中国科学技术大学

博士学位论文

基于可编程网卡的

高性能数据中心系统

作者姓名：　李博杰

学科专业：　计算机软件与理论

导师姓名：　陈恩红 教授　张霖涛 教授

完成时间：　二〇一九年五月二十六日

University of Science and Technology of China

A dissertation for doctor's degree

High Performance Data Center Systems with Programmable Network Interface Cards

Author: Bojie Li

Speciality: Computer Software and Theory

Supervisors: Prof. Enhong Chen, Prof. Lintao Zhang

Finished time: May 26, 2019

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____ 签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

☑ 公开 ☐ 保密（____ 年）

作者签名：_____ 导师签名：_____

签字日期：_____ 签字日期：_____

摘　要

数据中心是支持当今世界各种互联网服务的基础设施，面临硬件和应用两方面的挑战。硬件方面，通用处理器的性能提升逐渐放缓；应用方面，大数据与机器学习对算力的需求与日俱增。不同于容易并行的 Web 服务，大数据与机器学习需要各计算节点间更多的通信，这推动了数据中心网络性能的快速提高，也对共享数据存储的性能提出了更高的要求。然而，数据中心的网络和存储基础设施主要使用通用处理器上的软件处理，其性能落后于快速增长的网络、存储、定制化计算硬件性能，日益成为系统的瓶颈。与此同时，在云化的数据中心中，灵活性也是一项重要需求。为了同时提供高性能和灵活性，近年来，可编程网卡在数据中心被广泛部署，利用现场可编程门阵列（FPGA）等定制化硬件加速虚拟网络。

本文旨在探索基于可编程网卡的高性能数据中心系统。可编程网卡在加速虚拟网络之外，还可以加速网络功能、数据结构、操作系统等。为此，本文用 FPGA 可编程网卡实现云计算数据中心计算、网络、内存存储节点的全栈加速。

首先，本文提出用可编程网卡加速云计算中的虚拟网络功能，设计和实现了首个在商用服务器中用 FPGA 加速的高灵活性、高性能网络功能处理平台 ClickNP。为了简化 FPGA 编程，本文设计了类 C 的 ClickNP 语言和模块化的编程模型，并开发了一系列优化技术，以充分利用 FPGA 的海量并行性；实现了 ClickNP 开发工具链，可以与多种商用高层次综合工具集成；基于 ClickNP 设计和实现了 200 多个网络元件，并用这些元件组建起多种网络功能。相比基于 CPU 的软件网络功能，ClickNP 的吞吐量提高了 10 倍，延迟降低到 1/10。

其次，本文提出用可编程网卡加速远程数据结构访问。本文基于 ClickNP 编程框架，设计实现了一个高性能内存键值存储系统 KV-Direct，在服务器端绕过 CPU，用可编程网卡通过 PCIe 直接访问远程主机内存中的数据结构。通过把单边 RDMA 的内存操作语义扩展到键值操作语义，KV-Direct 解决了单边 RDMA 操作数据结构时通信和同步开销高的问题。利用 FPGA 可重配置的特性，KV-Direct 允许用户实现更复杂的数据结构。面对网卡与主机内存之间 PCIe 带宽较低、延迟较高的性能挑战，通过哈希表、内存分配器、乱序执行引擎、负载均衡和缓存、向量操作等一系列性能优化，KV-Direct 实现了 10 倍于 CPU 的能耗效率和微秒级的延迟，是首个单机性能达到 10 亿次每秒的通用键值存储系统。

最后，本文提出用可编程网卡和用户态运行库相结合的方法为应用程序提供套接字通信原语，从而绕过操作系统内核。本文设计和实现了一个用户态套接字系统 SocksDirect，与现有应用程序完全兼容，能实现接近硬件极限的吞吐量

和延迟，多核性能具有可扩放性，并在高并发负载下保持高性能。主机内和主机间的通信分别使用共享内存和 RDMA 实现。为了支持高并发连接数，本文基于 KV-Direct 实现了一个 RDMA 可编程网卡。通过消除线程间同步、缓冲区管理、大数据拷贝、进程唤醒等一系列开销，SocksDirect 相比 Linux 提升了 7 至 20 倍吞吐量，降低延迟到 1/17 至 1/35，并将 Web 服务器的 HTTP 延迟降低到 1/5.5。

关键词：数据中心；可编程网卡；现场可编程门阵列；网络功能虚拟化；键值存储；网络协议栈

ABSTRACT

Data centers are the infrastructure that hosts Internet services all around the world. Data centers face challenges on hardware and application. On the hardware side, performance improvement of general processors is slowing down. On the application side, big data and machine learning impose increasing computational power requirements. Different from Web services that are easy to parallelize, big data and machine learning require more communication among compute nodes, which pushes the performance of data center network to improve rapidly, and also proposes higher requirements for shared data storage performance. However, networking and storage infrastructure services in data centers still mainly use software processing on general processors, whose performance lags behind the rapidly increasing performance of hardware in networking, storage and customized computing. As a result, software processing becomes a bottleneck in data center systems. In the meantime, in cloud data centers, flexibility is also of great importance. To provide high performance and flexibility at the same time, recent years witnessed large scale deployment of programmable NICs (Network Interface Cards) in data centers, which use customized hardware such as FPGAs to accelerate network virtualization services.

This thesis aims to explore high performance data center systems with programmable NICs. Besides accelerating network virtualization, programmable NICs can also accelerate network functions, data structures and operating systems. For this purpose, this thesis proposes a system that uses FPGA-based programmable NIC for full stack acceleration of compute, network and in-memory storage nodes in cloud data centers.

First, this thesis proposes to accelerate virtualized network functions in the cloud with programmable NICs. This thesis proposes ClickNP, the first FPGA accelerated network function processing platform on commodity servers with high flexibility and high performance. To simplify FPGA programming, this thesis designs a C-like ClickNP language and a modular programming model, and also develops optimization techniques to fully exploit the massive parallelism inside FPGA. The ClickNP tool-chain integrates with multiple commercial high-level synthesis tools. Based on ClickNP, this thesis designs and implements more than 200 network elements, and constructs various network functions using the elements. Compared to CPU-based software network functions, ClickNP improves throughput by 10 times and reduces latency

to 1/10.

Second, this thesis proposes to accelerate remote data structure access with programmable NICs. This thesis designs and implements KV-Direct, a high performance in-memory key-value storage system based on ClickNP programming framework. KV-Direct bypasses CPU on the server side and uses programmable NICs to directly access data structures in remote host memory via PCIe. KV-Direct extends memory semantics of one-sided RDMA to key-value semantics and therefore avoid the communication and synchronization overheads in data structure operations. KV-Direct further leverages the reconfigurability of FPGA to enable users to implement more complicated data structures. To tackle with the performance challenge of limited PCIe bandwidth and high latency between NIC and host memory, this thesis design a series of optimizations including hash table, memory allocator, out-of-order execution engine, load balancing, caching and vector operations. KV-Direct achieves 10 times power efficiency than CPU and microsecond scale latency. KV-Direct is the first general key-value storage system that achieves 1 billion operations per second performance on a single server.

Lastly, this thesis proposes to co-design programmable NICs and user-space libraries to provide kernel-bypass socket communication primitives for applications. This thesis designs and implements SocksDirect, a user-space socket system that is fully compatible with existing applications, achieves throughput and latency that are close to hardware limits, has scalable performance for multi-cores, and preserves high performance with many concurrent connections. SocksDirect uses shared memory and RDMA for intra-host and inter-host communication, respectively. To support many concurrent connections, SocksDirect implements an RDMA programmable NIC based on KV-Direct. SocksDirect further removes overheads such as thread synchronization, buffer management, large payload copying and process wakeup. Compared to Linux, SocksDirect improves throughput by 7 to 20 times, reduces latency to 1/17 to 1/35, and reduces HTTP latency of Web servers to 1/5.5.

Key Words: Data Center; Programmable NIC; FPGA; Network Function Virtualization; Key-Value Store; Networking Stack

目　录

第 1 章　绪　　论

1.1　研究的背景和意义

数据中心是互联网的“大脑”，也是人类存储海量数据、进行大规模计算和提供互联网服务的基础设施。21 世纪第一个十年，数据中心主要处理 Web 网站、搜索引擎等容易并行的任务；通用处理器的高速性能提升也使得专用硬件的优势不甚明显。因此，互联网数据中心多使用大量低成本的标准服务器搭建[1]。

近十年来，大数据与人工智能的兴起改变了数据中心的应用负载特性。一方面，大数据处理、机器学习等负载对算力要求很高。然而，由于摩尔定律的放缓和 Dennard 缩放定律的终结，近十年来，通用处理器的频率提升和多核核数增加都受到功耗墙的限制[2]。因此，通用处理器性能提升“免费的午餐”已经结束，体系结构的创新迎来了春天，GPU、FPGA、TPU[3] 等定制化硬件在数据中心内大量部署。另一方面，大数据处理、机器学习等负载需要多个节点紧密协同处理，对节点间的通信带宽和延迟要求较高。为了给分布式系统高效地提供消息传递和共享内存两种进程间通信范式，在网络方面需要实现高效的消息传递，在存储层面需要实现高性能的共享数据结构存储。因此，近十年来，数据中心网络从 1 Gbps 发展到 40 Gbps，并有向 100 Gbps 演进的趋势。定制化硬件之间的专用互连也成为趋势。

与此同时，数据中心的运营模式也在逐步云化，即少数几家云厂商维护数据中心的基础架构，IT 企业只需按需从云厂商租用计算、网络、存储等资源。在云数据中心中，不同的租户（tenant）共享一个巨大的计算、存储和网络资源池。为了实现资源共享和性能隔离，数据中心需要虚拟化的计算、存储和网络。如图 1.1 所示，在基础设施作为服务（IaaS）的云服务模式下，计算节点上需要提供虚拟网络、虚拟云存储、虚拟本地存储等服务，实际的网络和云存储资源则位于独立的网络节点和存储节点上。计算节点上的虚拟网络和存储服务把数据中心内物理上分散的网络和存储资源虚拟化成逻辑上统一调配的资源（“多虚一”），如同一台规模巨大的计算机[4]。网络和存储节点则不仅需要把物理资源共享给多个计算节点上不同租户的虚拟机使用（“一虚多”），还需要提供数据处理功能和高层抽象。例如，网络节点需要提供防火墙、负载均衡、加密隧道网关、网络地址转换（NAT）等网络功能（Network Function）[①]；存储节点需要进行数据结构处理以提供对象存储、文件系统存储等高层抽象，还需要进行复制（replication）以实现容灾；除了持久化存储，数据中心还需要提供内存数据结构存储，以支持

[①]本文中网络功能是一个专有名词，并不是指交换机和路由器等网络设备，而是指防火墙等网络基础架构中的功能。

分布式系统的通信[①]。

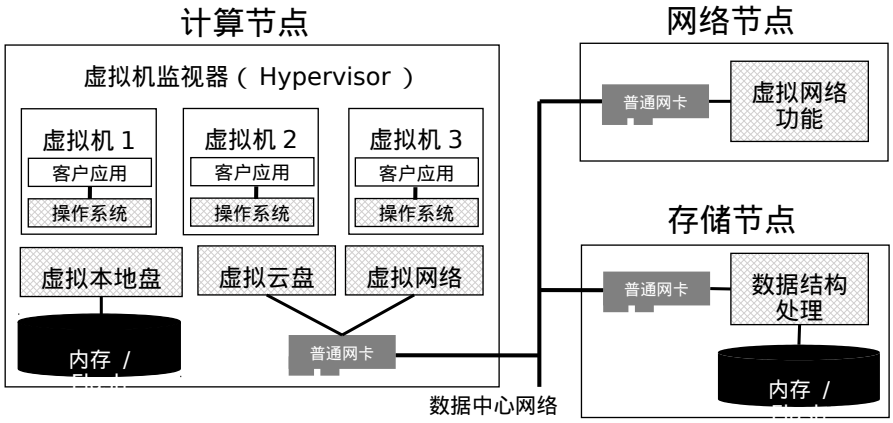


图 1.1　虚拟化的数据中心架构。

由于云服务的快速迭代，这些虚拟化的网络、存储功能还需要灵活性、可编程性和可调试性，传统上往往用运行在通用处理器上的软件实现。除了虚拟化开销，传统操作系统的开销也不容忽视。这些图 1.1 中阴影背景的方框所示的软件开销被称为“数据中心税”（data center tax）[1,4-5]。在 1 Gbps 网络和机械硬盘的时代，网络、存储虚拟化以及操作系统网络、存储协议栈引入的 CPU 开销和延迟是可以接受的。在网络、存储、定制化计算硬件越来越快的趋势下，数据中心税不仅浪费了可观的 CPU 资源，还导致应用程序无法充分利用硬件的低延迟和高吞吐量[6]。例如，本文第 4 章将说明，计算节点需要分配一部分 CPU 核专门用来实现网络和存储虚拟化，这部分核将无法出售给客户。此外，虚拟网络和网络功能都会增加数十乃至上千微秒的延迟。作为比较，数据中心网络本身的延迟只有数微秒至数十微秒，虚拟化增加的延迟比网络本身的延迟还高。第 5 章将说明，软件实现的内存共享数据结构存储与内存硬件的吞吐量相去甚远。第 6 章将说明，应用程序普遍使用操作系统中的套接字（socket）原语来进行通信，对于 Web 服务器等通信密集型的应用程序，操作系统占用了很大部分的 CPU 时间；而且，操作系统实现的套接字原语比硬件提供的远程直接内存访问（RDMA）原语延迟高一个数量级。

综上，通过软硬件结合的全栈优化降低“数据中心税”对现代数据中心的性能和成本有重要意义，这也是本文研究的课题。

[①]分布式系统的通信有消息传递和共享内存两种范式。消息传递可以直接映射到网络通信。共享内存范式对开发者更友好，可以支持高可用性和可扩放性，在网络互连的分布式系统中需要通过消息传递来实现。然而，共享内存的抽象层次较低，因此大多数分布式系统使用共享数据结构存储来取代共享内存。

1.2　国内外研究现状

为了降低“数据中心税”的开销，学术界和工业界提出了很多方案，大致可以分为优化软件、利用新型商用硬件和设计新硬件三类。

1.2.1　优化软件

传统的网络功能由部署在数据中心特定位置的专用设备实现。这些专用网络功能设备不仅价格高昂，也不够灵活，不足以支持云服务中的多租户。为此，云服务商部署了软件实现的虚拟网络功能。例如，Ananta[7] 是一个部署在微软数据中心的软件负载均衡器，用于提供云规模的负载均衡服务。RouteBricks[8] 等工作表明，基于多核 x86 CPU 的每台服务器转发数据包的速度可达 10 Gbps，并且可以通过多核和搭建更多网络节点的集群来扩展容量。虽然软件实现的虚拟交换机和网络功能可以使用更多数量的 CPU 核和更大的网络节点集群来支持更高的性能，但这样做会增加相当大的资产和运营成本[7,9]。云服务商在 IaaS 业务中的盈利能力是客户为虚拟机支付的价格与托管虚拟机的成本之间的差异。由于每台服务器的资产和运营成本在上架部署时就已基本确定，因此降低托管虚拟机成本的最佳方法是在每台计算节点服务器上打包更多的客户虚拟机，并减少网络和存储节点的服务器数量。目前，物理 CPU 核（2 个超线程，即 2 个 vCPU）的售价为 0.1 美元/小时左右，亦即最大潜在收入约为 900 美元/年[10]。在数据中心，服务器通常服役 3 到 5 年，因此一个物理 CPU 核在服务器生命周期内的售价最高可达 4500 美元[10]。即使考虑到总有一部分 CPU 核没有被售出，并且云经常为大客户提供折扣，与专用硬件相比，即使专门分配一个物理 CPU 核用于虚拟网络也是相当昂贵的。

绝大多数应用程序通过操作系统提供的套接字（socket）接口访问网络。Linux 等现有操作系统的套接字是为几十年前的低速网络设计的，在当今高吞吐量、低延迟的数据中心网络中有很高的开销。近年来，大量工作致力于提供高性能套接字。第一类工作是优化操作系统内核的 TCP/IP 网络协议栈。但是，大量的内核开销仍然存在，这将在第 6 章详细讨论。第二类工作完全绕过内核 TCP/IP 协议栈并在用户空间中实现 TCP/IP，称为用户态协议栈。在这个类别中，一些工作[11-12] 提出了新的操作系统架构，使用虚拟化来确保安全性和隔离性。除了这些新的操作系统体系结构，许多用户态协议栈利用 Linux 上已经存在的高性能数据包 I/O 框架[13-15]。其中，一些用户态协议栈[16-19] 认为 Linux 套接字 API 是性能开销的根源，进而提出了新的 API，因此需要修改应用程序。大多数 API 更改旨在支持零拷贝（zero copy）。另一些系统[20-21] 更进一步，认为套接字的抽象层次太低，应用程序应当使用更高层次的远程过程调用（RPC）接口。由于套接字的应用非常广泛，要求应用程序修改接口在很多情况下并不现实。因此，工

业界[22-24] 和学术界[25] 的一些系统提出了符合标准套接字 API 的用户态 TCP/IP 协议栈。这些用户态协议栈提供了比 Linux 更好的性能，但它仍然不接近硬件的性能极限。目前，数据中心网络内主机间通信的性能标杆是远程直接内存访问（RDMA），而主机内进程间通信最高效的方法是共享内存（shared memory）。这些用户态协议栈的主机间通信延迟比 RDMA 高一个数量级，主机内通信延迟比共享内存高一到两个数量级。

作为分布式系统通信和存储的重要基础设施，键值存储（Key-Value Storage）系统的研究和开发一直是系统学术界和工业界的热点。早期的内存键值存储系统[26] 性能不令人满意。由于分布式系统是通过网络互连的，键值存储系统的用户客户端和存储服务器之间也需要通过网络通信，引入了前面讨论的套接字网络协议栈和网络虚拟化等开销。为了摆脱操作系统内核的开销，近期的键值存储系统[27-31] 利用高性能网络数据包处理框架，轮询来自网卡的网络数据包，并且使用上文所述的用户态轻量级网络协议栈处理它们。然而，即使不考虑网络的开销，键值存储系统进行数据结构处理的成本也很高。为了降低计算成本，一系列工作[32-34] 优化锁、缓存、哈希和内存分配算法。为了降低多个 CPU 核共同处理同一个键的核间同步开销，较高效的办法是由固定的 CPU 核处理每个键的写操作[30]。然而，由于 CPU 并行性的限制，第 5 章将讨论，即使优化到极致，每个 CPU 核每秒也只能处理约 500 万次键值操作请求，远低于内存随机访存所能提供的硬件性能。此外，现实世界工作负载的键往往服从长尾分布，即少量键访问非常频繁，大多数键访问不频繁。在长尾分布负载下，由于同一个键总是映射到同一个 CPU 核，将导致 CPU 核之间的负载不平衡[30]。

1.2.2　利用新型商用硬件

由于大规模 Web 服务、大数据处理、机器学习等应用对计算和网络的性能需求非常迫切，图形处理器（GPU）等计算加速设备和远程直接内存访问（RDMA）[35] 等网络加速技术在越来越多的数据中心广泛部署。

为了加速虚拟网络和网络功能，以前的工作已经提出使用 GPU[36]，网络处理器（Network Processor，NP）[37-38] 和硬件网络交换机 ①[9]。GPU 早期主要用于图形处理，近年来扩展到具有海量数据并行性的其他应用程序。GPU 适合批量操作，但是，批量操作会导致高延迟。网络处理器的历史则可以追溯到 20 世纪 90 年代的网络交换机。网络处理器由大量的嵌入式处理器核心构成，每个处理器核心的处理能力有限，而有状态连接通常由固定的处理器核心处理，从而单条连接的吞吐量有一定的局限性。硬件网络交换机的主要问题是灵活性和查找表

