线。图 2-16 是英特尔公司的至强 E5 服务器的架构示意图。

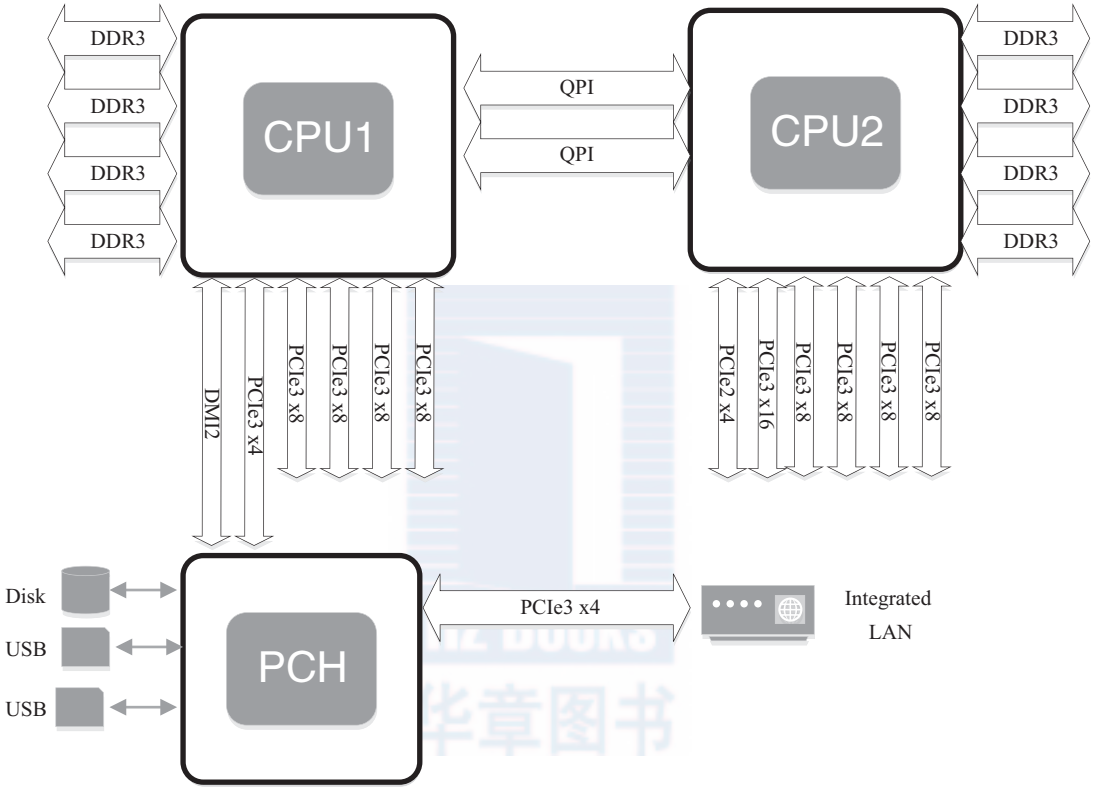


图 2-16　至强 E5 服务器架构示意图

可以看到，该架构有两个处理器，处理器通过 QPI 总线相连。每个处理器都有本地的四个通道的内存系统，并且也有属于自己的 PCIE 总线系统。两个处理器有点不同的是，第一个处理器集成了南桥芯片，而第二个处理器只有本地的 PCIE 总线。

和 SMP 系统相比，NUMA 系统访问本地内存的带宽更大，延迟更小，但是访问远程的内存成本相对就高多了。因此，我们要充分利用 NUMA 系统的这个特点，避免远程访问资源。

以下是 DPDK 在 NUMA 系统中的一些实例。

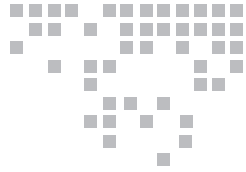
1）Per-core memory。一个处理器上有多个核（core），per-core memory 是指每个核都有属于自己的内存，即对于经常访问的数据结构，每个核都有自己的备份。这样做一方面是为了本地内存的需要，另外一方面也是因为上文提到的 Cache 一致性的需要，避免多个核访问

同一个 Cache 行。

2）本地设备本地处理。即用本地的处理器、本地的内存来处理本地的设备上产生的数据。如果有一个 PCI 设备在 node0 上，就用 node0 上的核来处理该设备，处理该设备用到的数据结构和数据缓冲区都从 node0 上分配。以下是一个分配本地内存的例子：

```
/* allocate memory for the queue structure */
q = rte_zmalloc_socket("fm10k", sizeof(*q),  RTE_CACHE_LINE_SIZE, socket_id);
```

该例试图分配一个结构体，通过传递 socket_id，即 node id 获得本地内存，并且以 Cache 行对齐。



第 3 章 Chapter 3

并行计算

处理器性能提升主要有两个途径，一个是提高 IPC（每个时钟周期内可以执行的指令条数），另一个是提高处理器主频率。每一代微架构的调整可以伴随着对 IPC 的提高，从而提高处理器性能，只是幅度有限。而提高处理器主频率对于性能的提升作用是明显而直接的。但一味地提高频率很快会触及频率墙，因为处理器的功耗正比于主频的三次方。

所以，最终要取得性能提升的进一步突破，还是要回到提高 IPC 这个因素。经过处理器厂商的不懈努力，我们发现可以通过提高指令执行的并行度来提高 IPC。而提高并行度主要有两种方法，一种是提高微架构的指令并行度，另一种是采用多核并发。这一章主要就分享这两种方法在 DPDK 中的实践，并在指令并行方法中上进一步引入数据并发的介绍。

3.1 多核性能和可扩展性

3.1.1 追求性能水平扩展

多核处理器是指在一个处理器中集成两个或者多个完整的内核（及计算引擎）。如果把处理器性能随着频率的提升看做是垂直扩展，那多核处理器的出现使性能水平扩展成为可能。原本在单核上顺序执行的任务，得以按逻辑划分成若干子任务，分别在不同的核上并行执行。在任务粒度上，使指令执行的并行度得到提升。

那随着核数的增加，性能是否能持续提升呢？ Amdahl 定律告诉我们，假设一个任务的工作量不变，多核并行计算理论时延加速上限取决于那些不能并行处理部分的比例。换句话

说，多核并行计算下时延不能随着核数增加而趋于无限小。该定律明确告诉我们，利用多核处理器提升固定工作量性能的关键在于降低那些不得不串行部分占整个任务执行的比例。更多信息可以参考［Ref3-1］。

对于 DPDK 的主要应用领域——数据包处理，多数场景并不是完成一个固定工作量的任务，更主要关注单位时间内的吞吐量。Gustafson 定律对于在固定工作时间下的推导给予我们更多的指导意义。它指出，多核并行计算的吞吐率随核数增加而线性扩展，可并行处理部分占整个任务比重越高，则增长的斜率越大。带着这个观点来读 DPDK，很多实现的初衷就豁然开朗。资源局部化、避免跨核共享、减少临界区碰撞、加快临界区完成速率（后两者涉及多核同步控制，将在下一章中介绍）等，都不同程度地降低了不可并行部分和并发干扰部分的占比。

3.1.2 多核处理器

在数据包处理领域，多核架构的处理器已经广泛应用。本节以英特尔的至强主流多核处理器为例，介绍 DPDK 中用到的一些概念，比如物理核、逻辑核、CPU node 等。

下面结合图形详细介绍了单核、多核以及超线程的概念。

通过单核结构（见图 3-1），我们先认识一下 CPU 物理核中主要的基本组件。为简化理解，将主要组件简化为：CPU 寄存器集合、中断逻辑（Local APIC）、执行单元和 Cache。一个完整的物理核需要拥有这样的整套资源，提供一个指令执行线程。

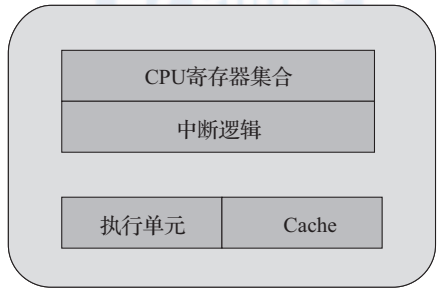


图 3-1 单核结构

多处理器结构指的是多颗单独封装的 CPU 通过外部总线连接，构成的统一计算平台，如图 3-2 所示。每个 CPU 都需要独立的电路支持，有自己的 Cache，而它们之间的通信通过主板上的总线。在此架构上，若一个多线程的程序运行在不同 CPU 的某个核上，跨 CPU 的线程间协作都要走总线，而共享的数据还会付出因 Cache 一致性产生的开销。从内存子系统的角度，多处理器结构进一步衍生出了非一致内存访问（NUMA），这一点在第 2 章就有介绍。在 DPDK 中，对于多处理器的 NUMA 结构，使用 Socket Node 来标示，跨 NUMA 的内存访问是性能调优时最需要避免的。

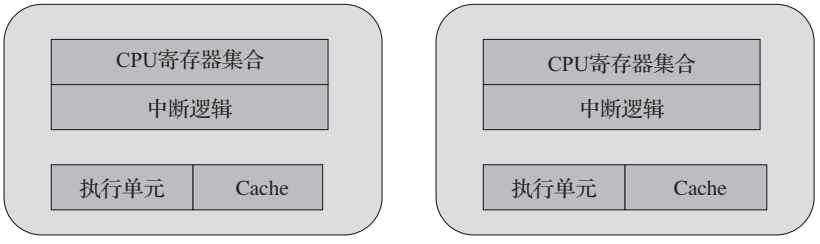


图 3-2 多处理器结构

如图 3-3 所示，超线程（Hyper-Threading）在一个处理器中提供两个逻辑执行线程，逻辑线程共享流水线、执行单元和缓存。该技术的本质是复用单处理器中的超标量流水线的多路执行单元，降低多路执行单元中因指令依赖造成的执行单元闲置。对于每个逻辑线程，拥有完整独立的寄存器集合和本地中断逻辑，从软件的角度，与单线程物理核并没有差异。例如，8 核心的处理器使用超线程技术之后，可以得到 16 个逻辑线程。采用超线程，在单核上可以同时进行多线程处理，使整体性能得到一定程度提升。但由于其毕竟是共享执行单元的，对 IPC（每周期执行指令数）越高的应用，带来的帮助越有限。DPDK 是一种 I/O 集中的负载，对于这类负载，IPC 相对不是特别高，所以超线程技术会有一定程度的帮助。更多信息可以参考［Ref3-2］。

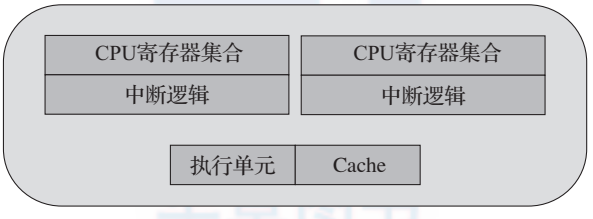


图 3-3 超线程

如果说超线程还是站在一个核内部以资源切分的方式构成多个执行线程，多核体系结构（见图 3-4）则是在一个 CPU 封装里放入了多个对等的物理核，每个物理核可以独立构成一个执行线程，当然也可以进一步分割成多个执行线程（采用超线程技术）。多核之间的通信使用芯片内部总线来完成，共享更低一级缓存（LLC，三级缓存）和内存。随着 CPU 制造工艺的提升，每个 CPU 封装中放入的物理核数也在不断提高。