①本文中，交换机（switch）与路由器不加区分，数据中心的相关文献中通常使用交换机一词指代网络互连设备。

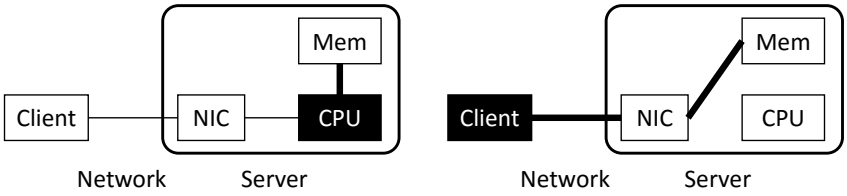
容量不足[9]。

为了降低操作系统通信原语的开销，一系列工作将操作系统网络协议栈的一部分卸载[①]到网卡硬件上。TCP 卸载引擎（TCP Offload Engine，TOE）[39]将部分或全部的 TCP/IP 协议栈卸载到网卡。但由于通用处理器的性能按摩尔定律迅速增长，这些专用硬件的性能优势有限，仅在专用领域中获得成功。取得商业成功的 TCP 卸载功能大多是无状态卸载（stateless offload）[②]，例如 TCP 校验和（checksum）的卸载。Infiniband[35] 技术设计了一套全新的通信原语、传输层协议、网络层协议和物理传输介质。其中通信原语和传输层协议称为远程直接内存访问（RDMA）。Infiniband 在硬件中实现了整个网络协议栈，达到了很高的吞吐量和很低的延迟，在高性能计算领域广泛使用。由于传输层需要维护每个连接的状态以实现拥塞控制、丢包重传、乱序重排等，Infiniband 属于有状态卸载。近年来，由于数据中心的硬件趋势和应用需求，有状态卸载的故事开始复兴[40]。基于 Infiniband 技术的远程直接内存访问（RDMA）[35] 在数据中心广泛部署[41]。为了与数据中心现有的以太网兼容，数据中心不使用 Infiniband 物理层网络，而是将 RDMA 原语和传输层协议从 Infiniband 技术栈中剥离出来，把 RDMA 传输层数据包封装在 UDP/IP 数据包中，通过以太网传输，这称为 RoCEv2[42]。与基于软件的 TCP/IP 网络协议栈相比，RDMA 使用硬件卸载来提供超低延迟和接近零的 CPU 开销。

RDMA 通信原语与应用程序通常使用的套接字通信原语有很多不同。RDMA 是基于消息的，而套接字是基于字节流的。RDMA 需要应用程序显式注册并管理发送和接收缓冲区。RDMA 还需要应用程序管理发送、接收和事件队列，并适时向网卡发送流控（flow control）通知。RDMA 提供了两类通信原语，第一类是双边（two-sided）操作，与套接字类似，发送端调用 send，接收端调用 recv。与套接字不同的是，RDMA 的接收端应用程序需要事先声明所要接收的消息，为网卡准备好接收缓冲区。另一类是单边（one-sided）操作，提供了共享内存的原语，即直接读写远程内存，或在远程内存上执行原子操作。因此，RDMA 编程比套接字复杂很多，一个用套接字收发的示例程序仅需数十行代码，而同样功能的 RDMA 程序需要数百行代码。为了使套接字应用程序能够使用 RDMA，RSocket 等工作[43-45] 将套接字操作转换为 RDMA 原语。它们具有相似的设计，其中 RSocket 的开发最活跃，是套接字转换 RDMA 的事实标准。但是，RSocket 的性能和兼容性都不令人满意。第 6 章将提出一个与现有应用兼容，并能充分利用 RDMA 网卡性能的套接字系统。

[①]在本文中，卸载（offload）是一个术语，表示把主机 CPU 上软件实现的功能用主机 CPU 以外的硬件实现。

[②]无状态（stateless）意味着网卡的内部存储在数据包处理过程是只读的。



(a) 软件 / 双边 RDMA。　　**(b)** 单边 RDMA。

图 1.2　键值存储系统的架构。线表示数据路径。一个键值操作（细线）可能需要多个基于地址的存储器访问（粗线）。黑色背景的方框表示键值数据结构处理发生的位置。

除了主机之间的网络通信，主机内的进程间通信也很重要。一些工作[11-12,43]让主机内的进程间通信绕行 RDMA 网卡。但由于 PCIe 总线的限制，RDMA 网卡的延迟和吞吐量都比主机内的共享内存差很多。因此，第 6 章使用共享内存实现主机内的进程间通信。

为了加速键值存储系统，近年来的工作[46,46-47,47] 利用 RDMA 网卡基于硬件的网络协议栈，使用双边 RDMA 作为键值存储客户端和服务器之间的远程过程调用机制，进一步提高每核吞吐量并减少延迟，如图 1.2a 所示。尽管这些工作进一步降低了网络通信的开销，但如第 1.2.1 节所讨论的，这些系统仍然使用服务器端的 CPU 进行数据结构处理，性能受限。

另一种不同的方法是利用单边 RDMA，把服务器端 CPU 的数据结构处理转移到客户端，如图 1.2b 所示。客户端通过单边 RDMA 发起对服务器端共享内存的读写请求，服务器端的网卡直接访问内存而无需经过 CPU。但是，使用共享内存模式访问数据结构往往需要多个网络往返（例如先查询索引后访问数据），增加了访问延迟，浪费了网络带宽。此外，这种模式对写入密集型的工作负载不友好。当多个客户端试图操作同一个数据结构（例如分配内存或修改同一个键值对）时，必须加锁或在客户端间进行同步，带来额外的延迟和带宽开销。

1.2.3　设计新硬件

随着通用处理器的性能提升遇到了瓶颈，各大云服务商开始探索使用定制化硬件来降低“数据中心税”，也就是把数据中心的虚拟化、操作系统和高层抽象的开销从通用处理器转移到定制化硬件上。使用定制化硬件并不是在硬件中原样实现已有的软件，而是对已有软件进行重构，划分出数据面和控制面，数据面优化后在硬件中实现，控制面则留在软件上。采用新硬件的方案尽管性能较高，但不仅需要新硬件的资产和运营成本，还需要软硬件协同设计的开发成本，往往比单纯优化软件的一次性研发（Non-Recurring Engineering，NRE）成本更高。此外，相比开发和测试软件，设计、验证和量产新硬件需要较长的研发周期。可编程硬件尽管具有一定的灵活性，但相对通用处理器是受限的，因此需要对未来

的数据中心应用负载和基础架构有一定的预见性。因此，不是所有软件都适合用硬件来加速，需要权衡成本、收益、研发周期、灵活性等多方面的因素。由于云服务的规模巨大，“数据中心税” 的开销也很广泛和显著，因此值得设计可编程硬件来加速。

为了尽可能减少计算节点上用于虚拟网络的 CPU 核，以微软 Azure 云为代表的云服务商在数据中心的每台服务器上部署了一块可编程网卡，用以加速虚拟网络[10]。为了在提供高性能的同时保持一定的可编程性和灵活性，业界提出了专用芯片（ASIC）、网络处理器（Network Processor）、多核通用处理器片上系统（SoC）、现场可编程门阵列（FPGA）等可编程网卡架构，将在 2.3 节中详细讨论。FPGA 在性能和灵活性间达到了折中，因此微软采用了基于 FPGA 的可编程网卡[48]。

FPGA 长期被用于实现网络路由器和交换机。然而，FPGA 通常使用 Verilog、VHDL 等硬件描述语言编程。众所周知，硬件描述语言难以调试、编写和修改，给软件人员使用 FPGA 带来了很大挑战。为了提高 FPGA 的开发效率，FPGA 厂商提供了高层次综合（HLS）工具[49-50]，可以把受限的 C 代码编译成硬件模块。但这些工具只是硬件开发工具链的补充，程序员仍然需要手动将从 C 语言生成的硬件模块插入到硬件描述语言的项目中，且 FPGA 与主机 CPU 之间的通信也需要自行处理。学术界和工业界提出了 Bluespec[51]、Lime[52] 和 Chisel[53] 等高效硬件开发语言[54-56]，但它们需要开发者具有足够的硬件设计知识。高层次综合工具和高效硬件开发语言可以提高硬件开发人员的工作效率，但仍然不足以让软件开发人员使用 FPGA。

近年来，为了让软件开发人员使用 FPGA，FPGA 厂商提出了基于 OpenCL 的编程工具链[57-58]，提供了类似 GPU 的编程模型。软件开发人员可以把用 OpenCL 语言编写的核（kernel）卸载到 FPGA 上。但是，这种方法中多个并行执行的核间需要通过板上共享内存进行通信，而 FPGA 上的 DRAM 共享内存吞吐量和延迟都不理想，共享内存还会成为通信瓶颈。其次，FPGA 与 CPU 之间的通信模型是类似 GPU 的批处理模型，这使得处理延迟较高（约 1 毫秒），不适用于需要微秒级延迟的网络数据包处理。本文第 4 章将提出一个让软件开发人员可用，且网络数据包处理性能高的模块化 FPGA 编程框架。在此基础上，第 5 章将利用可编程网卡，将 RDMA 的共享内存读写原语扩展到键值操作原语，用可编程网卡实现高吞吐量、低延迟的内存键值存储。

1.3　本文的研究内容和贡献

本文旨在探索基于可编程网卡的高性能数据中心系统。本文提出一个基于 FPGA 可编程网卡，对云计算数据中心计算、网络、存储节点实现全栈加速的系统。如图 1.3 所示，通过把计算、网络、存储节点上的普通网卡替换为可编程网卡，本文在计算节点上实现了虚拟网卡和虚拟网络，虚拟本地存储和云存储，以及轻量级用户态运行库和硬件传输协议相结合的通信原语，替代了图 1.1 中的软件虚拟化服务和操作系统网络协议栈。本文还基于数据面与控制面分离的思想，实现了网络节点的虚拟网络功能和存储节点的内存数据结构处理，用可编程网卡提高数据面性能，并保留原有软件控制面的灵活性。

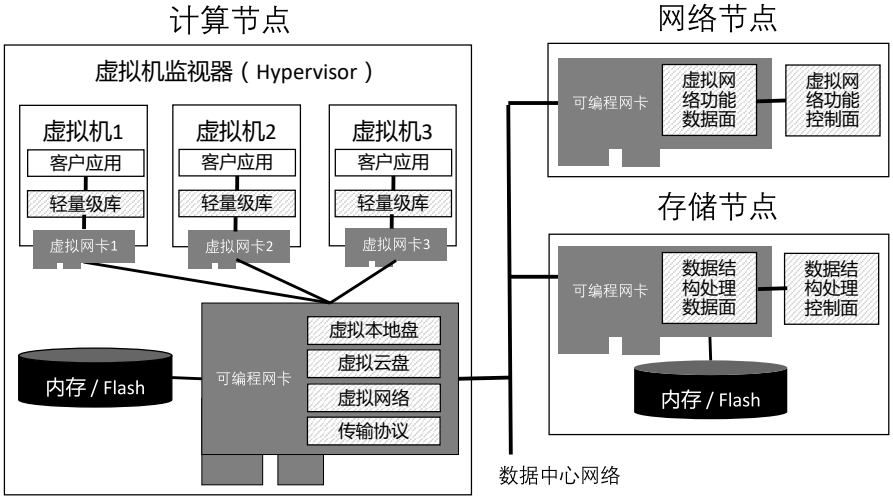


图 1.3　基于可编程网卡的数据中心系统总体架构。

首先，本文提出用可编程网卡加速云计算中的虚拟网络功能。提出了首个在商用服务器中用 FPGA 加速的高灵活性、高性能网络功能处理平台 ClickNP。众所周知，FPGA 编程对软件工程师很不友好。为了简化 FPGA 编程，设计了类 C 的 ClickNP 语言和模块化的编程模型，并开发了一系列优化技术，以充分利用 FPGA 的海量并行性。实现了 ClickNP 开发工具链，可以与多种商用高层次综合工具集成。基于 ClickNP 设计和实现了 200 多个网络元件，并用这些元件组建起多种网络功能。相比基于 CPU 的软件网络功能，ClickNP 的吞吐量提高了 10 倍，延迟降低到 1/10；且具有可忽略的 CPU 开销，可以为云计算中的每个计算节点节约 20% 的 CPU 核。

其次，本文提出用可编程网卡加速远程数据结构访问。键值存储是最常用的基本数据结构之一，也是很多数据中心内的关键分布式系统组件。基于 ClickNP 编程框架，设计实现了一个高性能内存键值存储系统 KV-Direct，在服务器端绕过 CPU，用可编程网卡通过 PCIe 直接访问主机内存。把单边 RDMA 的内存操作语义扩展到键值操作语义，解决了单边 RDMA 操作数据结构时通信和同步开销

高的问题。还利用 FPGA 可重配置的特性，允许用户实现更复杂的数据结构。面对网卡与主机内存之间 PCIe 带宽较低、延迟较高的性能挑战，通过哈希表、内存分配器、乱序执行引擎、负载均衡和缓存、向量操作等一系列性能优化，实现了 10 倍于 CPU 的能耗效率和微秒级的延迟，实现了首个单机性能达到 10 亿次每秒的通用键值存储系统。

最后，本文提出用可编程网卡和用户态运行库相结合的方法为应用程序提供系统原语，从而绕过操作系统内核。套接字是操作系统提供的最常用的通信原语。设计实现了一个用户态套接字系统 SocksDirect，与现有应用程序完全兼容，能实现接近硬件极限的吞吐量和延迟，多核性能具有可扩放性，并在高并发负载下保持高性能。主机内和主机间的通信分别使用共享内存和 RDMA 实现。为了支持高并发连接数，本文基于 KV-Direct 实现了一个 RDMA 可编程网卡。通过消除线程间同步、缓冲区管理、大数据拷贝、进程唤醒等一系列开销，SocksDirect 相比 Linux 提升了 7 至 20 倍吞吐量，降低延迟到 1/17 至 1/35，并将 Web 服务器的 HTTP 延迟降低到 1/5.5。

1.4　论文结构安排

本论文的内容结构安排如下：第 1 章为绪论。第 2 章介绍数据中心的发展趋势，指出数据中心软硬件结合优化的需求与机遇，讨论可编程网卡的架构，并调研可编程网卡在数据中心的应用。第 3 章提出基于可编程网卡的数据中心系统架构。第 4 章是网络功能加速部分，提出用可编程网卡加速云计算中的虚拟网络功能。为了简化 FPGA 编程，提出首个适用于高速网络数据包处理、基于高级语言的模块化 FPGA 编程框架 ClickNP。第 5 章是数据结构加速部分，提出用可编程网卡加速远程数据结构访问，并设计实现一个高性能内存键值存储系统 KV-Direct。第 6 章是操作系统加速部分，提出用可编程网卡和用户态运行库相结合的方法为应用程序提供操作系统原语，并设计实现一个用户态套接字系统 SocksDirect。第 7 章总结全文并展望未来研究方向。

第 2 章　数据中心与可编程网卡概论

本章介绍全文的背景和相关工作。图 2.1 概述了本章的逻辑结构。第 1 节从数据中心应用需求、硬件和运营模式出发，介绍数据中心的四个发展趋势，即资源虚拟化、分布式计算、定制化计算和细粒度计算。这些趋势催生了高性能数据中心网络和内存数据结构存储两大基础设施。第 2 节从虚拟网络、网络功能、操作系统、数据结构处理四方面分析数据中心的性能挑战，即所谓“数据中心税”。可编程网卡是用于降低“数据中心税”的定制化硬件。第 3 节比较基于专用芯片、网络处理器、通用处理器、FPGA 的四种可编程网卡架构。第 4 节调研可编程网卡在微软 Azure、亚马逊 AWS 等数据中心的部署。

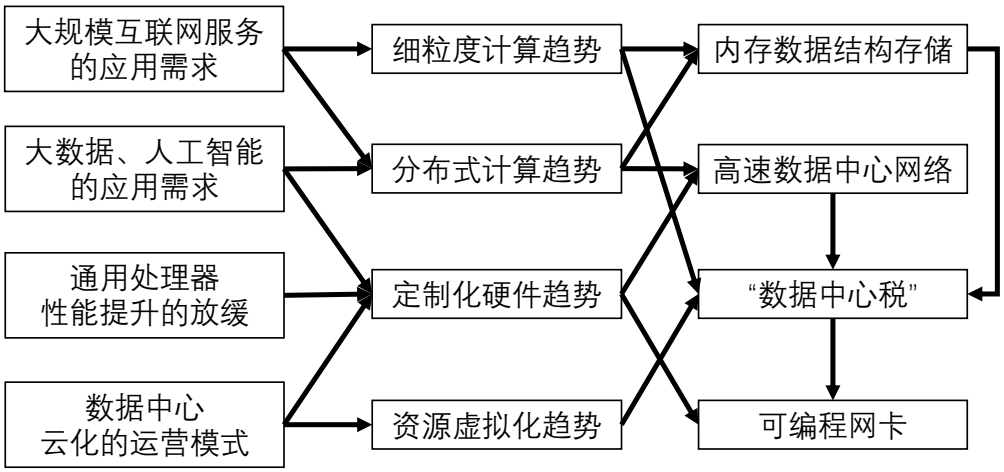


图 2.1　本章逻辑结构。

2.1　数据中心的发展趋势

数据中心（data center）的历史可以追溯到 20 世纪 40 年代早期计算机的机房。早期计算机是对环境要求很高的大型精密仪器，需要专人维护，而且运行的往往是保密的军事任务，因此需要严格管控的数据中心来保护。随着计算机技术的发展，越来越多的用户和业务需要使用计算机。由于早期计算机仍然非常昂贵，一家公司或研究机构往往只有少数几台计算机，用户通过终端（terminal）连接到数据中心内的计算机上，成为客户端-服务器（client-server）架构的雏形。

20 世纪 80 至 90 年代，硬件摩尔定律和操作系统软件驱动了个人计算机（PC）的蓬勃发展，其计算和存储能力较低，稳定性较差，但成本便宜。与此同时，从早期计算机演化而来的大型机（mainframe）、小型机仍然是企业的主流选择，其计算和存储能力较强，稳定性较高，但成本昂贵。这些昂贵的大型机也需要放在专用的数据中心内，由专人维护。

20 世纪末，随着互联网的迅速发展，由于互联网 “免费” 的商业模式和数据量、用户数量的迅速膨胀，传统的大型机、小型机成本过于昂贵，也不容易随快速扩张的业务而增加计算和存储资源。为此，越来越多的互联网公司开始使用与个人计算机（PC）结构类似的标准服务器来搭建互联网服务，组成分布式系统。由于标准服务器的数量多，数据中心的规模不断扩张。这些服务器需要巨大的机房空间，高速、稳定的互联网连接，稳定的温湿度，还需要较便宜的电力以降低能源成本，因此数据中心建设逐渐成为一个专门的行业，数据中心也成为一个规范的名词。

2010 年前后，互联网公司如雨后春笋般成长起来，需要高度可扩放（scalable）的计算、存储和网络资源。按需租用计算、存储、网络资源的云计算成为越来越流行的商业模式。越来越多的企业也把传统 IT 系统迁移到云计算平台上，以降低运维成本。相比互联网数据中心，云数据中心的应用类型和用户交互方式更丰富，网络互连性能更高、硬件更加定制化，也支持更好的资源复用。下面四节将分别介绍云数据中心的四个发展趋势：资源虚拟化、分布式计算、定制化计算和细粒度计算。

2.1.1　资源虚拟化

早期计算机和大型机成本昂贵。为了让多个任务充分利用计算和存储资源，虚拟化的概念应运而生。20 世纪 50 至 60 年代的分时系统[59-60] 实现了多个用户任务分时复用硬件，发展成为 70 年代 UNIX 等现代操作系统[61] 的前身。20 世纪 70 至 90 年代，虚拟机监视器（Virtual Machine Monitor，VMM；或称 hypervisor）进一步实现了多个操作系统分时复用硬件[62-63]，为云计算的发展做了技术准备。

21 世纪的第一个十年，互联网的发展让越来越多的企业需要提供 24 小时运行的网络服务，互联网数据中心（Internet Data Center，IDC）托管服务逐渐兴起。然而，IDC 托管需要客户事先购买服务器硬件，并需要运维人员维护，有较高的资产成本（capex）和运营成本（opex）。很多企业的网络服务一方面具有较高的季节性（如亚马逊的黑五促销），因此在闲时大量的计算资源被闲置；另一方面数据和用户规模的扩张很快，给购买硬件和 IDC 选址带来了时间压力。为此，虚拟机托管服务提供了按需租用的虚拟机资源，实现了服务器资源按 CPU 核的切片和在不同客户间的分时复用，也便于客户内部运维人员的管理和调度。

云计算是虚拟机托管服务的升级版，标志性的变革是计算和存储的解耦。虚拟主机托管服务把一台主机上的计算和存储资源分片成多个虚拟机，一旦主机的硬件或虚拟化软件（hypervisor）发生故障，虚拟机也就停机，其中的数据还有丢失的风险。在云计算中，虚拟机的存储资源在分布式存储系统中有多个副本，从而计算节点发生故障时可以从其他计算节点重启虚拟机，存储节点的故障则