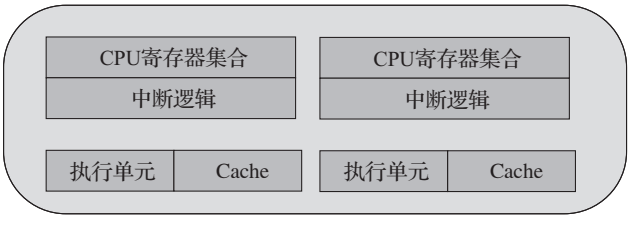


图 3-4 多核体系结构

各种架构在总线占用、Cache、寄存器以及执行单元的区别大致可以归纳为表 3-1。

表 3-1 并行计算的底层基础架构

架构类型	CPU 数量	执行单元（ALU）	架构状态信息（寄存器）	Cache	总线 / 内存 / 中断 / 外设	应用模式 / 操作系统需求	说明
单内核 / 多线程	单个，复用	单个，复用	1 套，复用	1 套，共用	共用		通过延迟隐藏提升系统性能
SMT/HT（超线程）	单个，复用	单个，复用	多套，并行	1 套，共用	共用	SMP	通过延迟隐藏提升系统性能。性能提升有限
多核	多个，独立并行运行	多个，独立并行运行	多套，并行	1 套或者多套，共享或者独立	一般为共用，也可分段使用	SMP/AMP	真正并行，理论上可达到核数 N 的加速比
多处理器	多个，独立运行	多个独立运行	多套，独立	多套独立	一般为独立使用，也可共用内存	AMP/SMP	并行，理论上可达到 CPU 数 N 的加速比

一个物理封装的 CPU（通过 physical id 区分判断）可以有多个核（通过 core id 区分判断）。而每个核可以有多个逻辑 CPU（通过 processor 区分判断）。一个核通过多个逻辑 CPU 实现这个核自己的超线程技术。

查看 CPU 内核信息的基本命令如表 3-2 所示。

表 3-2 内核信息的基本命令

CPU 信息	命 令
处理器核数	cat /proc/cpuinfo| grep“cpu cores”| uniq
逻辑处理器核数	cat /proc/cpuinfo 如果“siblings”和“cpu cores”一致，则说明不支持超线程，或者超线程未打开。如果“siblings”是“cpu cores”的两倍，则说明支持超线程，并且超线程已打开
系统物理处理器封装 ID	cat /proc/cpuinfo| grep“physical id”| sort| uniq| wc –l 或者 lscpu	grep“CPU socket”
系统逻辑处理器 ID	cat /proc/cpuinfo| grep“processor”| wc –l

处理器核数：processor cores，即俗称的“CPU 核数”，也就是每个物理 CPU 中 core 的个数，例如“Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz”是 10 核处理器，它在每个 socket 上有 10 个“处理器核”。具有相同 core id 的 CPU 是同一个 core 的超线程。

逻辑处理器核心数：sibling 是内核认为的单个物理处理器所有的超线程个数，也就是一个物理封装中的逻辑核的个数。如果 sibling 等于实际物理核数的话，就说明没有启动超线程；反之，则说明启用超线程。

系统物理处理器封装 ID：Socket 中文翻译成“插槽”，也就是所谓的物理处理器封装个数，即俗称的“物理 CPU 数”，管理员可能会称之为“路”。例如一块“Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz”有两个“物理处理器封装”。具有相同 physical id 的 CPU 是同

一个 CPU 封装的线程或核心。

系统逻辑处理器 ID：逻辑处理器数的英文名是 logical processor，即俗称的“逻辑 CPU 数”, 逻辑核心处理器就是虚拟物理核心处理器的一个超线程技术，例如“ Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz”支持超线程，一个物理核心能模拟为两个逻辑处理器，即一块“Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz”有 20 个“逻辑处理器”。

3.1.3 亲和性

当处理器进入多核架构后，自然会面对一个问题，按照什么策略将任务线程分配到各个处理器上执行。众所周知的是，这个分配工作一般由操作系统完成。负载均衡当然是比较理想的策略，按需指定的方式也是很自然的诉求，因为其具有确定性。

简单地说，CPU 亲和性（Core affinity）就是一个特定的任务要在某个给定的 CPU 上尽量长时间地运行而不被迁移到其他处理器上的倾向性。这意味着线程可以不在处理器之间频繁迁移。这种状态正是我们所希望的，因为线程迁移的频率小就意味着产生的负载小。

Linux 内核包含了一种机制，它让开发人员可以编程实现 CPU 亲和性。这意味着应用程序可以显式地指定线程在哪个（或哪些）处理器上运行。

1. Linux 内核对亲和性的支持

在 Linux 内核中，所有的线程都有一个相关的数据结构，称为 task_struct。这个结构非常重要，原因有很多；其中与亲和性相关度最高的是 cpus_allowed 位掩码。这个位掩码由 n 位组成，与系统中的 n 个逻辑处理器一一对应。具有 4 个物理 CPU 的系统可以有 4 位。如果这些 CPU 都启用了超线程，那么这个系统就有一个 8 位的位掩码。

如果针对某个线程设置了指定的位，那么这个线程就可以在相关的 CPU 上运行。因此，如果一个线程可以在任何 CPU 上运行，并且能够根据需要在处理器之间进行迁移，那么位掩码就全是 1。实际上，在 Linux 中，这就是线程的默认状态。