一般对客户透明。计算和存储的解耦不仅大大提高了服务可用性和数据安全性，也方便了虚拟化软件升级和虚拟机热迁移。

除了与其他公司共享硬件资源，IT 企业使用云计算进行虚拟化还有一个目的，即复用硬件基础设施，为公司内不同类型的服务提供不同的服务质量保证。例如，响应用户请求的 Web 前端服务器、在线事务处理（OLTP）数据库、在线机器学习推理（inference）等通常需要较低的延迟；离线数据处理（OLAP）、数据挖掘、分布式机器学习训练等需要访问海量数据，进行大量计算，需要较高的吞吐量。低延迟和高吞吐量某种程度上是互相矛盾的 ①，因此需要将计算、网络、存储等资源切片（slicing），为不同需求的应用提供不同的服务质量保证（Quality of Service，QoS）。

如第 1.1 节介绍过的，云数据中心的客户虚拟机位于计算节点，而存储服务和网络服务运行于解耦的存储和网络节点上。此外，还需要管理节点进行调度和监控。如图 2.2 所示，数据中心通常由计算、网络、存储、管理等节点以及节点之间的互连网络构成。

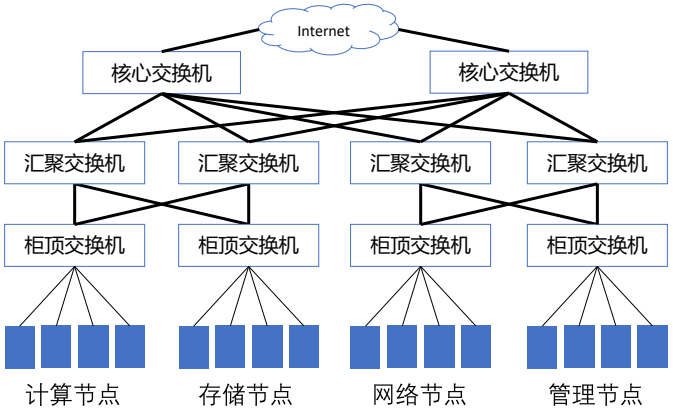


图 2.2　数据中心架构。

2.1.2　分布式计算

20 世纪末，连接信息孤岛的搜索引擎拉开了互联网时代的序幕。搜索引擎不仅需要搜集、处理和索引海量信息，还需要实时响应大量用户的信息检索请求。传统的大型机和企业级存储不仅成本高昂，也无法满足海量信息存储和用户请求处理的可扩放性 ②。为此，Google 提出用普通商用服务器组建可扩放的数据中心，用软件来实现存储的分区和冗余、用户请求的分派，在相对不可靠的硬件基础上组建起容错的系统[64-66]，引领了大规模分布式计算的浪潮。信息检索

①延迟指处理结束与处理开始的时间差。吞吐量指单位时间内处理请求的数量。延迟和吞吐量是系统性能的两个重要指标。大致上，吞吐量等于并行处理的请求数量除以平均延迟。

②分布式系统的可扩放性（scalability）指系统的吞吐量随节点数量增加。理想的可扩放性是吞吐量随节点数量线性增加。

和 Web 服务等工作负载相对容易并行化。每个用户请求之间几乎没有关联，因此可以被分派到不同的服务器上并行处理，增加服务器数量几乎可以获得系统吞吐量的线性提升。在有合适索引的情况下，处理每个用户请求所需访问的数据量也是较少的，对通信性能没有很高的要求。

随着搜索广告、社交网络、移动互联网的发展，互联网公司积累了海量的用户数据。为了从海量数据中挖掘出知识，大数据开始兴起，催生了以 Hadoop[67]、Spark[68] 为代表的大数据处理框架。例如，MapReduce[66] 从函数式编程语言中借鉴了映射（map）和归约（reduce）的概念，提出了在不可靠硬件集群上进行大规模数据集并行处理的编程框架。大数据处理通常是批量（batch）操作，需要访问海量数据，不容易并行处理并获得线性加速。难以并行的根本原因是节点间的通信开销。例如，图计算（graph computing）中经典的 PageRank 算法[69] 由若干阶段组成。每个阶段，每个点需要根据其邻接点的权重来更新自己的权重。在分布式计算中，每个节点处理一部分的点集，因此每个阶段都需要节点间进行大量的通信。

由于 MapReduce 的中间结果存储在磁盘上，I/O 开销很高，Spark[68] 大数据处理框架提出在内存中维护计算的中间状态。在内存中计算（in-memory computing）成为大数据处理的新范式[28,70]。消除了磁盘的瓶颈后，数据中心网络的延迟和吞吐量就成为分布式系统的新瓶颈。这推动了数据中心网络近十年来从 1 Gbps 到 40 至 100 Gbps 的性能飞跃[71]（如图 2.3），以及以 RDMA 为代表的高性能数据中心网络传输技术的大规模部署[41]。

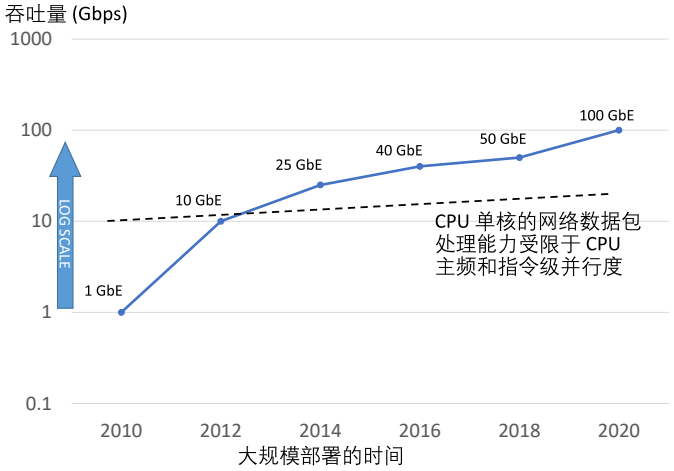


图 2.3　数据中心网络性能的快速提升。

近年来，基于 RDMA 的内存中计算使很多分布式系统的性能实现了飞跃，这些系统包括键值存储[70]、分布式事务[72-74]、远程过程调用[20]、图计算[75] 等。有了高性能的网络硬件并不意味着分布式系统通信性能的提高。本文第 2.2 节将讨论，传统操作系统的 “数据中心税” 使分布式应用难以充分利用数据中心网络

硬件的高性能。

分布式系统中的进程间通信有消息传递和共享内存两种范式[76]。在消息传递范式中，进程把需要发送的数据结构序列化成一个字符串，通过网络消息发送给另一个进程。分布式系统中的消息传递通常采用远程过程调用（RPC）或消息队列（message queue）模型，或者两者的结合。在 RPC 模型中，服务器端注册一个过程（procedure）来响应客户端的 RPC 请求。在消息队列模型中，生产者将消息广播或者分发到若干消费者。为了实现生产者与消费者的解耦以及消息的缓冲和可靠投递，消息队列模型往往引入一个经纪人（broker）服务，如 Kafka[77]。在编程接口方面，分布式应用程序通常使用 RPC 库和消息队列中间件（middleware），这些库和中间件则依赖操作系统的套接字（socket）接口来发送和接收消息。

在共享内存范式中，多个进程共享内存空间，一个进程写入的内容能被所有进程读取。远程直接内存访问（RDMA）技术旨在为分布式系统提供共享内存的抽象，近年来在越来越多的数据中心部署，也是系统学术界的研究热点。但是，在很多情况下，共享内存的抽象层次太低，应用程序需要自行管理内存的分配释放和对象在内存中的布局，多个进程同时写入时还需要进程间同步。而传统的关系型数据库抽象层次又太高，过于重量级，性能受限。共享数据结构的抽象层次介于共享内存和关系型数据库之间，可以高效、灵活地存储结构化和半结构化的数据。共享数据结构存储的典型例子是 Redis[78]，其基本抽象是一张键-值映射表，用户可以指定某个键（key），获取或修改对应的值（value）。其中的值可以是简单的字符串，也可以是更复杂的数据结构，如列表、集合、优先队列、字典等，Redis 也提供了这些数据结构的操作原语。由于键-值映射是常用的基本数据结构，很多数据结构存储又被称为键值存储（key-value storage）。共享数据结构存储可以常驻内存，也可以持久化到磁盘或闪存。

内存共享数据结构存储是越来越多分布式应用的选择。例如，Spark[68] 大数据处理框架以容错分布式数据集（RDD）作为基本抽象，把大数据处理抽象为一个数据流图，图中的每个节点从 RDD 中读取数据，进行变换并把结果输出到 RDD。由于 RDD 的存储可以实现高可用性和可扩放性，用 RDD 抽象编写的大数据处理程序也就具备容错性 ① 和可扩放性。

消息传递和共享数据结构两种进程间通信范式在不同的场景下各有优劣，它们对分布式系统的性能都很重要。两种通信范式不仅在理论上可以互相转化[79]，在实现中也经常转化到另一种范式。例如，RDMA 共享内存和分布式数据结构存储通过网络消息发送读写请求和数据；FaSST[20] 通过 RDMA 共享内存读写原

① 分布式系统中的容错性（resilience）指节点发生不可预期的硬件故障或操作系统崩溃时，分布式系统能利用其余的节点自动恢复，继续运行。

语实现高性能 RPC；Redis[78] 键值存储系统可以当作轻量级消息队列服务使用。本文第 4 章和第 6 章致力于提升消息传递的性能。本文第 5 章致力于提高内存共享数据结构存储的性能。

自 2012 年起，神经网络开始复兴，深度学习成为机器学习的新范式，也使人工智能进入了持续至今的又一次高潮。众所周知，数据和算力是推动深度学习复兴的两大动力，其中算力主要是不同于 CPU 的定制化硬件 GPU 提供的。深度学习的发展需要越来越高的算力，反过来推动了 GPU 和各种深度学习处理器的发展。由于单机的算力不足以在大数据上训练大模型，分布式机器学习训练成为主流。分布式机器学习训练通常采用随机梯度下降（SGD）方法。SGD 由若干阶段（epoch）构成，每一阶段开始时，各个计算节点之间共享一个模型，并使用不同的训练数据分别计算出模型的梯度，然后把各个计算节点的模型梯度汇总起来，修改模型，作为下一阶段各个计算节点的共享模型。分布式机器学习训练主要有迭代式 MapReduce（IMR）、参数服务器（parameter server）和数据流（data flow）三种架构[80]。迭代式 MapReduce 使用大数据处理平台的基础架构，适用于数据并行 ① 和同步通信。

同步通信的问题是整个系统的性能会被最慢的节点所拖慢，而且只要有一个节点发生故障，整个系统就无法继续运行。因此，近年来异步随机梯度下降和半同步随机梯度下降训练方法开始流行。在异步通信方法中，每个节点仍然在本地的数据上进行训练，但不必等待其他节点训练完成，就把本地的模型梯度更新分发到所有其他节点上。为了便于分发模型和汇总模型的梯度，分布式机器学习训练系统常常配备一个分布式的参数服务器（parameter server）[81]。各个工作节点从参数服务器获取当前参数，并将本地的梯度更新上传到参数服务器。分布式的参数服务器不仅平衡了各个参数存储节点的负载，还能在工作节点仅访问部分参数时降低通信开销。

参数服务器架构不仅可以支持数据并行，还可以支持模型并行 ②。例如，DistBelief[82] 同时使用数据并行和模型并行；AlexNet[83] 利用了卷积神经网络各层间计算的独立性，使用了模型并行。参数服务器是键-值存储的典型应用。以微软的 Multiverso 参数服务器[84] 为例，参数可能是向量、矩阵、张量，也可能是哈希表，或者稀疏形式的矩阵。目前大型互联网公司的分布式机器学习系统多采用参数服务器架构。

由于数据与模型混合并行的分布式机器学习训练方法越来越流行，近年来

① 机器学习中的数据并行指的是不同的节点使用不同的数据进行训练。注意与后文中 FPGA 内部的数据并行不同。

② 机器学习中的模型并行指的是多个节点分别计算模型的一部分，即一个训练数据的梯度需要经过多个节点才能计算出来。

出现了基于数据流的分布式深度学习系统，如 Tensorflow[85]。在数据流系统中，数据流图中的每个节点代表一个数据处理算子，是一个有限状态自动机。节点之间用控制消息流和数据流相连，采用消息传递的分布式系统通信范式。事实上，数据流图中的节点也可以是参数服务器，这就转化成了共享数据结构的分布式通信范式。

2.1.3　定制化硬件

在大型机时代，需要处理大量信息的大型机乃至超级计算机一般采用软硬件一体化系统，即软硬件是由同一个公司团队开发的。由于采用了高速硬件互连、硬件冗余，这些系统往往同时具备高性能和高可靠性，但成本随系统规模的扩大急剧增加。

如第 2.1.2 节所述，20 世纪末以来，互联网数据中心通常由标准的商用服务器构成。这些普通商用服务器之所以成本较低，是因为它的架构与大量商用的个人计算机（PC）类似，CPU、内存、主板、硬盘、网卡等组成部分都是标准组件，由各个专业公司独立设计实现。操作系统、数据库、Web 服务器等软件也是标准化的，或者由专业公司开发，或者是开源软件。产量大的标准组件能更好地平摊研发、流片等一次性工程费用（Non-Recurring Engineering，NRE），从而降低标准组件的价格。由标准组件构成的系统虽然降低了数据中心的软硬件成本，但也给标准组件的开发者施加了限制：大家需要遵守标准组件之间的接口和协议，只能在自己的边界内创新。数据中心系统的搭建者则只能像搭积木一样组合标准组件，而很难通盘考虑、全局优化。

2010 年以来，云计算规模化的趋势、数据中心应用的需求以及通用处理器的性能局限使定制化计算成为数据中心的趋势，工程师重新获得了软硬件协同设计的机会。首先，在云计算平台中，软件和硬件的环境都由服务商控制，在达到了一定的规模以后，各种形式的定制化都成为可能。只要能够提高性能，降低价格，增强竞争力，云服务商有足够的动力去定制芯片，改变网络协议，改变服务器架构，更改操作系统，甚至重新编写应用程序。其次，第 2.1.2 节已经提到，大数据和机器学习等应用对算力的需求很高。最后，通用处理器的性能局限是定制化硬件最主要的推动力，下面将详细讨论。

摩尔定律预言，可以通过把信息的存储和处理单元做得越来越小来提升单位面积集成电路的存储和处理单元数量，从而提升单位面积集成电路的性能。更为深刻的是 Dennard 缩放定律[86]，即集成电路的性能在不消耗更多能源和面积的情况下能够每两年翻倍。它的理论基础是每两年采用一代新的半导体工艺，晶体管尺寸缩小 30%，从而芯片面积缩小 50%。为了保持电场的恒定，电压随晶体管尺寸同比例降低了 30%。与此同时，由于芯片尺寸缩小了，延迟降低了 30%，

时钟频率就可以提升 40%[2,87]。在那个年代，集成电路的动态功耗占了功耗的主要部分，其与电容、电压的平方和频率成正比，从而可以计算出功耗降低了 50%。按照这个理想模型，每两年集成电路的面积和功耗都减半，就可以在原有的面积和功耗下塞进两倍数量的晶体管，而且时钟频率还提高到了 1.4 倍。对于冯 · 诺伊曼体系结构的单线程微处理器，这些增加的晶体管主要用于更大的缓存、更复杂的流水线、超标量、乱序执行、寄存器重命名、分支预测等，以提高每时钟周期所能执行的指令数。根据 Pollard 经验定律[88]，每时钟周期的计算能力大约与晶体管数量的平方根成正比。单位时间的算力等于时钟频率乘以每时钟周期的算力，因此每两年微处理器的性能提升到 2 倍，还不消耗更多的能源和面积。

不幸的是，进入 21 世纪以来，摩尔定律和 Dennard 缩放定律的红利正在逐渐消失。首先，随着集成电路特征尺寸的缩小，电压也随之降低。但控制晶体管的阈值电压越低，晶体管的漏电流就会迅速增长，成为集成电路功耗的重要组成部分。为了控制漏电流，阈值电压不仅不能降低，甚至需要比前一代集成电路有所提高[87]。因此，每一代新的半导体工艺，每晶体管的功耗不会像预期的那样降低一半。其次，由于每晶体管的面积缩小一半，功耗却没有降低这么多，单位面积集成电路的功耗就会升高。芯片的散热问题成为制约集成电路规模的主要因素[2]。再次，对于同一个集成电路，在允许的范围内，为了提高一倍的性能而把时钟频率提高一倍，电压就要相应提高一倍来降低晶体管的翻转延迟，因此功耗大致与时钟频率的三次方成正比。由于散热的限制，集成电路的时钟频率也受到限制，靠 “超频” 来大幅提升集成电路性能是不现实的。最后，目前 7 nm 半导体工艺已经量产，而硅原子的半径为 0.1 nm。随着集成电路的特征尺寸越来越接近原子尺寸，量子效应不可忽略，给光刻技术带来了很大的技术挑战[2]。事实上，约 2010 年以来，集成电路特征尺寸的缩小已经明显放缓，不再能保持两年一代的速度。综上，在当前的半导体技术框架下，单位面积集成电路的性能已经不再能维持两年提升一倍的速度，而且性能的提升也意味着功耗的提升，“免费的午餐” 结束了。

然而，从芯片的体系结构角度看，传统的冯 · 诺伊曼体系结构并不能充分利用每个晶体管的计算能力。因此，还有机会 “从摩尔定律这个柠檬里又榨出这么多汁来”[89]。理论上，每时钟周期的算力可以与晶体管数量成正比，但前述的 Pollard 经验定律[88] 指出，冯 · 诺伊曼微处理器每时钟周期的算力经验上与晶体管数量的平方根成正比。例如，1971 年的全球第一款微处理器 Intel 4004 使用 10 微米制程，有 2300 个晶体管，时钟频率 108 KHz，每秒能执行 90 K 次 4 位运算。2016 年 Intel 基于 Broadwell 架构的 Xeon E5 微处理器使用 14 纳米制程（4004 的 1 M 倍），有 72 亿个晶体管（4004 的 3 M 倍），基频 2.2 GHz（4004 的 20 K 倍），

每秒能执行约 300 G 次 64 位运算[①]（4004 的 3 M 倍）[90]。可以看到，Xeon E5 每时钟周期能执行的运算数是 4004 的约 150 倍，而晶体管数是 300 万倍。即使考虑到 64 位计算比 4 位计算复杂的因素，仍然意味着 4004 每个晶体管对算力的贡献比 Xeon E5 高数百倍。这是由于冯 · 诺伊曼结构微处理器的指令集和微体系结构越来越复杂，一是为了提高单线程性能，二是为了添加更深的缓存层次来解决“内存墙”问题，三是为了支持多核间的通信与同步、操作系统和虚拟化技术。真正用于计算的晶体管占比越来越少了。

由于冯 · 诺伊曼结构处理器的性能瓶颈，定制化硬件成为趋势。定制化硬件的基本操作不需要通过指令表达，数据操作流程也相对固定，因此不需要冯 · 诺伊曼结构与指令译码执行、流水线控制有关的开销。定制化硬件可以定制数据通路和内存层次，避免冯 · 诺伊曼结构所有内存地址共享访问的“内存墙”问题。定制化硬件可以构建大量处理单元并行处理相同类型的数据（如矩阵运算），或者很深的流水线来处理深逻辑层次的计算（如对称加密）。如图 2.4 所示，K80、P100、P40、V100 等英伟达 GPU，Arria 10、Stratix 10 等英特尔 FPGA，以及谷歌深度学习处理器 TPU 的能耗效率远远高于通用处理器（注意 y 轴是对数坐标系），且基本上遵循摩尔定律在性能方面的预言，即定制硬件的能耗效率每 18 个月提高一倍。

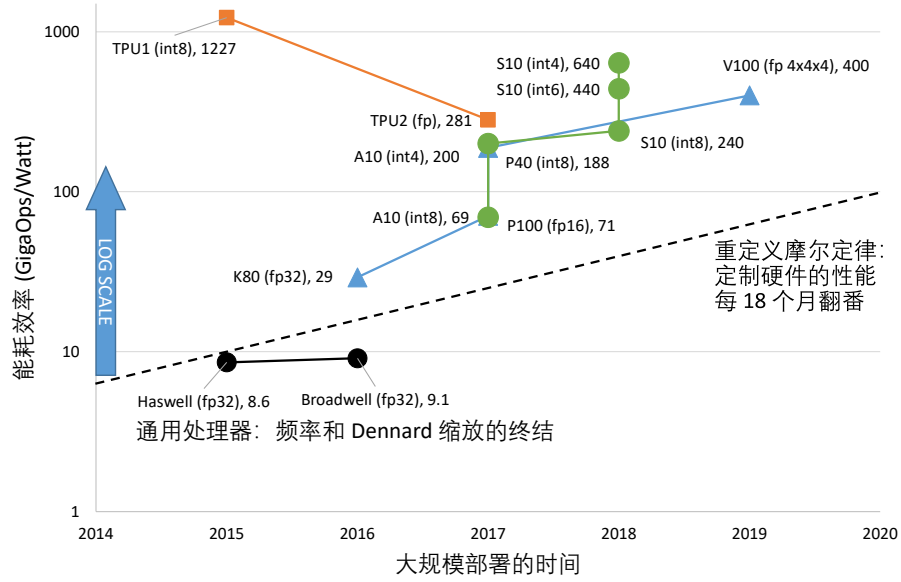


图 2.4　通用处理器的频率和 Dennard 缩放逐渐终结，但定制化硬件重新定义并延续了摩尔定律。

[①]假设应用使用 AVX2 指令，不使用 FMA3 指令，没有超频。

2.1.4　细粒度计算

2013 年 Docker 框架提出以来，大量互联网公司使用容器（container）部署服务[91]。容器不仅是轻量级的虚拟机，更重要的是重新定义了互联网服务的架构，从少量复杂的宏服务（macroservice）解耦合成大量简单的微服务（microservice）。每个微服务使用容器的形式部署，可以实现高效的可扩放数据中心调度、软件依赖管理和微服务间的隔离，提高开发、测试和运维的效率。微服务架构相比传统架构的计算粒度更细，因而对总请求服务能力的需求更高，对微服务间通信的需求也很高。例如，微信有超过 3000 个微服务运行在超过 2 万台服务器上。入口层的微服务每天响应 100 亿至 1000 亿次用户请求，而每个用户请求会在系统内部触发更多的微服务请求，因此整个微信后端每秒需要响应数以亿计的微服务请求[92]。

容器是比虚拟机更细粒度的计算范式。一台服务器主机上可以部署数百个容器，然而一般只能部署数十个虚拟机。第一，需要支持更多的容器给数据中心虚拟化带来了挑战。计算虚拟化方面，传统的虚拟机一般是把 CPU 核心直接分配给虚拟机；而容器的数量一般大于 CPU 核心的数量，从而需要操作系统进行调度，让各个容器分时复用 CPU 核心，增加了调度开销，也提高了性能隔离和服务质量保证的难度。网络和存储虚拟化方面，容器比虚拟机的数量大一个数量级，给查找表容量、队列数量、缓存容量等带来了压力。

第二，将宏服务划分为微服务后，容器之间的通信增加了。很多本来是同一个虚拟机内部的通信变成了容器之间的通信，给服务器内的容器网络带来了压力。如果要使微服务的性能不明显下降，就需要使服务器内容器网络的性能与共享内存通信的性能接近。

第三，为了提高微服务的容错性和可扩放性，无状态的微服务设计逐渐成为趋势，即容器本身不存储状态，输入数据、输出数据、容器的配置和内部状态都存储在容器外的数据结构存储服务中。这要求数据结构存储具有低延迟、高吞吐量、高可用性。

基于容器的微服务并不是计算粒度缩小的终点。容器内仍然运行着传统操作系统的进程，这些进程即使没有外部请求，也在消耗一定的内存资源。为了保证容器能及时响应用户随时可能到来的请求，CPU 等计算资源也需要预留。容器也需要占据一定的存储空间。因此，云服务商需要为每个容器预留计算、内存和存储资源，并为这些预留的资源收费。容器的扩放、调度和运维也需要容器的使用者负责，虽然有 Kubernetes[93] 等开源框架，但也增加了容器使用者的负担。2015 年，亚马逊 AWS 推出了名为 Lambda 的无服务器计算（serverless computing）服务，使得用户只需在云服务商的编程框架内编写事件驱动的业务代码，而把执

行环境、扩放、调度等任务都留给云服务商。目前，亚马逊、微软、谷歌、阿里、腾讯等主流云服务商都提供了无服务器计算服务。

2019 年，加州大学伯克利分校对无服务器计算的预测报告[94] 表明，尽管无服务器计算的范式有诸多优势，也是各大云服务商争相推广的新服务，但由于云存储的性能问题和临时存储服务的缺失，很多类型的应用使用无服务器计算的性能和成本不理想。事实上，无服务器计算并不是一个新概念。目前的大数据处理框架（如 Spark[68]）和云计算厂商的数据湖（data lake）服务[95] 中，为了容错性和可扩放性，数据处理函数一般是无状态的，已经在广泛采用无服务器计算的范式。传统机器学习框架的训练和推理往往是分离的，而在强化学习（reinforcement learning）中，智能体需要不断与周围的环境交互，训练和推理是不断循环进行的，因此需要细粒度的计算。分布式强化学习框架 Ray[96] 也采用无状态数据处理函数的编程模型，通过键-值存储来保存数据流处理过程的中间状态。

综上，细粒度计算对数据中心虚拟化、容器网络、数据结构存储服务和操作系统调度的性能提出了挑战。

2.2 “数据中心税”

数据中心分布式计算的趋势催生了高性能数据中心网络，也对内存数据结构存储的性能提出了高要求。定制化硬件之间的通信需求也催生了高性能数据中心互连，推动了网络与互连的融合。在高性能数据中心网络中实现资源虚拟化带来很大的开销。除了资源虚拟化，传统操作系统中的网络协议栈性能也不令人满意，很大程度上浪费了数据中心网络的低延迟和高吞吐量[6]。细粒度计算的趋势加剧了虚拟化和操作系统的性能压力，还需要高性能的内存数据结构存储。本文将数据中心除执行用户应用程序外的成本统称为 “数据中心税”[4]，包括计算节点上的操作系统和虚拟机监控器、网络节点和存储节点，如图 2.5 中阴影背景的方框所示。下文将分四个小节，分别讨论虚拟网络、网络功能、操作系统、数据结构处理带来的 “数据中心税”。

2.2.1　虚拟网络

公有云中的大客户不再只是需要虚拟机，而是需要模拟企业网络的架构。为了支持安全组、访问控制列表等网络安全功能，以及在 Internet 中隐藏内部网络结构、减小攻击面，公有云服务提供了虚拟网络（Virtual Private Cloud，VPC）服务，即在公有云中划分出一个逻辑隔离的部分，提供丰富的网络语义，例如具有客户提供的地址空间的私有虚拟网络，安全组和访问控制列表（ACL），虚拟路

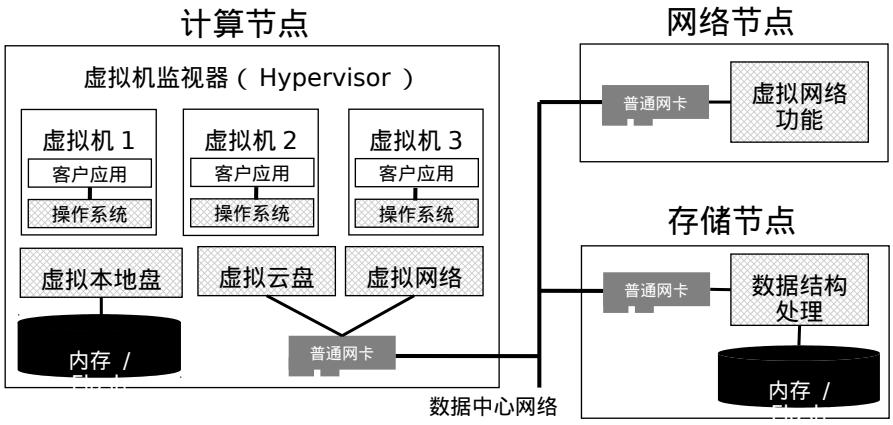


图 2.5　虚拟化的数据中心架构。阴影背景的方框是数据中心在执行客户应用之外的成本，即“数据中心税”。

由表，带宽计量，服务质量保证（QoS）等。为此，公有云中的网络被虚拟化了，虚拟网络和物理网络实现了解耦。如图 2.6 所示，两个计算节点上的客户虚拟机之间通信时，需要经过虚拟网络软件的封装和解封装，还可能需要经过网络节点上的若干网络功能的处理。网络功能将是下一节讨论的主题。

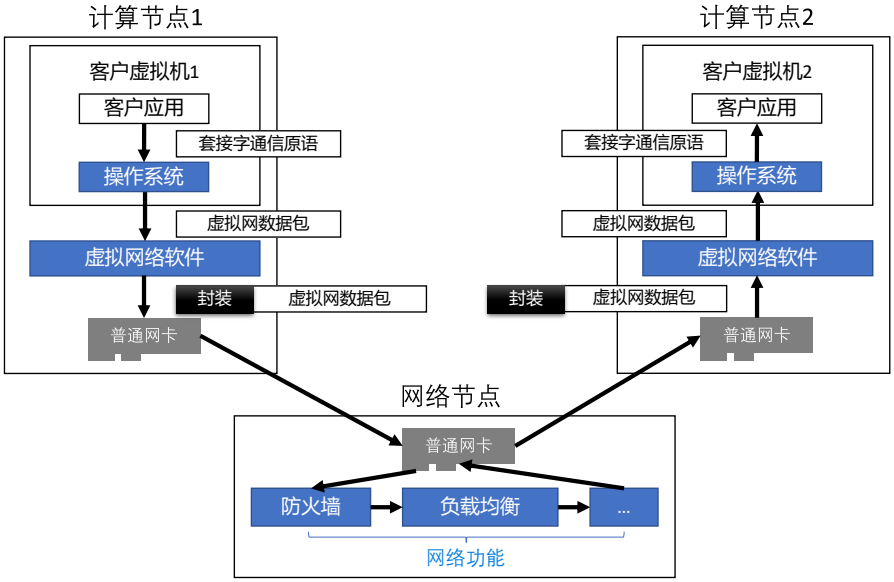


图 2.6　数据中心虚拟网络架构。

虚拟网络的数据面可以用匹配-操作表（Match-Action Table）来描述，理论上可以在商用网络交换机上实现。2007 年，斯坦福大学提出的 OpenFlow[97] 统一了不同厂商交换机的控制面接口，从而可以用软件来对网络进行编程，即软件定义网络（Software Defined Networking，SDN）。为了支持软件定义网络的控制面，Onix[98] 提出了一个大规模交换机的控制框架，Frenetic 等编程语言[99-100] 提出用函数式响应型编程（Functional Reactive Programming，FRP）的范式简化控制面事件处理，Covisor[101] 实现了控制面的虚拟化。随着交换机的可编程性越来越高，用软件自顶向下地定义交换机的数据面转发行为成为可能，而不是像

OpenFlow 那样自下而上地适配交换机的固定功能。

为此，2013 年，斯坦福大学提出了 P4[102]，提供可编程的数据包解析、有状态的匹配-操作流水线等编程抽象。学术界提出了 P4 语言在可编程交换机芯片[103]、可编程网卡[104]、FPGA[105]、CPU 虚拟交换机[106] 上的多种实现。工业界的 Barefoot Tofino 交换机芯片[107]、Mellanox Connect-X 网卡[108]、基于 FPGA 的 Xilinx SDNet 网络处理器[109] 等也支持了 P4 语言。

然而，基于网络交换机的虚拟网络在云数据中心中有两个根本挑战。首先，实际云数据中心虚拟网络的语义非常复杂，而且变化太频繁，以至于固定功能的传统交换机硬件更新的速度很难匹配需求变更的速度。其次，一台柜顶交换机连接了数十台机架服务器，一台机架服务器又可以虚拟化出数十台虚拟机，因此交换机需要支持至多上千台虚拟机的数据面封装和转发规则，现有商用交换机芯片的查找表容量不足。为此，微软提出了类似 P4 的 VFP 编程抽象[110] 以支持基于主机的软件定义网络，在虚拟交换机软件中实现虚拟网络。基于主机的虚拟网络可以很好地随计算节点服务器的数量扩放，并保持了物理网络的简单性。

在这种基于虚拟交换机的网络虚拟化模型中，虚拟机发送和接收的每个数据包都由虚拟机监控器中的虚拟交换机（vSwitch）处理。接收数据包通常涉及虚拟交换机软件将每个数据包复制到虚拟机可见的缓冲区，模拟虚拟机的软中断，然后让虚拟机的操作系统网络协议栈继续进行网络处理。发送数据包类似，但顺序相反。与非虚拟化环境相比，这种额外的主机处理会降低性能，需要对权限级别进行额外更改，降低吞吐量，增加平均延迟和延迟变化，并提高主机 CPU 利用率。

如图 2.3 所示，数据中心网络性能的提升速度远远超过通用处理器，在 10 Gbps 网络中只需要一个 CPU 核，而在现在的 40 Gbps 网络中就需要 5 个左右的 CPU 核，而在未来的 100 Gbps 网络中甚至需要 12 个 CPU 核。这就带来了“数据中心税”。

2.2.2　网络功能

除了虚拟网络，数据中心还需要多种网络功能。例如，大型互联网服务有多台前端服务器并发处理用户请求，这就需要一个高可用、高性能的负载均衡器接受用户的请求，并分发到前端服务器上。企业级负载均衡器还可能支持一系列高级功能，例如基于用户请求的灵活负载均衡规则，HTTPS 安全连接，日志记录和统计，检测并过滤可能的拒绝服务（DoS）攻击、Web 应用注入攻击等[7]。

再如，很多政府和企业已经有自己的 IT 信息系统，如果全部迁移到公有云上，就不符合一些机构对数据隐私和安全的要求，一次性迁移的成本和风险也过高。因此，连接客户已有的 IT 信息系统（on-premises）和公有云上的虚拟网

络就是很必要的。此外，针对客户对数据安全和隐私的顾虑，很多云厂商提供了私有云（或称专有云）模式，也就是把公有云的软硬件架构部署到客户专属的数据中心基础设施上。为了支持办公场所（on-premises）、私有云和公有云之间的连接，云厂商需要提供虚拟专线服务，这就需要能实现加密、路由、访问控制列表等基础网络功能以及缓存、TCP 加速等高级网络功能的专线网关。在一些情况下，这些云之间和云与办公场所之间的连接不是通过 SDH 或 MPLS 专线，而是通过 Internet 公共互联网。此时就需要用 IPSec 等协议对数据进行加密和签名，使用的是 IPSec 网关[111]。

此外，运营商网络的数据中心也运行着大量的网络功能。例如，AT&T 在美国运行了超过 5000 个中心局（Central Office），每个中心局支持数万个用户，运行着宽带网络网关（BNG）和 LTE 网络中的 EPC（Evolved Packet Core）网关[112]。运营商不仅希望降低中心局的资产和运营成本，还希望把中心局的边缘计算资源出租给第三方使用[112]。

传统上，这些网络功能使用专用的设备实现，例如 F5 公司的负载均衡器[113]、分布式拒绝服务攻击（DDoS）保护、Web 应用防火墙（WAF）等；思科核心路由器或 F5 BIG-IP 网关等提供的 IPSec 网关功能；运营商则使用华为、爱立信等公司的核心网设备。然而，这些专用硬件设备价格高昂，而且灵活性不足。公有云和 5G 网络需要灵活的网络功能以支持客户之间的安全隔离和性能隔离，满足不同客户的服务质量保证（QoS）。数据中心网络和 5G 核心网都需要在同一张物理网络上支持多种不同需求的服务。如 5G 的三种典型应用场景：极高带宽（eMBB）、极大规模（MTC）、极低和极稳定延迟（URLLC）。高带宽、大规模、低延迟等需求某种程度上是互相矛盾的，需要根据应用的需求进行权衡。为了高灵活性，亚马逊、微软、谷歌等数据中心都采用虚拟化的软件网络功能来取代专用设备。3GPP 标准也指明了 5G 核心网采用基于服务的系统架构，这些服务需要使用虚拟化的网络功能来实现[114-115]。

通过负载均衡器、专线 / IPsec 网关和运营商网络这几个例子，可以看到，网络功能的复杂程度明显高于虚拟网络。虚拟网络运行在网络层、数据链路层，一般只需要处理数据包头部信息，且不需要维护复杂的可变状态；而网络功能覆盖了应用层、传输层、网络层等，需要处理数据包的有效载荷（payload），且需要根据数据包查找到所属网络连接，根据连接的当前状态做出处理，再更新连接的状态。P4[103] 的编程灵活性不足以实现灵活的数据包有效载荷处理和基于连接的有状态处理。

2000 年，麻省理工学院的 Eddie Kohler 教授提出了 Click[116]，一个模块化的路由器编程框架。Click 将网络处理分解为若干基本元件（element），每个元件使用一个 C++ 的类实现；路由器的数据包处理是这些元件组成的数据流图，

Click 编程语言允许这些元件之间的灵活互连。由于 Click 开源项目中已经有大量元件，很多网络研究者只需将这些元件互连，即可组装出一个复杂的网络功能；由于 Click 使用 C++ 语言编程，其灵活性很高。因此，近年来的一系列研究工作[117-118] 基于 Click 编程框架实现网络功能。本文第 4 章将提出一个在 FPGA 上实现 Click 编程框架的高性能网络功能处理平台。

2.2.3　操作系统

操作系统主要包括三方面的功能：资源虚拟化、进程间通信和高层次抽象。资源虚拟化，即操作系统中的多个进程共享计算、网络和存储资源，需要保证进程间的安全隔离和性能隔离。进程间通信包括消息传递、共享内存和锁、信号量等同步原语。高层次抽象是把硬件资源抽象成应用程序容易使用的统一接口，例如 Linux 的“一切皆文件”模型。网络被抽象成顺序读写的套接字（socket）连接，存储被抽象成文件系统。本文关注操作系统在网络方面的功能，即网络资源在多进程间的共享、基于消息传递的进程间通信和套接字抽象。

分布式应用程序普遍使用操作系统中的套接字（socket）原语进行通信。如图 2.7 所示，对于 HTTP 负载均衡器、DNS 服务器、Memcached[26] 和 Redis[78] 键-值存储服务器等通信密集型的应用程序，操作系统占用了 50% 至 90% 的 CPU 时间，大部分用来处理套接字操作。

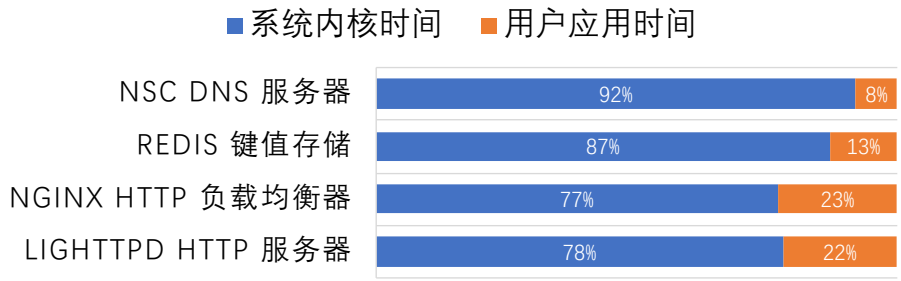


图 2.7　通信密集型应用程序在操作系统内核中消耗了大量的 CPU 时间。

在 Linux 操作系统中，应用程序通过文件描述符（File Descriptor，FD）进行 I/O 操作。从概念上讲，Linux 网络协议栈由四层组成。首先，虚拟文件系统（VFS）层为应用程序提供基于文件描述符的套接字 API。其次，在传输层，传统的 TCP 传输协议提供 I / O 复用，拥塞控制，丢包恢复等功能。第三，在网络和链路层，IP 协议提供了路由功能，Ethernet 提供了数据链路层的流控和物理信道复用，Linux 还实现了服务质量保证（QoS）和基于 netfilter 框架的防火墙等。第四，在设备驱动层，网卡驱动程序与网卡硬件（或用于主机内套接字的虚拟环回（loopback）接口）通信以发送和接收数据包。

众所周知，虚拟文件系统层贡献了网络 I/O 操作的很大一部分开销[119-120]。在本文第 6 章中，将通过一个简单的实验来验证：主机中两个进程之间的 Linux

TCP 套接字的延迟和吞吐量只比 Linux 管道（pipe）、FIFO 和 Unix 域（domain）套接字稍差。管道，FIFO 和 UNIX 域套接字绕过了传输层和网卡层，但它们的性能仍然不尽如人意。