Linux 内核 API 提供了一些方法，让用户可以修改位掩码或查看当前的位掩码：

- sched_set_affinity()（用来修改位掩码）
- sched_get_affinity()（用来查看当前的位掩码）

注意，cpu_affinity 会被传递给子线程，因此应该适当地调用 sched_set_affinity。

2. 为什么应该使用亲和性

将线程与 CPU 绑定，最直观的好处就是提高了 CPU Cache 的命中率，从而减少内存访问损耗，提高程序的速度。

在多核体系 CPU 上，提高外设以及程序工作效率最直观的办法就是让各个物理核各自负责专门的事情。每个物理核各自也会有缓存，缓存着执行线程使用的信息，而线程可能会被内核调度到其他物理核上，这样 L1/L2 的 Cache 命中率会降低，当绑定物理核后，程序就会一直在指定核上跑，不会由操作系统调度到其他核上，省却了来回反复调度的性能消耗，

线程之间互不干扰地完成工作。

在 NUMA 架构下，这个操作对系统运行速度的提升有更大的意义，跨 NUMA 节点的任务切换，将导致大量三级 Cache 的丢失。从这个角度来看，NUMA 使用 CPU 绑定时，每个核心可以更专注地处理一件事情，资源体系被充分使用，减少了同步的损耗。

通常 Linux 内核都可以很好地对线程进行调度，在应该运行的地方运行线程（这就是说，在可用的处理器上运行并获得很好的整体性能）。内核包含了一些用来检测 CPU 之间任务负载迁移的算法，可以启用线程迁移来降低繁忙的处理器的压力。

一般情况下，在应用程序中只需使用默认的调度器行为。然而，您可能会希望修改这些默认行为以实现性能的优化。让我们来看一下使用亲和性的三个原因。

❑ 有大量计算要做

基于大量计算的情形通常出现在科学计算和理论计算中，但是通用领域的计算也可能出现这种情况。一个常见的标志是发现自己的应用程序要在多处理器的机器上花费大量的计算时间。

❑ 测试复杂的应用程序

测试复杂软件是我们对内核亲和性技术感兴趣的另外一个原因。考虑一个需要进行线性可伸缩性测试的应用程序。有些产品声明可以在使用更多硬件时执行得更好。我们不用购买多台机器（为每种处理器配置都购买一台机器），而是可以：①购买一台多处理器的机器，②不断增加分配的处理器，③测量每秒的事务数，④评估结果的可伸缩性。

如果应用程序随着 CPU 的增加可以线性地伸缩，那么每秒事务数和 CPU 个数之间应该会是线性的关系。这样建模可以确定应用程序是否可以有效地使用底层硬件。

如果一个给定的线程迁移到其他地方去了，那么它就失去了利用 CPU 缓存的优势。实际上，如果正在使用的 CPU 需要为自己缓存一些特殊的数据，那么所有其他 CPU 都会使这些数据在自己的缓存中失效。

因此，如果有多个线程都需要相同的数据，那么将这些线程绑定到一个特定的 CPU 上是非常有意义的，这样就确保它们可以访问相同的缓存数据（或者至少可以提高缓存的命中率）。

否则，这些线程可能会在不同的 CPU 上执行，这样会频繁地使其他缓存项失效。

❑ 运行时间敏感的、决定性的线程

我们对 CPU 亲和性感兴趣的最后一个原因是实时（对时间敏感的）线程。例如，您可能会希望使用亲和性来指定一个 8 路主机上的某个处理器，而同时允许其他 7 个处理器处理所有普通的系统调度。这种做法确保长时间运行、对时间敏感的应用程序可以得到运行，同时可以允许其他应用程序独占其余的计算资源。下面的应用程序显示了这是如何工作的。

3. 线程独占

DPDK 通过把线程绑定到逻辑核的方法来避免跨核任务中的切换开销，但对于绑定运行

的当前逻辑核，仍然可能会有线程切换的发生，若希望进一步减少其他任务对于某个特定任务的影响，在亲和的基础上更进一步，可以采取把逻辑核从内核调度系统剥离的方法。

Linux 内核提供了启动参数 isolcpus。对于有 4 个 CPU 的服务器，在启动的时候加入启动参数 isolcpus=2,3。那么系统启动后将不使用 CPU3 和 CPU4。注意，这里说的不使用不是绝对地不使用，系统启动后仍然可以通过 taskset 命令指定哪些程序在这些核心中运行。步骤如下所示。

命令：vim /boot/grub2.cfg

在 Linux kernel 启动参数里面加入 isolcpus 参数，isolcpu=2,3。

命令：cat /proc/cmdline

等待系统重新启动之后查看启动参数 BOOT_IMAGE=/boot/vmlinuz-3.17.8-200.fc20.x86_64 root=UUID=3ae47813-79ea-4805-a732-21bedcbdb0b5 ro LANG=en_US.UTF-8 isolcpus=2,3。

3.1.4 DPDK 的多线程

DPDK 的线程基于 pthread 接口创建，属于抢占式线程模型，受内核调度支配。DPDK 通过在多核设备上创建多个线程，每个线程绑定到单独的核上，减少线程调度的开销，以提高性能。

DPDK 的线程可以作为控制线程，也可以作为数据线程。在 DPDK 的一些示例中，控制线程一般绑定到 MASTER 核上，接受用户配置，并传递配置参数给数据线程等；数据线程分布在不同核上处理数据包。

1. EAL 中的 lcore

DPDK 的 lcore 指的是 EAL 线程，本质是基于 pthread（Linux/FreeBSD）封装实现。Lcore（EAL pthread）由 remote_launch 函数指定的任务创建并管理。在每个 EAL pthread 中，有一个 TLS（Thread Local Storage）称为 _lcore_id。当使用 DPDK 的 EAL ‘ -c ’ 参数指定 coremask 时，EAL pthread 生成相应个数 lcore 并默认是 1:1 亲和到 coremask 对应的 CPU 逻辑核，_lcore_id 和 CPU ID 是一致的。

下面简单介绍 DPDK 中 lcore 的初始化及执行任务的注册。

（1）初始化

1）rte_eal_cpu_init() 函数中，通过读取 /sys/devices/system/cpu/cpuX/ 下的相关信息，确定当前系统有哪些 CPU 核，以及每个核属于哪个 CPU Socket。