以 Linux 网络协议栈为例，其开销是多方面的。对每次 I/O 操作，都需要进行内核穿越（kernel crossing），还需要获取文件描述符锁以保护多线程并发操作。每发送和接收一个数据包，都涉及到传输协议、缓冲管理、I/O 多路复用、中断处理、进程唤醒等一系列操作系统开销。每发送和接收一个字节，其内存都要复制数次。每建立一个 TCP 网络连接，都要分配内核文件描述符、TCP 控制块，TCP 服务器端还要对新连接进行调度。这些开销将在第 6 章详细讨论。

本文第 6 章将提出一个用户态网络协议栈，将操作系统的网络协议栈开销转移到用户态库和可编程网卡，实现接近硬件性能的网络传输。

2.2.4　数据结构处理

传统基于软件的内存数据结构存储系统在客户端（计算节点）和服务器端（存储节点）都需要通过操作系统内核访问网络，还需要通过软件处理共享数据结构的并发访问，带来一系列开销。

其中操作系统内核的开销是上一小节已经讨论的。即使这些开销被全部消除，内存数据结构处理的吞吐量瓶颈仍受限于数据结构操作中的计算和随机存储器访问中的延迟。一方面，基于 CPU 的键值存储需要花费数百个 CPU 周期来进行键比较和哈希槽计算。另一方面，键值存储哈希表比 CPU 高速缓存大几个数量级，因此存储器访问等待时间由实际访问模式的高速缓存未命中等待时间决定。第 5 章将详细分析，即使所有 CPU 核心全部处理键值操作，其吞吐量仍远远低于主机 DRAM 内存所能提供的随机访问能力。为此，第 5 章将提出基于可编程网卡的内存数据结构存储。

2.3　可编程网卡的架构

可编程网卡一方面需要提供相比主机 CPU 更高的成本效率，另一方面需要提供可编程性以应对工作负载和数据中心功能的改变（例如，从网络虚拟化扩展到存储虚拟化，甚至本文提出的网络功能和数据结构处理）。因此，可编程网卡通常不是单个固定功能的芯片，而是一个片上系统（system on chip，SoC）。一些可编程网卡还具有与主机 CPU 相似的控制面处理能力。因此，根据数据面的体系结构来区分可编程网卡的架构，大致可分为四类：专用芯片、网络处理器、通用处理器和可重构硬件。

2.3.1　专用芯片（ASIC）

ASIC（Application Specific Integrated Circuit，专用集成电路）是为某些应用专门开发的集成电路芯片。ASIC 开发门槛比较高，研发周期也比较长。在目前的技术水平下，中等复杂度的 ASIC 前期投入的一次性研发（NRE，Non-Recurring Engineering）成本会在数百万到一两千万美元左右，并且需要一两年的开发周期。在过去，开发 ASIC 往往是专业硬件芯片公司才能做到的事情。但随着云计算平台规模的不断扩大，云计算系统公司也开始尝试针对自己的云独立设计专用芯片。

大多数网卡厂商设计的数据中心网卡具有一定的可编程性。例如 Mellanox ConnectX-5[121] 硬件网卡不仅有标准网卡的收发包功能、接收端扩展（RSS）和虚拟机队列（VMQ）、TCP 校验（checksum）和分段卸载（LSO）、RDMA 功能、QoS 网络队列和路由表卸载功能等，还包括一个网络交换机，具有可配置的匹配-操作表（Match-Action Table），可以实现虚拟交换机（OVS）卸载功能。匹配-操作表尽管可以编程，但不是图灵完全的，灵活性不足以解析新的数据包封装格式，不能解析应用层协议，也不能实现新的加密算法。换言之，专用芯片的数据面可编程性是通过配置匹配-操作表，而非通用编程语言实现的。

传统上，微软与网络专用芯片供应商（如英特尔、Mellanox、Broadcom 等）合作，为 Windows 中的主机网络实现卸载。例如，在 20 世纪 90 年代的 TCP 校验（checksum）和分段卸载（LSO），接收端扩展（RSS）和虚拟机队列（VMQ）用于 2000 年代的多核可扩展性，以及 2010 年的无状态卸载，用于 Azure 的虚拟网络方案中的 NVGRE 和 VxLAN 封装。事实上，微软提出的通用流表（Generic Flow Table，GFT)[110] 最初设计为由 ASIC 供应商实现。微软在业界广泛分享早期设计理念，以观察供应商是否能满足要求。随着时间的推移，微软对这种方法的热情逐渐降低，因为没有出现能够满足云计算平台规定的所有设计目标和约束的设计[10]。

近年来，学术界也提出了多种可编程网卡架构，如 FlexNIC[104,122], Emu[123], SENIC[124], sNICh[125], Uno[126], PANIC[127] 等，都是基于专用芯片的，主要旨在优化网络交换机的匹配-操作表，实现更丰富的网络虚拟化功能、主机内虚拟机间的高性能通信、更灵活的直接内存访问（DMA）等[104,122]。

可编程网卡供应商在实现一种专用芯片架构时，面临的一个主要问题是 SR-IOV 是一个全有或全无卸载的例子。如果在可编程网卡中无法成功处理任何所需的 SDN 功能，则 SDN 协议栈必须恢复为通过基于软件的 SDN 实现，几乎丧失了 SR-IOV 卸载的所有性能优势。

用于 SDN 处理的定制芯片设计提供了最高的性能潜力。然而，随着时间的

推移，它们缺乏可编程性和适应性。特别是，从提出需求规格与芯片的到货之间的时间跨度较长，大约需要 1 至 2 年。在这个范围内需求持续变化，使得新的芯片已经落后于软件要求。定制芯片的设计必须继续为服务器的 5 年生命周期提供所有功能（以微软 Azure 云的规模，改造大多数服务器是不可行的）。全有或全无卸载意味着今天制定的定制芯片设计规范必须满足未来 7 年的所有 SDN 要求。

诸如 Mellanox ConnectX 系列的可编程网卡添加了嵌入式 CPU 内核来处理新功能，嵌入式 CPU 上运行固件（firmware）。例如 DCQCN 拥塞控制协议在 Mellanox ConnectX-3 网卡上就是用固件实现的，在 Mellanox ConnectX-4 网卡上才被固化成硬件。但是，这些嵌入式 CPU 内核并不是为高速数据包处理设计的，可能成为性能瓶颈。此外，随着新功能的增加，这些内核可能会随着时间的推移而增加处理负担，从而加剧了性能瓶颈。最后，这些嵌入式 CPU 内核通常需要通过网卡的固件更新进行编程，这些固件往往需要网卡厂商来进行编程，因此会减慢新功能的部署。

因此，对云服务商来说，基于专用芯片搭建可编程网卡往往是不可行的，需要在数据面引入足够的可编程性。

2.3.2　网络处理器（NP）

早在 1990 年代，为了提供性能和灵活性，网络处理器就被广泛用于路由器和交换机中。用于数据中心的现代网络处理器一般由队列、数据包处理核心、流处理（flow processing）核心、加速专用电路和控制核心等部分构成。不同于专用芯片中运行固件的嵌入式 CPU，网络处理器的数据包处理核心和流处理核心的处理能力强大得多。

如图 2.8 所示，一个作为交换机的典型网络处理器从网络中接收输入。为了提供服务质量保证（QoS），网络处理器支持根据规则对输入数据包进行分类，或经过数据包处理核心的无状态处理的得到数据包的分类，确定优先级和队列号，进入相应的任务队列。每个任务队列可能关联一组或多组流处理核心。流处理核心从关联的队列中依次取出任务，进行有状态的流处理；处理结果进入输出队列，并经过另外一组数据包处理核心的无状态处理，输出到网络。在主机网卡的场景中，网络处理器不仅需要处理网络数据包，还需要与主机交互。这包括处理来自主机的数据发送请求和来自网络的接收主机数据请求。当主机收到这些请求时，它们与输入的数据包一样进入任务队列，并排队接受流处理核心的处理。有时，流处理核心对一个数据包的处理依赖从主机内存 DMA 的结果（例如 RDMA 单边读请求需要从主机内存中取出相应的数据，再组成数据包发送响应）或者对同一个流的下一数据包的处理（例如 Web 应用程序防火墙（WAF）和能

解析 HTTP 协议的七层负载均衡器）。为此，网络处理器一般提供了请求回挂功能，即把处理的中间状态和依赖关系提交给调度器，让调度器在依赖关系满足时重新处理该请求。

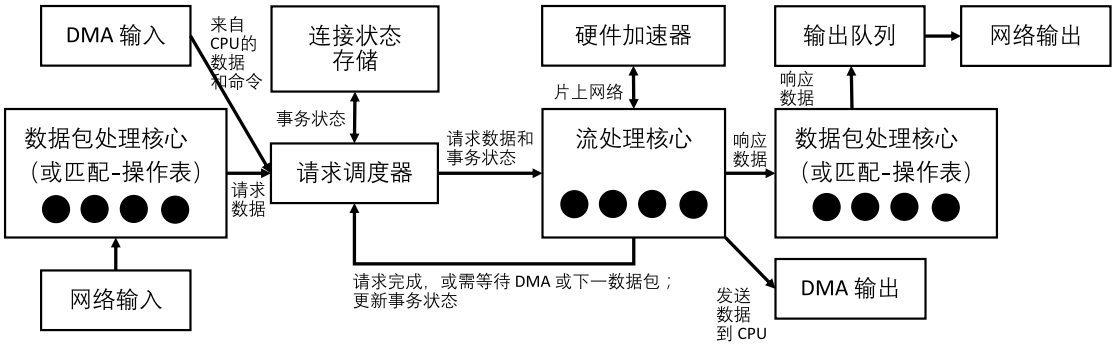


图 2.8　网络处理器的一般架构。

网络处理器之所以能比通用处理器更高效地处理网络数据包和主机请求，是因为它把很多数据包处理功能固化成了硬件逻辑。第一，在保证服务质量（QoS）的调度器方面，硬件调度器可以实现是中心化先到先处理调度模型（c-FCFS），而网卡根据连接哈希来进行分派的 RSS 技术试图模拟分布式先到先处理调度模型（d-FCFS）[128-129]。中心化调度中，中心调度器维护一个调度队列，并将队首任务分派给刚刚处理完上一任务的处理器。分布式调度中，中心调度器将请求均匀分派到各个处理器的队列，每个处理器只能处理自己的队列，因此可能存在有的处理器队列非空而其他处理器闲置的情况。排队论表明，c-FCFS 比 d-FCFS 的各个核心的负载更加均衡，请求处理的平均延迟和尾延迟 ① 也更低。况且，网卡 RSS 技术往往并不能准确地模拟 d-FCFS 模型，因为数据包的处理可能是有状态的，很多情况下需要把相同的连接分配给相同的处理器核心，这时只有在连接数量很大、每条连接的数据包数量相同时，才能模拟 d-FCFS 模型；而现实中每条连接的数据包数量常常呈现长尾分布，此时处理器核心的负载就是不均衡的。理论分析[130] 表明，长尾分布下负载的不均衡程度与处理器核心的数量正相关。例如，对 64 个流处理核心的网络处理器，如果使用根据哈希分配连接的 d-FCFS 方式，长尾分布下负载最高的核心负载是平均核心负载的 6 倍。

第二，网络处理器的数据包解析、查找表、数据结构、定时器、DMA 引擎等很多常用模块是用硬件实现的。这些查找表和数据结构如果用软件实现，将消耗较多的指令数，影响数据面处理性能。例如，在软件中设置和触发一次定时器约需要 200 条指令，解析 TCP/IP 协议报文需要约 100 条指令[119]。从延迟上看，在 1 GHz 的网络处理器中，如果需要把数据包发送延迟控制在 1.5 μs 以内，假

① 尾延迟（tail latency）是指一组延迟采样中最高的延迟。为了统计上的稳定性，一般对一组延迟采样从小到大排序，然后取 99%、99.9% 或其他分位点（percentile）的延迟作为尾延迟。

设 PCIe 延迟为 0.3 μs，网络数据链路层（MAC）延迟为 0.2 μs，每个数据包在可编程网卡内的处理是串行的，则软件处理每个数据包的指令数不能超过 1000 条。从吞吐量上看，如果需要支持 40 Gbps 网络下的 64 字节小包线速（line-rate）处理，每秒需要处理 60 M 个数据包，假设网络处理器有 64 个处理核心，每个核心每秒能执行 1 G 条指令，则每个处理核心平均每个数据包的指令数不能超过 1000 条。因此，节约数据包处理的指令数对延迟和吞吐量都很重要。网络处理器通过用硬件实现常用数据结构和算法，大大减轻了数据包处理器和流处理器的负担。这些硬件模块与处理核心之间通常使用片上网络互连，使得处理核心可以在处理过程中随时调用这些模块。

第三，网络处理器内流处理核心的线程调度与上下文管理是由硬件实现的，因此可以实现细粒度的延迟隐藏。例如，流处理核心如果调用了一个需要较长时间的 DMA 操作，或者需要等待一个定时器，硬件会自动保存其现场（包括寄存器和正在处理的流状态），将该线程放入未就绪队列，并切换到就绪队列中的下一线程；如果就绪队列为空，且并发线程数尚未达到硬件限制，则可以从任务队列中取出下一任务，新建一个线程。当 DMA 操作返回或定时器被触发时，未就绪队列中的线程就被切换到就绪队列。大多数网络处理器使用如上所述的非抢占式协作调度。为了支持严格优先级的服务质量保证，并在高优先级流量较小时能够充分利用处理能力来处理低优先级流量，一些网络处理器具备抢占式调度功能。不同于主机上的 CPU，网络处理器把操作系统的上下文管理和调度功能用硬件实现，相比 CPU 大大降低了上下文切换的开销。在数据中心场景下，CPU 应用程序的微秒级延迟隐藏成为越来越重要的问题[6]，网络处理器的 “硬件操作系统” 设计也可以给主机 CPU 的设计带来一些启示。

第四，网络处理器内的内存层次结构是定制化的，因此访存的效率比通用处理器高，这点与 FPGA 的体系结构优势相似。通用处理器的 “内存墙” 问题是众所周知的，所有流处理核心如果都从共享内存中读写流状态，缓存一致性的开销是很高的，给处理器的设计带来了很多负担。而网络处理器在硬件中实现了数据包内容与数据包处理核心的绑定、流状态与流处理核心的绑定，数据包内容和流状态通过定制的数据通路进行搬运和缓存，在相同芯片面积和制程下，提高了内存带宽。

需要指出的是，虽然网络处理器相比通用处理器能耗效率更高，但编程较为困难。早期的网络处理器通常使用专用指令集的微码（microcode）进行编程，由于缺少编译器，编程语言的抽象程度与汇编相似。现代网络处理器一般使用通用 CPU 核心（如 ARM 和 MIPS）作为数据包处理器和流处理器，因而可以利用完善的编译器和开发工具链，使用 C 语言编程。可编程网卡内硬化的功能则使用库函数的形式进行调用，类似 Intel CPU 上的原子操作和向量操作。相比通用处

理器，现代可编程网卡的编程困难性主要体现在开发者需要花时间了解这些专有库函数和网卡的体系结构，也不能直接使用通用处理器上成熟的基于 DPDK 等框架的代码。

在性能方面，网络处理器最大的问题是单核的性能较低，导致两个后果。首先，有状态流往往只映射到一个处理核心或线程，以防止在单个流中进行状态分片和无序处理。即使一些网络处理器支持把有状态处理划分为多个阶段，并用多核流水线化处理，但流水级的数量受到硬件和流处理逻辑依赖的限制。因此，对有状态处理而言，单流的数据包吞吐量不能超过网络处理器单核处理能力的数倍。单核性能较低的第二个后果是为了支持更高的网络带宽，处理核心的数量必须线性增长，不仅增加了芯片面积和耗电量，也给核心间互连、内存层次、片上网络的设计带来了挑战。在 40 Gbps 及以上的更高网络速度下，核心数量显著增加。分散和收集数据包的片上网络和调度程序变得越来越复杂和低效。将数据包送入处理核心，处理数据包，再发送到网络的整个流程经常需要 10 μs 或更多的延迟。此时的延迟明显高于专用芯片，并且具有更大的可变性。

业界的网络处理器产品包括 Netronome NFP-32xx，Cavium OCTEON，Tilera，Mellanox NP-5 等。其中有些网络处理器只有流处理核心，没有数据包处理核心，但总体架构是类似的。

2.3.3　通用处理器（SoC）

注意到网络处理器较难编程的问题，业界提出了基于通用处理器的可编程网卡架构。此架构与 2011 年微软亚洲研究院提出的 ServerSwitch[131] 架构类似，由一个硬件网络交换机和一个通用处理器构成。例如，Mellanox BlueField[132] 可编程网卡由一个 Mellanox ConnectX-5 硬件网卡和一个多核 ARM 处理器构成。其中硬件网卡就是第 2.3.1 节讨论的专用芯片，实现了基本的数据包解析、分类、排队和转发功能。如图 2.9 所示，多核 ARM 处理器、Mellanox ConnectX-5 传统网卡和网卡的板上 DRAM 通过一个 PCIe 交换机互连。PCIe 交换机进一步连接到主机 CPU 上，实现多核处理器、传统硬件网卡和主机 CPU 之间三方的互连。传统网卡与多核 ARM 处理器之间通过可编程网卡板上的 DRAM 通信。硬件网卡与主机 CPU 以及多核 ARM 处理器与主机 CPU 均可通过主机 DRAM 通信。典型的数据包接收流程是：传统网卡把接收的数据包发送到可编程网卡板上的 DRAM。多核 ARM 处理器从 DRAM 中取出数据包，进行处理后发送到主机 CPU 的 DRAM。

基于多核 SoC 的网卡使用大量的嵌入式 CPU 内核来处理数据包，牺牲一些性能以提供更好的可编程性。基于多核通用处理器的 SoC 架构与网络处理器（NP）架构有很多相似之处，也因此经常被统称为同一类架构，但事实上它们有

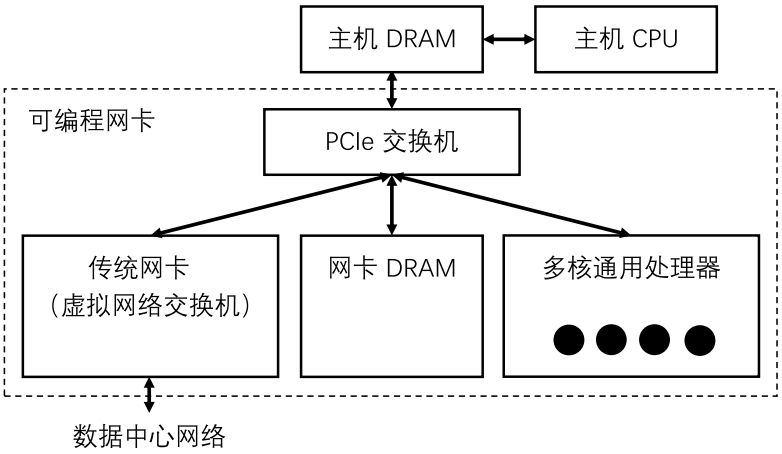


图 2.9　基于通用处理器的可编程网卡架构。

很多区别。相比网络处理器，多核 SoC 更容易编程，即可以采用标准的 DPDK 代码并在熟悉的 Linux 环境中运行。现有基于主机 CPU 的网络功能代码也很容易交叉编译到多核 SoC 上运行，而网络处理器的代码一般需要重新编写。但是，多核 SoC 中的通用处理器核心采用片外 DRAM 与传统网卡通信，比 NP 架构片上高速缓存的访问效率低。表 2.1 比较了数据中心可编程网卡的 SoC 和 NP 架构。

表 2.1　多核通用处理器与网络处理器架构的比较。其中的数字来自可编程网卡厂商的白皮书，实际应用性能受到应用复杂度的影响，不一定能达到理论性能。

比较项目	多核通用处理器（SoC）	网络处理器（NP）
指令类型	ARM / MIPS 标准指令集	ARM / MIPS 扩展指令集
操作系统	通用操作系统（如 Linux）	无操作系统或定制化操作系统
操作系统、分页等	支持	一般不支持
上下文切换与调度	软件操作系统	硬件
锁、定时器等	软件	硬件
板上/核间通信	共享内存	定制数据通路
数据包缓冲区	片外 DRAM	片上高速缓存
数据包处理框架	通用（如 DPDK）	专用
多核排队模型	d-FCFS（硬件分派）	c-FCFS（硬件调度）
平均处理延迟	约 5 μs	小于 2 μs
单核处理能力	约 3 M pps	约 1 M pps
处理器核心数量	约 8 个	约 64 个
总数据包处理能力	约 24 M pps	约 64 M pps
功耗	10 W 至 20 W	

与前面网络处理器一节讨论的相同，多核 SoC 和网络处理器都受限于单核性能。尽管多核 SoC 的单核性能高于网络处理器，但多核 SoC 的核间通信开销高于网络处理器处理核心组成的硬件流水线，从而多核 SoC 中每个数据包一般

是在一个处理器核心上处理到完成（run-to-completion）。因此，单流性能一般是在同一数量级的，即在 5 M pps（packet per second，数据包每秒）左右。在核心数量增加方面，由于多核 SoC 大多采用分布式先到先服务调度（d-FCFS）模型，如果软件不使用核间工作窃取（work stealing）等技术，处理器间负载的不均衡会随核心数量的增加变得越来越严重。在延迟方面，由于多核 SoC 处理器间的负载不平衡，以及多核 SoC 采用通用操作系统和通用的共享内存层次带来的操作系统调度、中断和缓存不命中等非确定性延迟，多核 SoC 的延迟稳定性一般比网络处理器更差，尾延迟（tail latency）一般比网络处理器更高。

因此，虽然多核 SoC 方法具有熟悉的编程模型和良好的应用兼容性，但在更高的网络速度下，单流性能、更高的延迟和更差的可扩展性是其明显的弱点。

从体系结构的角度看，多核 SoC 与主机 CPU 的相似程度是最高的，它们都使用通用处理器。但如表 2.2 所示，无论在成本还是性能功耗方面，可编程网卡使用的通用处理器都有较明显的优势。此外，如第 1 章所讨论的，在云计算数据中心，主机 CPU 可以用于出售，其潜在售价远高于单一 CPU 组件的硬件价格。因此，在数据中心使用嵌入在可编程网卡内的通用处理器，比传统上使用主机 CPU 的方法仍然是有优势的。

表 2.2　可编程网卡内的通用处理器与主机 CPU 的比较。

比较项目	可编程网卡通用处理器	服务器 CPU
架构	RISC（ARM / MIPS）	x86
时钟频率	1 至 2 GHz	2 至 3 GHz
功耗	10 W 至 20 W	约 100 W
价格	数十至上百美元	数百至上千美元
单核处理能力	约 3 M pps	约 5 M pps
处理器核心数量	约 8 个	约 20 个
总数据包处理能力	约 24 M pps	约 100 M pps
是否支持虚拟化	不一定	支持
平均处理延迟	约 5 μs	
操作系统	通用操作系统（如 Linux）	
数据包处理框架	通用（如 DPDK）	

2.3.4　可重构硬件（FPGA）

FPGA 是一种硬件可重构的体系结构，常年来被用作专用芯片（ASIC）的小批量替代品，被广泛应用于原型设计，逻辑电路模拟，以及高端路由器等领域。近年来 FPGA 在数据中心大规模部署，以同时提供强大的计算能力和足够的灵活性。微软和百度在有效利用 FPGA 加速云计算方面做了很多工作。例如微软

在 Azure 云平台上已经全面部署了 FPGA 模块[48,133]。百度将 FPGA 用于数据压缩[134]、数据库 SQL 处理[135]、深度学习[136] 和其他丰富的应用场景[137]。2016 年，Intel 公司以 167 亿美元的价格收购 FPGA 巨头 Altera 公司，以保持其在数据中心领域的领导地位，并探索 CPU 与 FPGA 结合的异构服务器计算架构。

直观上，FPGA 是一个可以用编程的方法重新组合的一大堆电子元器件。这些元器件包括逻辑门（如与，或，非门），寄存器（Register），加法器，静态内存（SRAM）等等，用户可以定制它们之间的连接从而组成不同的电路。图 2.10 展示了 FPGA 的基本计算单元，即由可编程的逻辑门和寄存器组成的逻辑元件。

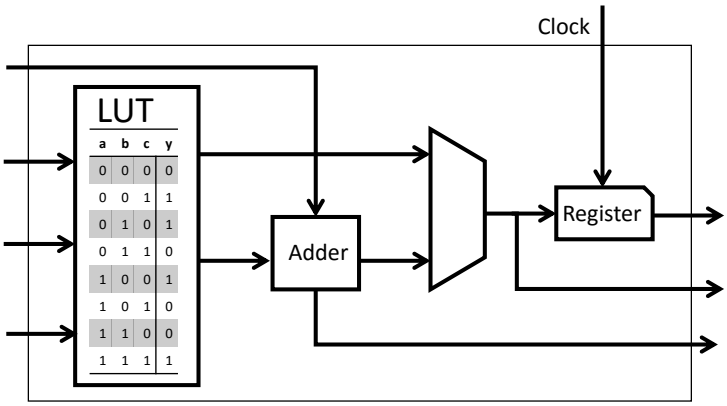


图 2.10　FPGA 的基本计算单元 – 逻辑元件（logic element）。

如今的 FPGA 除了基本元件，还加入了越来越多的 DSP 和硬核（hard IP），以提高乘法、浮点运算和访问外围设备的性能。FPGA 上的硬核可以支持 DDR、Ethernet、PCIe 等，以连接板上 DRAM、数据中心网络、主机 PCIe 插槽等。图 2.11 显示了本文使用的 FPGA 板的逻辑图。

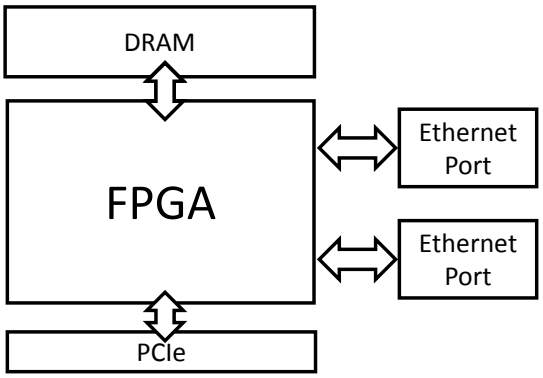


图 2.11　FPGA 板的逻辑图。

以 CPU 为代表的通用处理器通常采用冯 · 诺依曼结构及其变体。冯 · 诺依曼结构中，由于处理器（如 CPU 核）可能执行任意指令，就需要有指令存储器、

译码器、各种指令的运算器、分支跳转处理逻辑。由于指令流的控制逻辑复杂，不可能有太多条独立的指令流，因此 GPU 和 CPU 都可以使用 SIMD（单指令流多数据流）来让多个处理单元以同样的步调处理不同的数据。而 FPGA 每个逻辑单元的功能在重编程（烧写）时就已经确定，不需要指令。

冯 · 诺依曼结构中使用内存有两种作用：保存状态和处理器间通信。冯 · 诺依曼结构中，各处理器共享内存，因此需要做访问仲裁；为了利用访问局部性，每个处理器有一个私有的缓存，这就要维持执行部件间缓存的一致性。对于保存状态的需求，FPGA 内有大量的片上内存（BRAM）模块，每个模块可以分别连接到需要使用相应数据的逻辑模块，无需不必要的仲裁和缓存。对于逻辑模块间通信的需求，FPGA 每个逻辑模块与周围逻辑模块的连接在重编程（烧写）时就已经确定，并不需要通过共享内存来通信。

由于数据通路是定制化的，FPGA 可以同时利用流水线并行 ①、数据并行 ② 和请求并行 ③ 以降低延迟、提高吞吐量。FPGA 可以根据数据和控制的依赖关系，将大量处理单元组织成计算流图，其中有依赖关系的是流水线并行，没有依赖关系的是数据并行。本文第 4 章将详细讨论。FPGA 也可以构建调度器以实现灵活的请求并行，保持处理单元间的负载均衡，并能隐藏请求处理中的外部延迟，这是本文第 5 章将讨论的。

基于指令的 GPU 和 CPU ④ 对流水线并行的利用是有限的。尽管指令处理核心是流水线化的，但其深度有限；由于多核之间通信开销较高，利用多核组成流水线往往效率较低。

GPU 的并行计算单元和 CPU 的向量指令可以利用数据并行。但是，GPU 和 CPU 中的数据并行都是单指令流多数据流（SIMD）模式，并行处理的数据必须执行相同的运算，且并行的计算之间不能存在数据依赖。但在很多应用中，不同数据所需的运算不同（如防火墙中，不同规则匹配的数据包域和匹配方式不同），若要使用 SIMD 模式实现数据并行，就需要每个计算单元遍历所有可能的运算，从而浪费一些资源用于执行不必要的运算。此外，数据包处理中很多操作间存在数据或控制依赖关系，仅采用数据并行所能获得的加速比有限。而在 FPGA 中，只要数据和控制依赖关系可以在编译时确定，就可以将计算流图编译成一条硬

①流水线（pipeline）由若干流水级（stage）组成，每个任务依次经过各个流水级处理。各个流水级之间的处理一般存在依赖关系。在任一时刻，每个流水级在处理不同的任务。

②在本文中，数据并行是指并行处理互不关联的数据，例如向量点乘可以采用数据并行，不同的防火墙规则也可以并行处理。注意，与分布式机器学习中的 “数据并行” 术语含义不同。

③在本文中，处理一个网络数据包或主机 CPU 卸载到加速卡的一个操作称为一个工作请求（work request），这是 RDMA 中的术语。请求并行是指并发执行不同的工作请求。为了便于理解，文中经常用数据包代替工作请求。

④本节中，CPU 指通用处理器，包括服务器主机 CPU 和 SoC 可编程网卡上的处理器。

件逻辑组成的流水线。最后，很多 SIMD 指令对数据对齐和数据类型有限制，而 FPGA 可以实现灵活的数据通路和数据类型。

请求并行不能降低单个数据包的处理延迟，但可以提升系统的吞吐量，进而降低成本。GPU 和 CPU 都可以利用请求并行。GPU 同组的计算单元可以分别处理不同的数据包，然而 SIMD 架构的这些处理单元必须按照统一的步调，执行相同的操作。由于不同数据包所需的处理操作不同，与上文数据并行所讨论的一样，需要浪费资源执行不必要的操作。GPU 不同组的计算单元可以独立地处理不同的数据包，但 GPU 的编程模型通常要求这些数据包一起输入、一起输出，增加了输入输出延迟。尽管 CPU 的每个核心可以独立地从网卡接收和发送数据包，但由于 CPU 与网卡间的通信代价较高，为了维持高吞吐量，往往需要批量接收和发送数据包。当数据包是逐个而非成批到达的时候，FPGA 可实现相比 GPU 和 CPU 更低的延迟。此外，尽管 CPU 的不同核心可以异步地处理不同的数据包，但核心间的负载均衡是个难题。表 2.3 总结了 FPGA、GPU、CPU 对流水线、数据和请求并行的利用能力。

表 2.3　不同体系结构对流水线、数据和请求并行的利用能力。

	流水线并行	数据并行	请求并行
示例	依次处理数据包的 MAC 层、IP 层、TCP 层、应用层；计算哈希	匹配防火墙规则；计算校验和；向量计算	处理不同数据包；键值操作
FPGA	能利用：定制流水线	能利用：定制的并行处理单元	能利用：定制调度器
GPU	有限利用：指令处理流水线、多核组成的流水线	有限利用：SIMD 向量操作	能利用，但有显著的延迟问题
CPU	有限利用：指令处理流水线、多核组成的流水线	有限利用：SIMD 向量指令	能利用，但有延迟和负载均衡问题

因此，FPGA 相比 GPU 和 CPU 等基于指令的处理器具有延迟优势。对于网络数据包处理，FPGA 的处理延迟可达微秒甚至百纳秒级。如果使用 GPU，要想充分利用 GPU 的计算能力，批量大小（batch size）就不能太小，延迟将高达毫秒量级，是 FPGA 的 1000 倍以上。[①] 对于主机 CPU，即使使用 DPDK 这样高性能的数据包处理框架，延迟也有 4 至 5 微秒，比 FPGA 高一个数量级。更严重的问题是，通用 CPU 的延迟不够稳定。例如当负载较高时，转发延迟可能升到几十微秒甚至更高；现代操作系统中的时钟中断和任务调度也增加了延迟的不确定性。在数据中心，延迟，特别是尾延迟是很重要的。FPGA 处理网络数据包

[①]这是根据现有基于 GPU 批量处理模型的网络功能处理框架的估计。如果 GPU 支持流式处理的编程模型，其延迟将显著降低。

的延迟为百纳秒级别，即使需要通过 PCIe 来访问主机内存，也只需亚微秒级的 PCIe 延迟。[①] 综上，对流式计算的任务，FPGA 比 GPU 和 CPU 天生有延迟方面的优势。

FPGA 也并不是万能药，主要有如下几方面的技术挑战，如表 2.4 所示。

表 2.4　用 FPGA 作为可编程网卡的挑战。

对比的体系结构	FPGA 的挑战	解决方法
CPU/GPU/NP/SoC	时钟频率低	利用 FPGA 内的大规模并行性
CPU/GPU	DRAM 内存带宽低	定制数据通路，并行利用片上 BRAM 内存，减少 DRAM 使用
CPU/GPU/NP/SoC	硬件描述语言编程复杂、难以调试	基于高层次综合技术，对软件开发者友好的编程框架
CPU/GPU/SoC	软硬件生态系统较为封闭	开放硬件平台、编程框架和 IP 核
CPU/GPU/NP/SoC	芯片面积受限，不适合逻辑复杂的场景	分离控制面与数据面；基于定制化指令的数据面
CPU	访问主存 PCIe 延迟高	设计高效数据结构，并利用乱序执行实现延迟隐藏
CPU	访问主存 PCIe 带宽受限	设计高效数据结构，并利用板上缓存
CPU/GPU/NP/SoC	升级需要重新烧写，中断服务	支持动态重配置、无缝服务升级的 FPGA 操作系统
CPU/NP/SoC	任务切换开销高	空间多路复用，而非时分复用
ASIC	一些计算密集型负载效率较低	将通用模块固化为硬核

第一，与 CPU 或 GPU 相比，FPGA 通常具有更低的时钟频率和更小的存储器带宽。例如，FPGA 的典型时钟频率约为 200MHz，比 CPU 慢一个数量级（2 至 3 GHz）。同样，FPGA 的单块片上 BRAM 存储器或外部 DRAM 的带宽通常为 2 至 10 GBps，而 Intel Xeon CPU 的内存带宽约为 60 GBps，GPU 更可达数百 GBps。但是，CPU 或 GPU 只有有限的核心数量，这限制了并行性。FPGA 内置了大量的并行性。现代 FPGA 可能拥有数百万个逻辑单元，数百 K 比特的寄存器，数千个片上 BRAM（每个数 MB 容量）和数千个数字信号处理（DSP）模块。从理论上讲，它们中的每一个都可以并行工作。因此，FPGA 芯片内部可能会同时运行数千个并行的“核”。虽然单个 BRAM 的带宽可能有限，但如果并行访问数千个 BRAM，则总内存带宽可以达到数 TBps！因此，为了实现高性能，程序员必须充分利用这种大规模的并行性。

第二，传统上，FPGA 使用诸如 Verilog 和 VHDL 之类的硬件描述语言（HDL）

[①]未来 Intel 推出通过 QPI 连接的 Xeon + FPGA 之后，CPU 和 FPGA 之间的延迟更可以降到 100 纳秒以下，与 CPU 访问主存的延迟在同一数量级。

进行编程。这些语言的抽象层次较低，难以学习，编程也很复杂。尽管近年来 Chisel[53] 等高层次硬件描述语言开始流行，但仍然需要程序员具有数字逻辑设计的基础知识和硬件设计的思维方式。因此，软件程序员社区已经远离 FPGA 多年了[54]。为了简化 FPGA 编程，工业界和学术界开发了许多高级综合（HLS）工具和系统，试图将高级语言（主要是 C）的程序转换为 HDL。但是，它们或者没有充分利用 FPGA 中大规模的并行性，或者延迟过高，或者只是硬件开发工具链的补充，因而都不适合网络功能处理。第 4 章将提出此问题的一个解决方案。

第三，传统上，FPGA 的硬件开发者社区较为封闭。首先，FPGA 的开源生态系统不完善，因此 FPGA 开发者往往需要从头实现或从第三方厂商购买通用模块（IP 核）。通用模块的开发和购买成本使很多中小企业和学术界研究者望而却步。其次，小批量购买 FPGA 板卡的价格较高，削弱了 FPGA 相比其他体系结构的成本优势。近年来，随着 FPGA 成为数据中心通用加速器，FPGA 厂商和学术界不断推动 FPGA 生态系统的建设。例如，NetFPGA[138] 开放网络编程平台、Xilinx 的 SDx[58] 编程框架和 P4[102] 跨平台网络编程语言。各大云服务商也推出了按需租用 FPGA 的云服务以及 IP 核市场，开发者不再需要花费很高的一次性成本来购买板卡、重造轮子。本文第 4 章的网络元件库和第 5 章的键值数据结构是 FPGA 生态系统的有益补充。

第四，FPGA 用数字电路实现的逻辑规模受到 FPGA 可重构单元数量的限制，而后者又受到芯片面积的限制。因此，FPGA 不适合用来实现非常复杂的逻辑。对于逻辑复杂的情况，一般采用两种方案共同解决。首先，区分控制面和数据面，在 FPGA 中实现数据面，并在通用处理器上实现控制面。例如，微软的虚拟网络加速器[10] 把新连接发送到主机 CPU 上的控制面，由软件确定处理规则，并将规则卸载到 FPGA 数据面，使得该连接后续的数据包可被 FPGA 处理。控制面与数据面分离是贯穿本文始终的设计思想。相比使用通用处理器的可编程网卡，使用 FPGA 增加了分离控制面与数据面的编程复杂性。其次，提取复杂逻辑中的通用操作，用数字逻辑实现定制化指令。复杂的逻辑通过一系列定制化指令实现，这些指令存储在内存中，不占用可重构单元。微软的虚拟网络加速器[10] 设计了定制化的匹配-操作表。微软的神经网络处理器[139] 为神经网络计算定制了向量处理指令。本文第 5 章的原子操作、向量操作也是定制化指令的例子。

第五，为了实现 FPGA 与 CPU 的细粒度协同处理，FPGA 和 CPU 需要通过 PCIe 总线进行通信；由于 FPGA 板上 DRAM 容量相比主机 DRAM 一般小很多，大规模数据结构也需要存储到主机 DRAM 上，需要通过 PCIe 总线来访问。然而，PCIe 总线的延迟为数百纳秒，比 CPU 访问主存高一个数量级；Gen3 x8 的有效带宽约为 6 GB/s，比 CPU 访问主存低一个数量级。因此，可编程网卡上的加速应用需要设计高效的数据结构，节约访存次数，利用乱序执行技术隐藏延迟，

并充分利用片上 BRAM 和板上 DRAM 的缓存。这是第 5 章的主题。

第六，与基于指令的处理器相比，FPGA 的任务切换开销较高。一方面，尽管 FPGA 可以通过动态重配置（dynamic reconfiguration）来实现数据面无中断的任务切换，但这样就必须固定一部分资源在 FPGA 芯片上的物理位置，使得 FPGA 放置（placement）和路由（routing）的全局优化受到限制；还需要增加逻辑来辅助动态重配置。这带来了 FPGA 面积的开销。另一方面，FPGA 动态重配置需要几十毫秒的时间，远长于 CPU 任务切换的时间（基于通用操作系统的 CPU 任务切换一般为数微秒，基于专用操作系统的 CPU 任务切换一般为数百纳秒，基于专用处理器的任务切换可在数十纳秒内完成）。这使得 FPGA 不能像 CPU 那样实现细粒度的分时复用。因此，目前 FPGA 的多用户复用主要是在空间上而非时间上，类似把不同的 CPU 核心分配给不同的虚拟机。此外，FPGA 动态重配置期间，用户逻辑无法工作，因此需要 CPU 在这段时间内代替 FPGA 进行处理，或者暂停 FPGA 的服务，从而影响服务质量。本文所讨论的 FPGA 可编程网卡均为数据中心基础架构的第一方（first party）用途，而不是对外公开出售的第三方（third party）用途，因此几乎没有动态切换任务的需求；主要的挑战是利用动态重配置实现服务升级，并在升级期间实现服务质量保证。本文作者参与（但非主导）的一项工作 Feniks[140] 和最近的 AmorphOS[141] 旨在解决此问题。

第七，一些类型的工作负载是计算密集型的，且在 FPGA 内实现的效率明显低于专用芯片。第一类是加密解密等标准化操作。例如，Intel QuickAssist 加速卡[142] 基于 ASIC 的 RSA 非对称加密比第 4 章基于 FPGA 的实现，吞吐量约高 10 倍。第二类是查找表等常见数据结构。例如，内容寻址内存（Content-Addressable Memory，CAM）是很多并发操作调度器和数据结构的基础。CAM 在专用芯片中可以用三态门实现，而在 FPGA 中实现的效率较低。第 7.2.1 节的未来工作展望将提出借鉴网络处理器的架构，将这些 FPGA 内实现效率不高的操作硬化。