2）eal_parse_args() 函数，解析 -c 参数，确认哪些 CPU 核是可以使用的，以及设置第一个核为 MASTER。

3）为每一个 SLAVE 核创建线程，并调用 eal_thread_set_affinity() 绑定 CPU。线程的执行体是 eal_thread_loop()。eal_thread_loop() 的主体是一个 while 死循环，调用不同模块注册

到 lcore_config[lcore_id].f 的回调函数。

```
RTE_LCORE_FOREACH_SLAVE(i) {

        /*
         * create communication pipes between master thread
         * and children
         */
        if (pipe(lcore_config[i].pipe_master2slave) < 0)
              rte_panic("Cannot create pipe\n");
        if (pipe(lcore_config[i].pipe_slave2master) < 0)
              rte_panic("Cannot create pipe\n");

        lcore_config[i].state = WAIT;

        /* create a thread for each lcore */
        ret = pthread_create(&lcore_config[i].thread_id, NULL,
                        eal_thread_loop, NULL);
        if (ret != 0)
              rte_panic("Cannot create thread\n");
    }
```

（2）注册

不同的模块需要调用 rte_eal_mp_remote_launch()，将自己的回调处理函数注册到 lcore_config[].f 中。以 l2fwd 为例，注册的回调处理函数是 l2fwd_launch_on_lcore()。

```
rte_eal_mp_remote_launch(l2fwd_launch_one_lcore, NULL, CALL_MASTER);
```

DPDK 每个核上的线程最终会调用 eal_thread_loop()--->l2fwd_launch_on_lcore()，调用到自己实现的处理函数。

最后，总结整个 lcore 启动过程和执行任务分发，可以归纳为如图 3-5 所示。

2. lcore 的亲和性

默认情况下，lcore 是与逻辑核一一亲和绑定的。带来性能提升的同时，也牺牲了一定的灵活性和能效。在现网中，往往有流量潮汐现象的发生，在网络流量空闲时，没有必要使用与流量繁忙时相同的核数。按需分配和灵活的扩展伸缩能力，代表了一种很有说服力的能效需求。于是，EAL pthread 和逻辑核之间进而允许打破 1:1 的绑定关系，使得 _lcore_id 本身和 CPU ID 可以不严格一致。EAL 定义了长选项“--lcores”来指定 lcore 的 CPU 亲和性。对一个特定的 lcore ID 或者 lcore ID 组，这个长选项允许为 EAL pthread 设置 CPU 集。

格式如下：

```
--lcores='<lcore_set>[@cpu_set][,<lcore_set>[@cpu_set],...]'
```

其中，‘lcore_set’和‘cpu_set’可以是一个数字、范围或者一个组。数字值是

“digit([0-9]+)”；范围是“<number>-<number>”；group 是“(<number|range>[,<number|range>,...])”。如果不指定‘@cpu_set’的值，那么默认就使用‘lcore_set’的值。这个选项与 corelist 的选项‘-l’是兼容的。

```
For example, "--lcores='1,2@(5-7),(3-5)@(0,2),(0,6),7-8'" which means start 9 EAL thread;
    lcore 0 runs on cpuset 0x41 (cpu 0,6);
    lcore 1 runs on cpuset 0x2 (cpu 1);
    lcore 2 runs on cpuset 0xe0 (cpu 5,6,7);
    lcore 3,4,5 runs on cpuset 0x5 (cpu 0,2);
    lcore 6 runs on cpuset 0x41 (cpu 0,6);
    lcore 7 runs on cpuset 0x80 (cpu 7);
    lcore 8 runs on cpuset 0x100 (cpu 8).
```

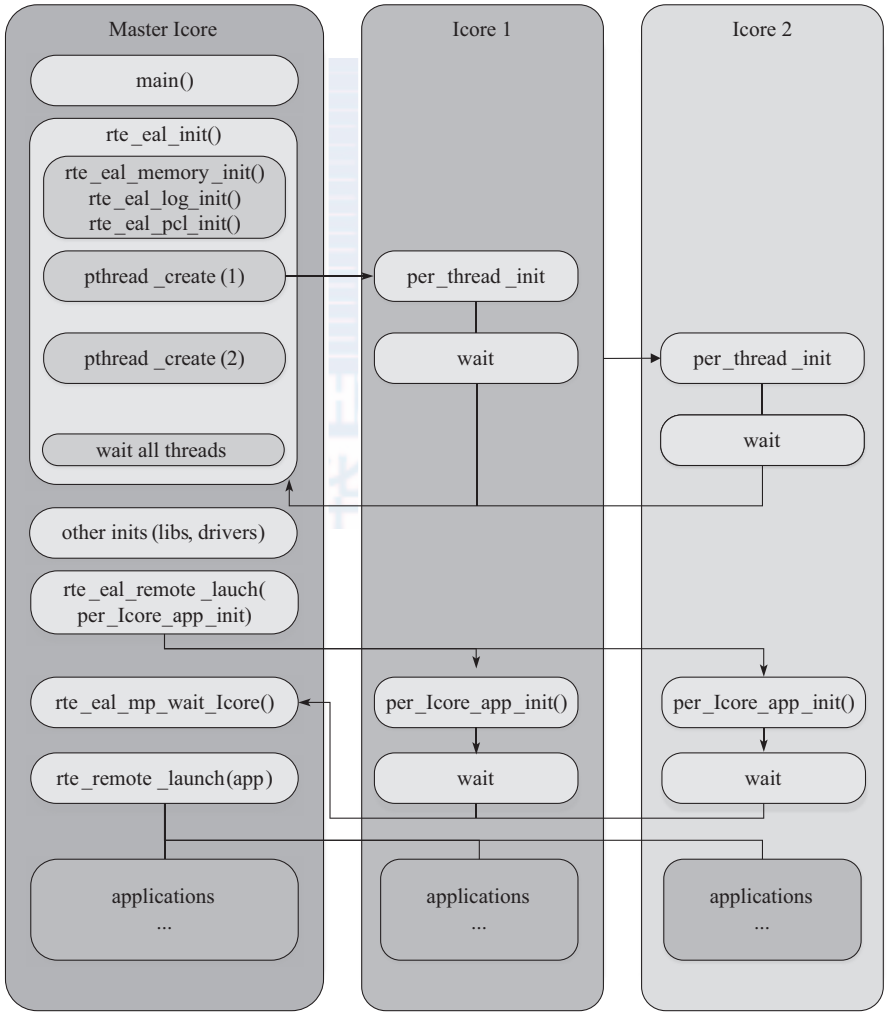


图 3-5 lcore 初始化及执行任务分发

这个选项以及对应的一组 API（rte_thread_set/get_affinity()）为 lcore 提供了亲和的灵活性。lcore 可以亲和到一个 CPU 或者一个 CPU 集合，使得在运行时调整具体某个 CPU 承载 lcore 成为可能。

而另一个方面，多个 lcore 也可能亲和到同一个核。这里要注意的是，同一个核上多个可抢占式的任务调度涉及非抢占式的库时，会有一定限制。这里以非抢占式无锁 rte_ring 为例：

1）单生产者 / 单消费者模式，不受影响，可正常使用。

2）多生产者 / 多消费者模式且 pthread 调度策略都是 SCHED_OTHER 时，可以使用，性能会有所影响。

3）多生产者 / 多消费者模式且 pthread 调度策略有 SCHED_FIFO 或者 SCHED_RR 时，建议不使用，会产生死锁。

3. 对用户 pthread 的支持

除了使用 DPDK 提供的逻辑核之外，用户也可以将 DPDK 的执行上下文运行在任何用户自己创建的 pthread 中。在普通用户自定义的 pthread 中，lcore id 的值总是 LCORE_ID_ANY，以此确定这个 thread 是一个有效的普通用户所创建的 pthread。用户创建的 pthread 可以支持绝大多数 DPDK 库，没有任何影响。但少数 DPDK 库可能无法完全支持用户自创建的 pthread，如 timer 和 Mempool。以 Mempool 为例，在用户自创建的 pthread 中，将不会启用每个核的缓存队列（Mempool cache），这个会对最佳性能造成一定影响。更多影响可以参见开发者手册的多线程章节。

4. 有效地管理计算资源

我们知道，如果网络吞吐很大，超过一个核的处理能力，可以加入更多的核来均衡流量提高整体计算能力。但是，如果网络吞吐比较小，不能耗尽哪怕是一个核的计算能力，如何能够释放计算资源给其他任务呢？

通过前面的介绍，我们了解到了 DPDK 的线程其实就是普通的 pthread。使用 cgroup 能把 CPU 的配额灵活地配置在不同的线程上。cgroup 是 control group 的缩写，是 Linux 内核提供的一种可以限制、记录、隔离进程组所使用的物理资源（如：CPU、内存、I/O 等）的机制。DPDK 可以借助 cgroup 实现计算资源配额对于线程的灵活配置，可以有效改善 I/O 核的闲置利用率。

3.2 指令并发与数据并行

前面我们花了较大篇幅讲解多核并发对于整体性能提升的帮助，从本节开始，我们将从另外一个维度——指令并发，站在一个更小粒度的视角，去理解指令级并发对于性能提升的帮助。

3.2.1 指令并发

现代多核处理器几乎都采用了超标量的体系结构来提高指令的并发度，并进一步地允许对无依赖关系的指令乱序执行。这种用空间换时间的方法，极大提高了 IPC，使得一个时钟周期完成多条指令成为可能。

图 3-6 中 Haswell 微架构流水线是 Haswell 微架构的流水线参考，从中可以看到 Scheduler 下挂了 8 个 Port，这表示每个 core 每个时钟周期最多可以派发 8 条微指令操作。具体到指令的类型，比如 Fast LEA，它可以同时在 Port 1 和 Port 5 上派发。换句话说，该指令具有被多发的能力。可以简单地理解为，该指令先后操作两个没有依赖关系的数据时，两条指令有可能被处理器同时派发到执行单元执行，由此该指令实际执行的吞吐率就提升了一倍。