最后需要指出，从商业和供应链的角度考虑，FPGA 存在一定的劣势。FPGA 零售价格较高，开发工具链的授权也较昂贵，因此如果应用规模不够大，硬件和工具链的平摊成本相比基于网络处理器或通用处理器的可编程网卡可能不具优势。而对应用规模非常大的公司，用于数据中心的 FPGA 目前仅有两家主要厂商，供应链安全有较大的变数，且在两家 FPGA 间切换的二次开发成本较高，因此自研网络处理器或嵌入式通用处理器芯片可能是长期来看成本更低也更可控的选择。FPGA 作为通用处理器和专用芯片间的折中，适用于应用规模中等、应用场景不确定性较大的场景。

2.4　可编程网卡在数据中心的应用

微软 Azure、亚马逊 AWS、阿里云、腾讯云、华为云等云服务商先后公开了基于可编程网卡的数据中心加速实践。

2.4.1　微软 Azure 云

目前，微软在数据中心部署定制化硬件的用途主要包括计算和基础架构两大方面。在计算加速方面，第一，2013 年起，用于必应（Bing）搜索的文档选择和排序算法[48]。2016 年起，使用硬件微服务（hardware microservice）将多个 FPGA 上没有占用完全的资源整合（consolidate）到较少数量的 FPGA 上，提高 FPGA 的使用率。第二，用于压缩和加密算法[143]，早期仅用于必应搜索，后期扩展到 Office 365、Cosmos / Azure 数据湖（data lake）、Onedrive、云存储等服务。第三，2015 年起，用于机器学习推理[139,144-146]，不仅支持深度学习模型，还支持传统机器学习模型。第四，2016 年起，推出支持 FPGA 的虚拟机实例，对第三方（third party）客户出租 FPGA 算力。

基础架构方面的用途主要包括网络和持久化存储。网络方面，2015 年开始，用于网络虚拟化加速[10]。持久化存储方面，将 FPGA 用于加速 Azure 云存储[147]，一方面是在存储后端节点上，利用计算加速部分的压缩和加密算法，提高吞吐量，采用更好（但计算量更大）的压缩算法提高压缩率，节约存储空间；另一方面是在存储前端节点和计算节点上的存储服务，通过 FPGA 加速存储协议栈的数据面，实现数据面绕过虚拟机监控器（hypervisor）并能按照服务质量保证共享存储资源。

考虑到上述需要加速的工作负载，微软在选择数据中心定制化加速硬件时，主要从定制化硬件体系结构、定制化硬件之间的连接范围和 CPU 与定制化硬件间的通信方式三个方面考虑。

在定制化硬件的体系结构方面，FPGA 同时适用于计算密集型与通信密集型负载，且具有低延迟，大规模部署情况下单位算力的成本较低，但编程复杂性高；GPU 适用于计算密集型负载的加速设备，比 FPGA 编程简单，但延迟较高；即使大规模部署，单位算力的成本仍然较高；专用芯片 ASIC，同时适用于计算密集型与通信密集型负载，延迟最低，单位功耗的算力最高，但功能固化后的灵活性较差。例如上述工作负载从必应搜索扩展到压缩加密、网络、存储、机器学习、深度学习等，是一个逐步发展的过程，专用芯片的设计很难一步到位，而重新设计一款专用芯片需要一到两年的时间和数千万美元的一次性研发成本（NRE）。基于以上考虑，微软 Azure 云使用 FPGA 作为数据中心的通用定制化硬件加速各类负载。

在 CPU 和定制化硬件间的通信方式方面，一致性内存（coherent memory）虽

然编程简单，但目前基于 x86 CPU 的体系结构上不容易实现高效率；通过网络访问的带宽和延迟较为受限；直接内存访问（DMA）作为 PCIe 总线上标准的通信模式，成为微软 FPGA 选定的通信方式。在定制化硬件之间的连接范围方面，单机性能不可扩放；机架内使用定制互连的带宽可以较高，但添加定制互连的成本较高；利用数据中心现有网络的互连可以获得最大的可扩放性，成本较低，带宽和延迟也可以满足大多数应用的需求。历史上，微软的 FPGA 部署尝试了上述三种定制化硬件间的连接方式，可以大致划分为三个发展阶段[148]：

1. 专用的 FPGA 集群，里面插满了 FPGA；
2. 每台机器一块 FPGA，采用专用网络连接；
3. 每台机器一块 FPGA，位于网卡和交换机之间，共享服务器网络。

第一个阶段是专用集群，里面插满了 FPGA 加速卡，就像是一个 FPGA 组成的超级计算机。像超级计算机一样的部署方式有几个问题：

1. 不同机器的 FPGA 之间无法通信，FPGA 所能处理问题的规模受限于单台服务器上 FPGA 的数量；
2. 数据中心里的其他机器要把任务集中发到这个加速机柜，构成了 in-cast 的网络流量特征，网络延迟很难做到稳定；
3. FPGA 专用机柜构成了单点故障；
4. 装 FPGA 的服务器是定制的，散热设计和运维都增加了复杂性。

第二个阶段，为了保证数据中心中服务器的同构性（这也是不用 ASIC 的一个重要原因），在每台服务器上插一块 FPGA，FPGA 之间通过专用网络连接。这也是微软在 Bing 搜索集群中最初采用的部署方式[48]。通过 FPGA 加速，必应的搜索结果排序整体性能提高到了 2 倍（换言之，节省了一半的服务器）[48]。本地和远程的 FPGA 均可以降低搜索延迟，远程 FPGA 的通信延迟相比搜索延迟可忽略。

为了加速网络功能和存储虚拟化，微软把 FPGA 部署在网卡和交换机之间。一块 FPGA（加上板上内存和网络接口等）的功耗大约是 30 W，仅增加了整个服务器功耗的十分之一。只要规模足够大，对 FPGA 价格过高的担心将是不必要的。每个 FPGA 有一个 4GB DDR3-1333 DRAM，通过两个 PCIe Gen3 x8 接口连接到一个 CPU socket（物理上是 PCIe Gen3 x16 接口，因为 FPGA 没有 PCIe Gen3 x16 的硬核（hard IP），就拆分为两个 PCIe Gen3 x8 接口使用）。微软 SmartNIC 可编程网卡的架构如图 2.12 所示，其中 FPGA 位于网络和传统网卡之间。

这就是微软部署 FPGA 的第三代架构，也是目前“每台服务器一块 FPGA”大规模部署所采用的架构。FPGA 复用主机网络的初心是加速网络和存储，更深远的影响则是把 FPGA 之间的网络连接扩展到了整个数据中心的规模，做成真正云规模（cloud-scale）的“超级计算机”[148]。第二代架构里面，FPGA 之间的网

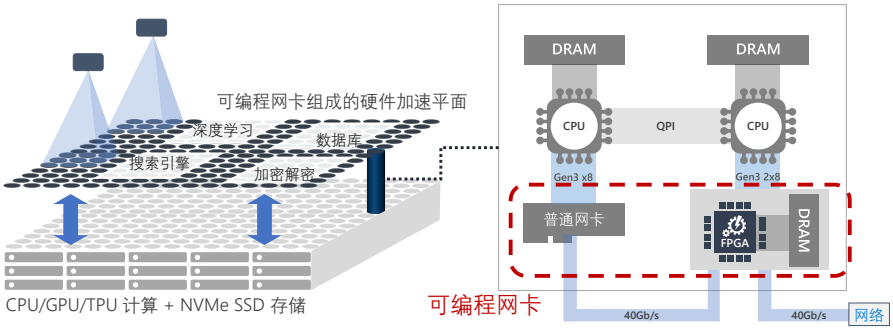


图 2.12　微软基于 FPGA 的可编程网卡。

络连接局限于同一个机架以内，FPGA 之间专网互连的方式很难扩大规模，通过 CPU 来转发则开销太高。

第三代架构中，FPGA 之间通过轻量级流控协议（Lightweight Transport Layer，LTL）通信。同一机架内延迟在 3 微秒以内；8 微秒以内可达 1000 块 FPGA；20 微秒可达同一数据中心的所有 FPGA。第二代架构尽管 8 台机器以内的延迟更低，但只能通过网络访问 48 块 FPGA。为了支持大范围的 FPGA 间通信，第三代架构中的 LTL 还支持 PFC 流控协议和 DCQCN 拥塞控制协议。通过高带宽、低延迟的网络互连的 FPGA 构成了介于网络交换层和传统服务器软件之间的数据中心加速平面。除了每台提供云服务的服务器都需要的网络和存储虚拟化加速，FPGA 上的剩余资源还可以用来加速必应搜索、深度神经网络（DNN）等计算任务。

在 NSDI'18 会议上，微软发表了自 2015 年起将基于 FPGA 的可编程网卡用于网络虚拟化加速的实践[10]。利用可编程网卡，Azure 虚拟机之间的吞吐量可高达 31 Gbps。同期，谷歌云平台（Google Cloud Platform，GCP）基于主机 CPU 软件的网络虚拟化实现[149] 只能达到 16 Gbps 的吞吐量。亚马逊 AWS 基于通用处理器的 Nitro 网络虚拟化加速[150] 只能达到 23 Gbps 的吞吐量。由于通用处理器单核性能的限制，AWS 的单个 TCP 流吞吐量只能达到 10 Gbps，而微软 Azure 和谷歌云平台的单流吞吐量均可达到虚拟机的峰值。这与本文第 2.3 节关于可编程网卡架构的讨论相符。

在延迟方面，使用 Linux 操作系统 TCP/IP 协议栈，微软 Azure 达到了 10 μs 的平均虚拟机间延迟，而使用内核绕过的 DPDK[14] 可以达到 5 μs 的平均延迟。同期，谷歌云平台的平均延迟为 20 μs，亚马逊 AWS 的平均延迟为 28 μs。使用 FPGA 加速后，微软 Azure 的尾延迟比平均延迟优势更加明显。例如，在 99.9% 分位，Azure 的延迟为 20 μs，谷歌云平台基于主机 CPU 的延迟为 75 μs，亚马逊 AWS 基于可编程网卡通用处理器的延迟为 32 μs。在 99.99% 分位，Azure 的延迟为 25 μs，而谷歌云平台和亚马逊 AWS 均达到或接近 100 μs。这是由于硬件流水线比软件处理的延迟更加可控，而软件处理中，由于主机 CPU 上运行着

客户虚拟机，延迟不稳定性比可编程网卡上的通用处理器更高。

2.4.2　亚马逊 AWS 云

在 2017 年 12 月的 Re:Invent 大会上，亚马逊 AWS 云发布了名为 “Nitro” 的计算加速架构[151]。根据 2017 年 Re:Invent 大会和 2018 年 AWS 峰会上亚马逊发布的信息[150,152]，AWS 使用了定制 ASIC 来实现多种加速和安全功能。最初，AWS 在 FPGA 和 ASIC 架构之间权衡，并决定采用 ASIC 方案。为此，2015 年 1 月，亚马逊用 30 多亿美元收购了 ASIC 设计公司 Annapurna 实验室[153]，该公司以设计基于 ARM 核的片上系统（SoC）见长。

Nitro 项目的发展是分阶段的。与微软 Azure 类似，虚拟机的 I/O 瓶颈最早体现在虚拟网络上。早在 2013 年 11 月，AWS 的 C3 实例就引入了一块独立的网卡以实现高性能网络（enhanced networking），采用 SR-IOV 方式让虚拟机直接访问网卡，绕过虚拟机监控器中的虚拟交换机软件。此技术帮助 Netflix 实现了每秒 200 万个数据包的虚拟机网络吞吐量[154]。

2015 年 1 月，AWS 的 C4 实例开始使用硬件加速弹性块存储（Elastic Block Storage，EBS）。弹性块存储的数据储存在存储节点上，而客户虚拟机运行在计算节点上，因此这是一种远程存储。对客户虚拟机而言，是一块虚拟存储设备，它是由虚拟机监控器 Xen Dom0 中的存储管理软件实现虚拟化的。C4 实例使用高性能网卡而非传统网卡来连接远程的弹性块存储，从而提高了性能。

2017 年 2 月，AWS 的 I3 实例引入了 NVMe 本地存储和专用的存储虚拟化芯片。以往，客户虚拟机访问本地存储，也需要经过虚拟机监控器中的存储管理软件，这是由于一台物理服务器中可能有多台虚拟机，每台虚拟机只能访问属于自己的那一部分存储空间，因此需要隔离。对延迟和吞吐量都很高的 NVMe 存储而言，存储虚拟化软件带来的开销太高了。为此，I3 实例引入的 Nitro 芯片在硬件上实现了存储隔离，因此可以通过 SR-IOV 把 NVMe 存储直通客户虚拟机，实现了每秒 300 万次 I/O 操作的存储性能[155]。

2017 年 11 月，AWS 的 C5 实例大幅改变了计算节点虚拟化的架构。首先，C4 实例中的远程存储仍然需要软件实现虚拟化，这一部分也可以像 I3 本地存储一样用硬件实现，不过块存储比本地存储的接口更复杂，因此硬件实现的难度更大。其次，在网络、远程和本地存储都已经使用硬件虚拟化后，事实上虚拟机监控器中的管理软件就只剩下数据平面的中断（APIC）功能和控制平面的管理功能了。控制平面的管理功能较为复杂，用纯数字逻辑显然是不现实的。为了把虚拟网络（VPC）、弹性块存储和虚拟化控制平面全部卸载（offload）到加速卡上，Nitro ASIC 采用了基于 ARM 核的片上系统架构，从而保持了数据平面的可编程性和灵活性，还能把控制平面一并卸载到加速卡上。

采用 Nitro 加速卡后，AWS 重新设计了一个轻量级的虚拟机监控器 Nitro 来取代 Xen，而原来运行在 Xen Dom0 上的控制平面转移到了 Nitro ASIC 里，客户虚拟机可以得到接近裸金属（bare-metal）主机的性能。此后，AWS 发布了裸金属实例，客户代码直接在物理机上运行，而所有的存储和网络资源都由 Nitro 卡提供。

Nitro 系列芯片主要包括三种芯片[150-152]：

1. 云网络（VPC）和弹性块存储（EBS）加速芯片，一边连接数据中心网络，一边以 PCIe 卡的形式连接 CPU；
2. 本地 NVMe 存储虚拟化芯片，作为 CPU 和 NVMe 存储设备之间的代理；
3. 安全芯片，用于验证服务器中各种设备固件的版本，以及在裸金属服务器切换租户时重刷固件清除痕迹。

Nitro 芯片的作用可以分为降低成本、提高性能和提高安全性三方面。下面详细讨论。

节约 CPU 核。网络和存储虚拟化需要占用大量的 CPU 资源来处理每个网络包和存储 I/O 请求。根据 ClickNP[156] 估计，每个客户虚拟机的 CPU 核，需要预留另外 0.2 个 CPU 核来实现虚拟化。如果这些功能可以被卸载到专用硬件，所节省的 CPU 核就可以用来安装客户虚拟机。不管是考虑公有云虚拟机上每个 CPU 核的售价，还是考虑 Xeon CPU 每个核的硬件成本，用专用硬件都能获得明显的成本节约[10]。

提高最大核数。节约 CPU 核不仅能够降低成本，还能够提高大型虚拟机实例的最大核数。因为各大公有云厂商都从 Intel 等相同的厂商购买 CPU，因此同一时期能买到的最大 CPU 核数是相对固定的。虚拟化被卸载到硬件后，所有 CPU 核都用于运行客户虚拟机，因此 AWS 的 M5 实例最多可达 96 个 CPU 核。如果不使用硬件卸载，将只有 80 个 CPU 核可用于客户虚拟机，从而降低对追求极致性能客户的吸引力。

提高单核频率。由于功耗墙的限制，CPU 的核心数目与平均核心频率不可兼得。同一代 CPU 架构下，较高核心频率的 CPU 核数一般较少。对于核数相等的虚拟机实例，如果使用传统的软件虚拟化，物理机就需要 1.2 倍的 CPU 核数，从而平均核心频率就可能降低。例如，在 C5 实例推出前的 72 核 EC2 实例，CPU 基频为 2.7 GHz，但采用同一代 Skylake 架构的 C5 实例虚拟机，CPU 基频就可以达到 3.0 GHz。

提高本地存储性能。首先，在裸金属服务器上，本地 NVMe 存储可以达到每盘高达 400 K IOPS（I/O 操作每秒）的吞吐量。AWS I3 实例有 8 块 NVMe SSD，达到 3 M IOPS 的吞吐量。而对于常见的存储虚拟化协议栈，每个 CPU 核只能处理 100 K IOPS 左右的吞吐量，这意味着要占用 30 个 CPU 核才能让虚拟机充分

利用 NVMe 存储的吞吐量，这个开销太高了。即使只有一块 NVMe 存储，4 个 CPU 核之间的负载均衡仍然是个难题[130]。如第 2.3 节所讨论的，由于硬件分配和处理任务是流水线式而非多个处理单元简单并行，硬件能够比多核软件更好地保证服务质量（QoS）。

其次，在延迟方面，裸金属服务器的 NVMe 存储平均延迟约为 80 微秒。虚拟化软件不仅会增加 20 微秒的平均延迟，而且由于操作系统调度、中断、缓存不命中等因素的影响，在高负载下的尾延迟（tail latency）可高达 1 毫秒（1000 微秒）。采用硬件卸载可以降低平均延迟 20%，并降低高负载下的尾延迟 90% 以上。

提高远程存储性能和安全性。与本地存储类似，硬件加速可以提高远程存储性能。还有一点额外的优势：远程存储被公有云的所有租户共享，对可靠性和安全性要求更高。传统上远程存储协议在虚拟机监控器中运行，虽然逻辑上与客户虚拟机隔离，但由于共享 CPU、内存等资源，仍然不能排除零日（0-day）漏洞和边信道攻击的潜在安全隐患。把远程存储协议从主机 CPU 卸载到 Nitro 卡后，就有了更高的隔离性和更小的攻击表面（attack surface）。

提高网络性能和安全性。在 Nitro 卡上实现网络虚拟化后，虚拟机可以直接通过 SR-IOV 访问网卡，达到数据中心网络 25 Gbps 的理论上限，尤其是对小数据包应用场景的性能提升明显。根据 ClickNP[156] 估计，对 25 Gbps 线速（line-rate）的 64 字节小数据包，每秒高达 37 百万个，如果用软件虚拟交换机处理，将需要 60 个 CPU 核，这显然是不可接受的。因此大多数云服务商对虚拟网络（VPC）对吞吐量不仅有字节数的限制，还有数据包数的限制。大多数公有云的虚拟网络只支持数百万数据包每秒的吞吐量，需要处理大量小请求的远程过程调用（RPC）、键值存储（KVS）服务器就会遇到性能瓶颈。在延迟方面，软件虚拟化的 AWS 端到端延迟可达 100 微秒以上，而使用 Nitro 虚拟化加速后延迟就降低到 50 微秒以内了。硬件虚拟化对虚拟网络的尾延迟、服务质量和安全性的提升，也与远程存储相似。

提高裸金属服务器安全性。最后，裸金属服务器上的客户代码可以直接访问服务器内的各种硬件设备，甚至可能烧写带外服务器管理（BMC）等组件的固件[157]。如果固件内被嵌入了恶意代码，并在下一个租户使用该裸金属服务器时被激活，后果不堪设想。事实上很大一部分客户选择裸金属服务器，正是出于对虚拟机隔离性的担忧。为了在租户开始使用裸金属服务器时提供安全和一致的硬件环境，Nitro 安全芯片会对固件进行重烧写。Nitro 还会在系统启动时进行完整性检查，这类似 UEFI 可信启动技术，但验证的范围不仅包括操作系统引导器，还包括硬件固件。

2.4.3　阿里云、腾讯云、华为云、百度

2018 年，我国云计算服务商也积极地在数据中心部署了可编程网卡。阿里云和腾讯云部署可编程网卡的首要目的是支持裸金属（bare-metal）服务器。相比虚拟机，裸金属服务器可以消除虚拟化带来的开销，实现同样硬件条件下最高的性能；方便部署客户自己的虚拟化软件（如 VMWare）；与客户在自有数据中心（on-premises）的部署环境完全相同，降低客户上云的迁移开销；方便使用不支持虚拟化或虚拟化后性能有较大损失的硬件，如 GPU 和 RDMA 网卡；不与其他租户共享服务器硬件，隔离性和安全性更强，也能符合一些客户的合规要求。