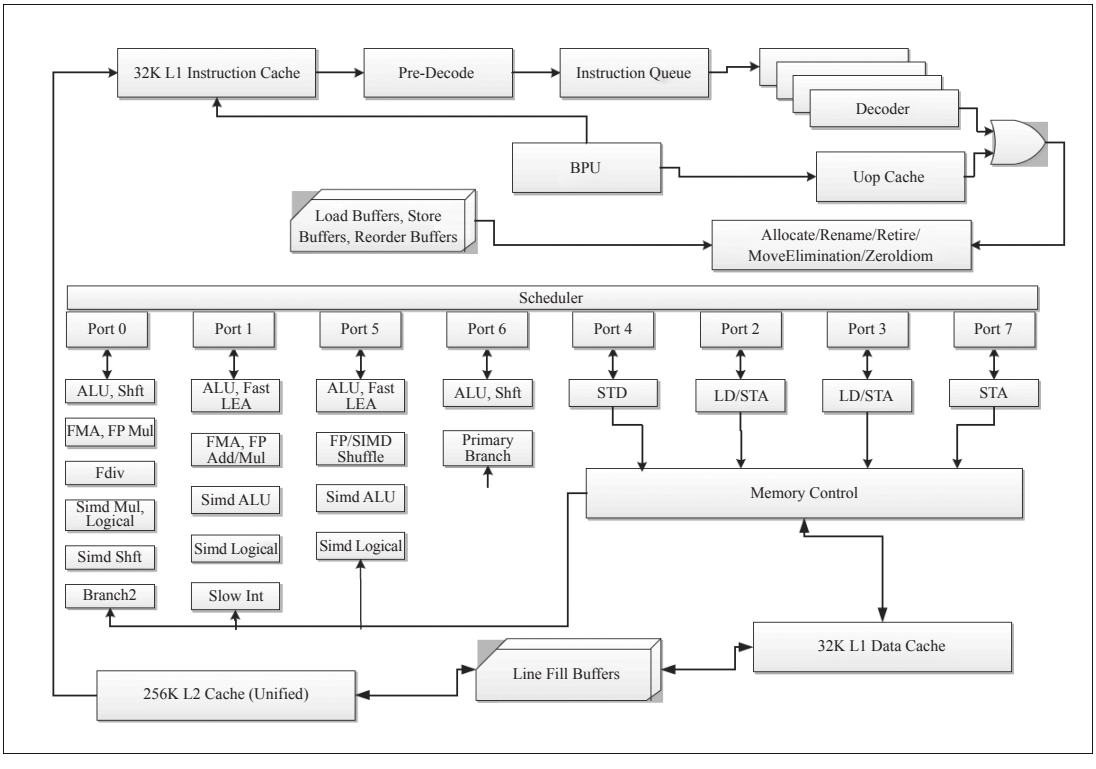


图 3-6 Haswell 微架构流水线

虽然处理器内部发生的指令并发过程，对于开发者是透明的。但不同的代码逻辑、数据依赖、存储布局等，会影响 CPU 运行时指令的派发，最终影响程序运行的 IPC。由于涉及的内容非常广泛，本书限于篇幅有限不能一一展开。理解处理器的体系结构以及微架构的设计，对于调优或者高效的代码设计都会很有帮助。这里推荐读者阅读 64-ia-32 架构优化手册，手册中会从前端优化、执行 core 优化、访存优化、预取等多个方面讲解各类技巧。

3.2.2 单指令多数据

在进入到什么是“单指令多数据”之前，先简单认识一下它的意义。“单指令多数据”给了我们这样一种可能，即使某条指令本身不再能被并（多）发，我们依旧可以从数据位宽的维度上提升并行度，从而得到整体性能提升。

3.2.2.1 SIMD 简介

SIMD 是 Single-Instruction Multiple-Data（单指令多数据）的缩写，从字面的意思就能理解大致的含义。多数据指以特定宽度为一个数据单元，多单元数据独立操作。而单指令指对于这样的多单元数据集，一个指令操作作用到每个数据单元。可以把 SIMD 理解为向量化的操作方式。典型 SIMD 操作如图 3-7 所示，两组各 4 个数据单元（X1，X2，X3，X4 和 Y1，Y2，Y3，Y4）并行操作，相同操作作用在相应的数据单元对上（X1 和 Y1，X2 和 Y2，X3 和 Y3，X4 和 Y4），4 对计算结果组成最后的 4 数据单元数。

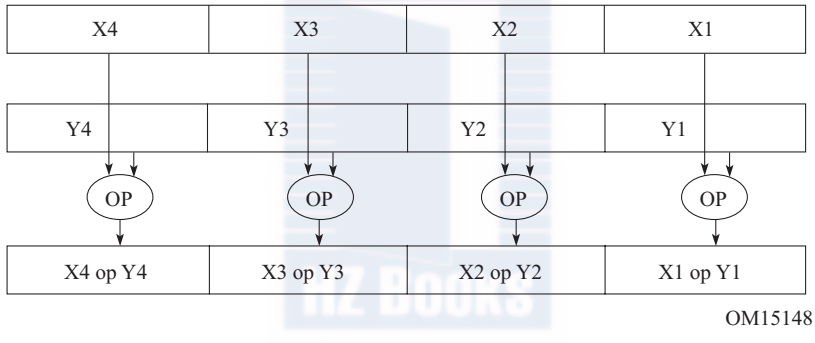


图 3-7 典型 SIMD 操作

SIMD 指令操作的寄存器相对于通用寄存器（general-purpose register，RPRS）更宽，128bit 的 XMM 寄存器或者 256bit 的 YMM 寄存器，有 2 倍甚至 4 倍于通用寄存器的宽度（在 64bit 架构上）。所以，用 SIMD 指令的一个直接好处是最大化地利用一级缓存访存的带宽，以表 3-3 所示 Haswell 微架构中第一级 Cache 参数为例，每时钟周期峰值带宽为 64B（load）（注：每周期支持两个 load 微指令，每个微指令获取最多 32B 数据）+32B（store）。可见，该微架构单时钟周期可以访存的最大数据宽度为 32B 即 256bit，只有 YMM 寄存器宽度的单指令 load 或者 store，可以用尽最大带宽。

表 3-3 Haswell 微架构中第一级 Cache 参数

Level	容量 / 每路 Cache	行的尺寸（单位：字节）	最快时延（单位：时钟周期）	加载吞吐率（单位：时钟周期）	带宽峰值（单位：字节 / 时钟周期）	更新策略
First Level Data	32KB/8	64	4	0.5	64（Load）+32（Store）	回写

对于 I/O 密集的负载，如 DPDK，最大化地利用访存带宽，减少处理器流水线后端因 I/O

访问造成的 CPU 失速，会对性能提升有显著的效果。所以，DPDK 在多个基础库中都有利用 SIMD 做向量化的优化操作。然而，也并不是所有场景都适合使用 SIMD，由于数据位较宽，对繁复的窄位宽数据操作副作用比较明显，有时数据格式调整的开销可能更大，所以选择使用 SIMD 时要仔细评估好负载的特征。

图 3-8 所示的 128 位宽的 XMM 和 256 位宽的 YMM 寄存器分别对应 Intel® SSE（Streaming SIMD Extensions）和 Intel® AVX（Advanced Vector Extensions）指令集。

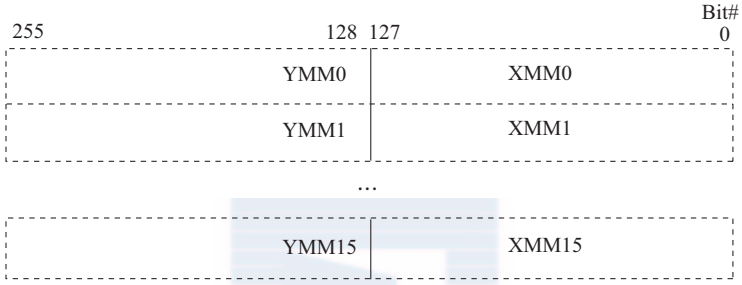


图 3-8 128 位宽和 256 位宽 SIMD 寄存器

3.2.2.2 实战 DPDK

DPDK 中的 memcpy 就利用到了 SSE/AVX 的特点。比较典型的就是 rte_memcpy 内存拷贝函数。内存拷贝是一个非常简单的操作，算法上并无难度，关键在于很好地利用处理器的各种并行特性。当前 Intel 的处理器（例如 Haswell、Sandy Bridge 等）一个指令周期内可以执行两条 Load 指令和一条 Store 指令，并且支持 SIMD 指令（SSE/AVX）来在一条指令中处理多个数据，其 Cache 的带宽也对 SIMD 指令进行了很好的支持。因此，在 rte_memcpy 中，我们使用了平台所支持的最大宽度的 Load 和 Store 指令（Sandy Bridge 为 128bit，Haswell 为 256bit）。此外，由于非对齐的存取操作往往需要花费更多的时钟周期，rte_memcpy 优先保证 Store 指令存储的地址对齐，利用处理器每个时钟周期可以执行两条 Load 这个超标量特性来弥补一部分非对齐 Load 所带来的性能损失。更多信息可以参考［Ref3-3］。

例如，在 Haswell 上，对于大于 512 字节的拷贝，需要按照 Store 地址进行对齐。

```
/**
 * Make store aligned when copy size exceeds 512 bytes
 */
dstofss = 32 - ((uintptr_t)dst & 0x1F);
n -= dstofss;
rte_mov32((uint8_t *)dst, (const uint8_t *)src);
src = (const uint8_t *)src + dstofss;
dst = (uint8_t *)dst + dstofss;
```

在 Sandy Bridge 上，由于非对齐的 Load/Store 所带来的的额外性能开销非常大，因此，除了使得 Store 对齐之外，Load 也需要进行对齐。在操作中，对于非对齐的 Load，将其首尾

未对齐部分多余的位也加载进来，因此，会产生比 Store 指令多一条的 Load。

```
xmm0 = _mm_loadu_si128((const __m128i *)((const uint8_t *)src - offset + 0 * 16));
len -= 128;
xmm1 = _mm_loadu_si128((const __m128i *)((const uint8_t *)src - offset + 1 * 16));
xmm2 = _mm_loadu_si128((const __m128i *)((const uint8_t *)src - offset + 2 * 16));
xmm3 = _mm_loadu_si128((const __m128i *)((const uint8_t *)src - offset + 3 * 16));
xmm4 = _mm_loadu_si128((const __m128i *)((const uint8_t *)src - offset + 4 * 16));
xmm5 = _mm_loadu_si128((const __m128i *)((const uint8_t *)src - offset + 5 * 16));
xmm6 = _mm_loadu_si128((const __m128i *)((const uint8_t *)src - offset + 6 * 16));
xmm7 = _mm_loadu_si128((const __m128i *)((const uint8_t *)src - offset + 7 * 16));
xmm8 = _mm_loadu_si128((const __m128i *)((const uint8_t *)src - offset + 8 * 16));
src = (const uint8_t *)src + 128;
_mm_storeu_si128((__m128i *)((uint8_t *)dst + 0 * 16), _mm_alignr_epi8(xmm1,
xmm0, offset));
_mm_storeu_si128((__m128i *)((uint8_t *)dst + 1 * 16), _mm_alignr_epi8(xmm2,
xmm1, offset));
_mm_storeu_si128((__m128i *)((uint8_t *)dst + 2 * 16), _mm_alignr_epi8(xmm3,
xmm2, offset));
_mm_storeu_si128((__m128i *)((uint8_t *)dst + 3 * 16), _mm_alignr_epi8(xmm4,
xmm3, offset));
_mm_storeu_si128((__m128i *)((uint8_t *)dst + 4 * 16), _mm_alignr_epi8(xmm5,
xmm4, offset));
_mm_storeu_si128((__m128i *)((uint8_t *)dst + 5 * 16), _mm_alignr_epi8(xmm6,
xmm5, offset));
_mm_storeu_si128((__m128i *)((uint8_t *)dst + 6 * 16), _mm_alignr_epi8(xmm7,
xmm6, offset));
_mm_storeu_si128((__m128i *)((uint8_t *)dst + 7 * 16), _mm_alignr_epi8(xmm8,
xmm7, offset));
dst = (uint8_t *)dst + 128;
```


3.3 小结

多核采用这种“横向扩展”的方法来提高系统的性能，该架构实现了“分治法”策略。通过划分任务，线程应用能够充分利用多个执行内核，并且可以在特定时间内执行更多任务。它的优点是能够充分并且灵活地分配 CPU，使它们的利用率最大化。但是，增加了上下文切换以及缓存命中率的开销。总之，由于多个核的存在，多核同步问题也是一个重要部分，由于很难严格做到每个核都不相关，因此引入无锁结构，这将在以后做更进一步介绍。