在公有云中使用裸金属服务器的主要技术挑战是访问数据中心内的虚拟网络（VPC）和远程存储（EBS）等资源。一种简单的方法是在同一个机架（rack）内放置若干虚拟网络和存储服务器，部署相应的软件；并在柜顶交换机（ToR switch）上配置转发规则，使裸金属服务器的所有网络数据包经过虚拟网络和存储服务器。这种方法需要增加额外的服务器资源，提高了成本。另一种方法是把虚拟网络和存储的数据平面卸载到柜顶交换机上。但是，柜顶交换机的编程灵活性一般较差[158]，不足以支持虚拟存储的应用层协议和虚拟网络的安全规则等。

因此，在服务器上增加一块可编程网卡便成为支持裸金属服务器性能最高的方案。阿里和腾讯采用了 FPGA 数据平面与多核 CPU 控制平面结合的 SoC 方案。2018 年，阿里云发布的 “神龙” 裸金属服务器使用自研的 MOC 卡[159-160] 实现类似 AWS Nitro 的网络、存储虚拟化。腾讯云在 APNet’18 上发布了基于 FPGA 的可编程网卡方案，主要用于网络虚拟化[158]。腾讯仅用 10 个硬件工程师，就在三个月内完成了 FPGA 逻辑设计，用四个月做出了可编程网卡的板卡，并在一年内部署[158]。这是 FPGA 编程也可以实现敏捷开发的例证。腾讯正在把可编程网卡的应用范围从裸金属服务器扩展到普通虚拟机，并对两种应用场景使用统一的可编程网卡架构。

华为基于海思半导体的技术积累，发布了两款可编程网卡，同时也用于华为云虚拟网络的加速。SD100 系列可编程网卡[161] 采用了基于多核 ARM64 CPU 的 SoC 架构，数据平面和控制平面都运行在 ARM CPU 上。IN5500 系列可编程网卡[162] 采用网络处理器（NP）提供数据平面的可编程性，可达到 100 Gbps 性能。利用可编程网卡，华为云发布了 C3ne 网络增强虚拟机实例，在国内云厂商中率先实现了每秒千万级的数据包转发[163]。

百度虽然尚未发布可编程网卡加速的虚拟机实例，但在 FPGA 加速数据中心计算密集型应用方面是先行者。早在 2010 年，百度就将 FPGA 用于数据压缩[134]。2014 年，百度发布了将 FPGA 用于深度学习推理的 SDA 框架[136]，随后将 FPGA 用于数据库 SQL 处理[135]。2017 年，百度提出了基于 FPGA 的数据中

心全栈加速器 XPU，将 FPGA 用于各种计算加速场景[137]。

第 3 章　系 统 架 构

本文提出一个基于可编程网卡的高性能数据中心系统架构。如图 3.1 所示，本文把普通网卡升级为可编程网卡，将虚拟化、网络和存储功能、操作系统中需要高性能的数据平面卸载到可编程网卡，以降低“数据中心税”，让 CPU 集中精力于客户的应用程序。

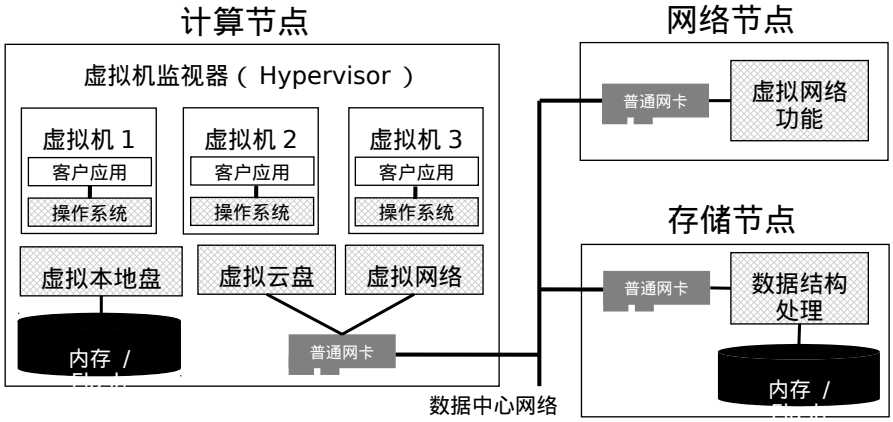


图 3.1　回顾：虚拟化的数据中心架构。

第 1 章已经讨论过，虚拟化的数据中心主要可以分为计算、网络、存储节点。在网络和存储节点，采用控制面与数据面分离的设计思想。数据面是操作相对频繁、逻辑相对简单的处理，而控制面是操作相对不频繁、逻辑相对复杂的处理。在可编程网卡中实现数据面，在主机 CPU 上实现控制面，实现了数据面完全不经过主机 CPU。这包括第 4 章的虚拟网络功能和第 5 章的数据结构处理。加速虚拟网络功能和远程数据结构访问也是本文最重要的创新。

在计算节点，亦即客户虚拟机所在的服务器主机上，用可编程网卡实现虚拟机监控器（hypervior）的虚拟化功能和操作系统原语。虚拟化分为“一虚多”和“多虚一”两个方面。“一虚多”，即可编程网卡把计算节点内的硬件资源虚拟化成多个逻辑资源，实现其他计算节点和本地多台虚拟机的多路复用。例如，第 4 章的 ClickNP 将硬件网卡和网络链路虚拟化为每个虚拟机一张虚拟网卡；第 5 章的 KV-Direct 实现了多个客户端并发访问共享键值存储，并能保证一致性。“多虚一”，即可编程网卡把数据中心内物理上分散的资源虚拟化成一个逻辑资源，实现逻辑资源到物理资源的映射和路由。例如，第 4 章的 ClickNP 将数据中心内网络功能虚拟化成逻辑上统一的网络功能；第 5 章的 KV-Direct 客户端将分布式键值存储虚拟化成逻辑上统一的键值映射；还可以实现存储和内存的解聚（disaggregation）。为了加速操作系统原语并控制硬件的复杂度，把操作系统原语划分为可编程网卡上的可靠通信协议和主机 CPU 上运行的用户态库、用户态管

理程序，如第 6 章 SocksDirect 实现的套接字通信原语。

图 3.2 显示了基于可编程网卡的系统总体架构。下面将按照网络、存储和高层抽象的顺序，简要介绍本文后续各章的总体设计。

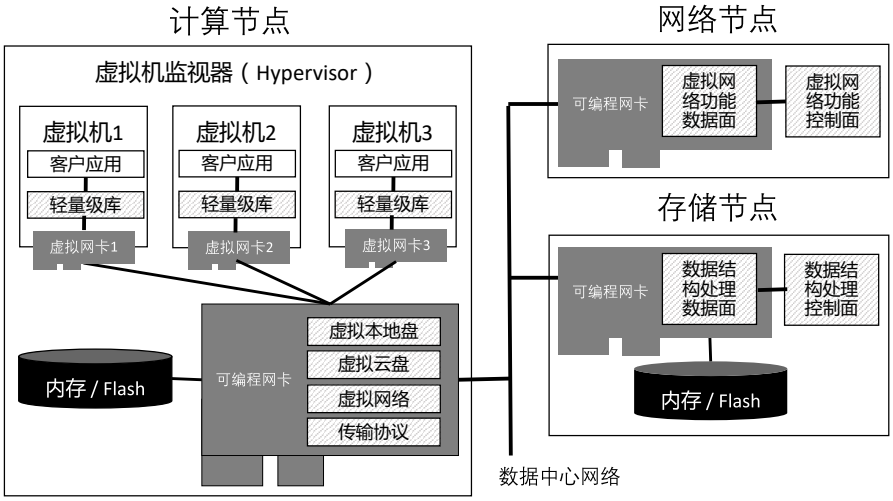


图 3.2　基于可编程网卡的数据中心系统总体架构。

3.1　网络加速

3.1.1　网络虚拟化加速

从第 1 章的传统数据中心架构（图 1.1）开始，本文逐步把“数据中心税”消除或者卸载（offload）到可编程网卡上。如图 3.3 所示，第一步是用可编程网卡替代原有的普通网卡，并把软件实现的虚拟网络卸载到可编程网卡上。

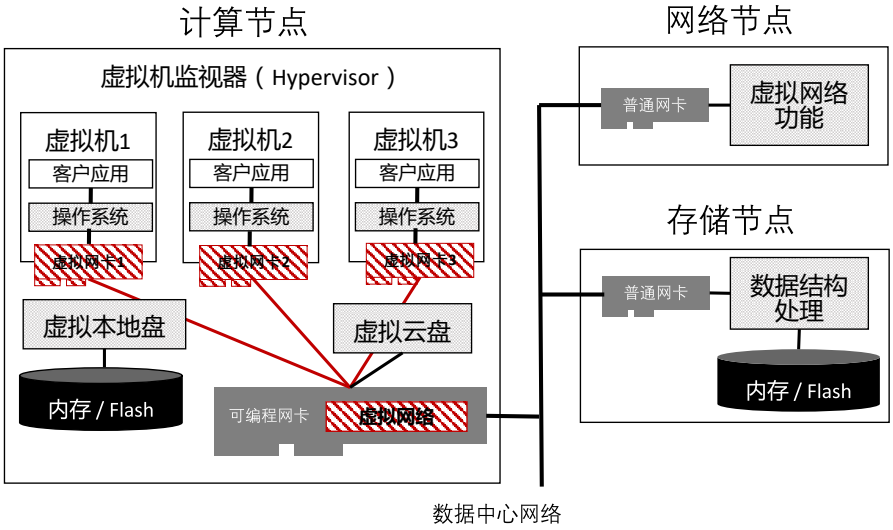


图 3.3　用可编程网卡加速虚拟网络后的架构。

为了让虚拟机上的操作系统网络协议栈能够使用虚拟网络收发数据包，可编程网卡利用 SR-IOV（Single Root I/O Virtualization）技术[164]，虚拟化成多个

PCIe 虚拟设备（VF，Virtual Function），并给每台虚拟机分配一个 PCIe 虚拟设备。虚拟机内原有的虚拟网卡驱动程序（如基于 virtio 技术[165] 的）需要替换为本文实现的 FPGA 驱动程序和基于 FPGA 的虚拟网卡驱动程序。第 4 章的 ClickNP 将硬件网卡和网络链路虚拟化为多个租户的虚拟网络。

如果可以绕过虚拟机操作系统的网络协议栈，直接替换虚拟机上应用程序所使用的标准库（即系统调用接口 libc），就不必实现 SR-IOV 硬件虚拟化。第 6 章的 SocksDirect 通过截获应用程序关于网络套接字的标准库调用，在用户态实现了容器覆盖网络（container overlay network），即适用于容器的虚拟网络。为了用户态运行库与可编程网卡间的高效通信，在虚拟机内安装 FPGA 驱动程序，将可编程网卡的 PCIe 地址空间的一部分映射到用户态，从而绕过了虚拟机内核和虚拟机监视器（Virtual Machine Monitor 或 Hypervisor）。

3.1.2　网络功能加速

如图 3.4 所示，第二步是在网络节点上，把软件实现的虚拟网络功能划分为数据面和用户面，并把数据面卸载到可编程网卡中。需要注意的是，网络节点和计算节点的划分是逻辑上的。有可能虚拟网络功能被编排到与虚拟机相同的服务器主机上，这时网络节点和计算节点的功能就合二为一了，虚拟网络和虚拟网络功能之间的连接也从数据中心网络简化成了可编程网卡内模块间的连接。

来自源计算节点（或上一个网络节点）的数据包被网络节点的可编程网卡接收之后，在网卡内的数据面进行处理，大多数情况下不需要 CPU 上的控制面介入，就可以把处理后的数据包再发送回数据中心网络，到达目的计算节点（或下一个网络节点）。第 4 章将介绍如何用高级语言模块化编程实现网络功能，实现 FPGA 数据面与 CPU 控制面的协同处理。

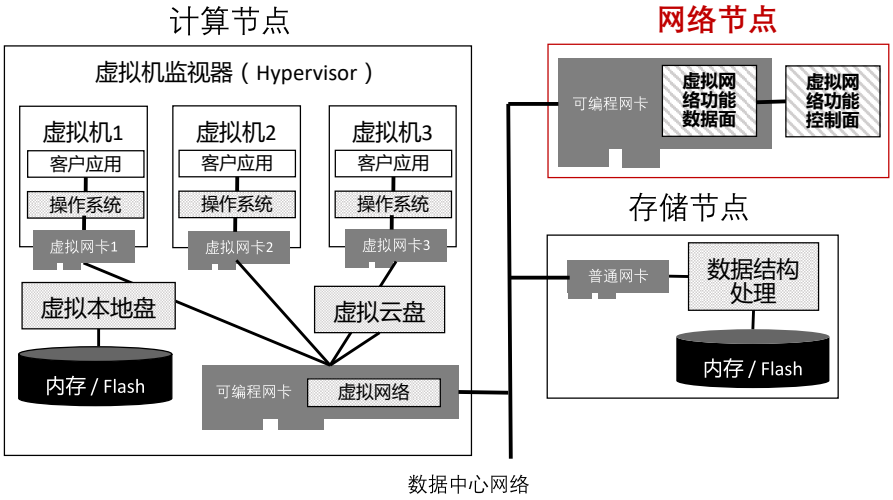


图 3.4　用可编程网卡加速网络功能后的架构。

3.2　存储加速

3.2.1　存储虚拟化加速

在网络加速之后，第三、四步是存储加速。作为第三步，首先将计算节点的存储虚拟化功能卸载到可编程网卡中，如图 3.5 所示[①]。

为了支持多个存储节点组成的分布式存储，虚拟云存储服务需要把逻辑地址映射到存储节点地址。例如，在第 4 章的分布式键值存储中，客户端需要根据一致性哈希（consistent hashing）[166]，把键（key）映射到存储节点，再把请求路由到该节点。

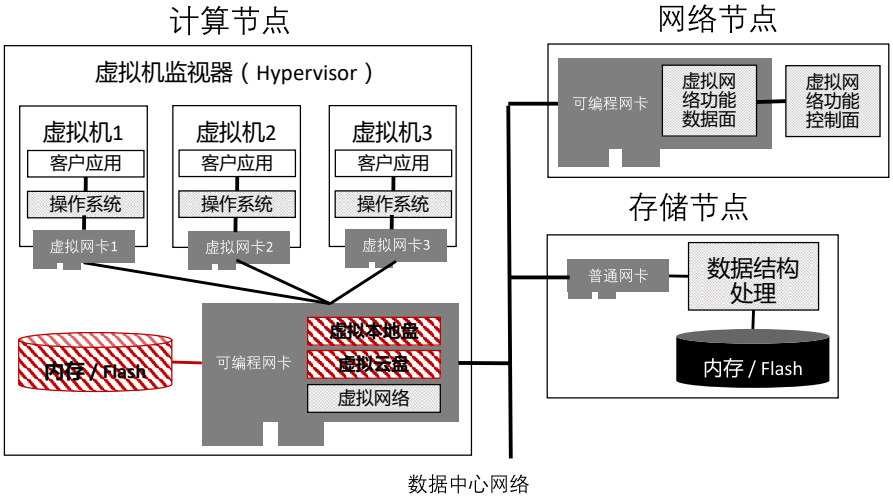


图 3.5　用可编程网卡加速本地存储和云存储后的架构。

3.2.2　数据结构处理加速

第四步，将存储节点上的数据结构处理卸载（offload）到可编程网卡。以第 5 章将详细介绍的键值存储为例。存储节点上的可编程网卡从网络上收到查询（GET）或写入（PUT）某个键（key）的请求后，要从本地的内存或闪存中查询出对应的键值对，处理请求后把结果发送给网络上的请求方。如图 3.6，这个过程称为数据结构处理的数据面，通常不需要控制面介入。然而，由于可编程网卡上不适合运行复杂的逻辑，把内存分配器分为网卡和主机 CPU 两部分。网卡上缓存若干固定大小的空闲内存块。当空闲内存块不足时，需要主机 CPU 上的控制面通过拆分更大的内存块来补充空闲内存块；当空闲内存块过多时，又需要主机 CPU 来进行垃圾回收，合并成更大的内存块。通过网卡直接访问内存数据结构的另一个挑战是网卡与内存之间的 PCIe 带宽较低、延迟较高。为此，第 5 章设计了一系列优化方法来节约带宽、通过并发处理来隐藏延迟。尽管请求是并发处理的，第 5 章的设计还能保证多个客户端并发访问的强一致性，即请求在逻辑

[①]本文没有在存储虚拟化方面做出贡献，包含在系统架构中是为了完整性。

上按照网络接收的顺序被依次处理。如果存储节点同时也作为计算节点运行虚拟机，为了解决本地和远程访问同一块存储区域时的一致性问题，不管是本地还是远程访问，都经过可编程网卡处理。

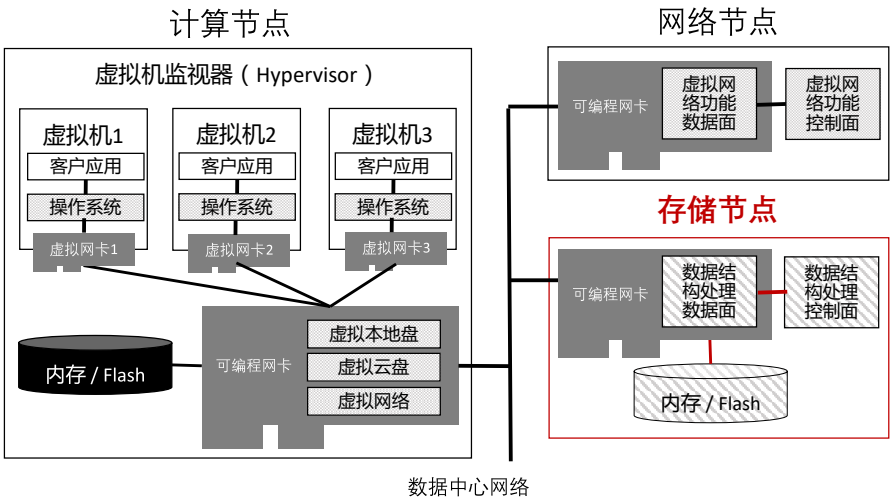


图 3.6　用可编程网卡加速数据结构处理后的架构。

3.3　操作系统加速

最后一步，将操作系统中需要高性能的功能拆分为三部分，分别在可编程网卡、主机 CPU 的用户态运行库和主机 CPU 的用户态守护进程（daemon）中处理。如图 3.7，操作系统在图中被替换成了用户态运行库，而可编程网卡中增加了传输协议的功能。用户态运行库通过替换标准库（如 libc），截获了应用程序的系统调用，从而可以在用户态实现操作系统功能的一部分，而把另一部分功能卸载到可编程网卡。用户态守护进程主要用于控制面操作，为简化起见，图 3.7 中没有画出。

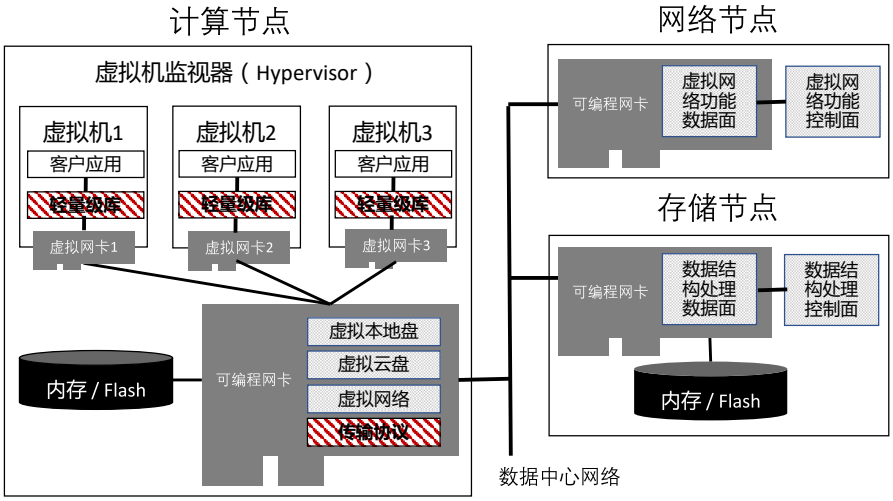


图 3.7　用可编程网卡加速操作系统通信原语后的架构。

操作系统中包括通信、存储等子系统，本文以通信系统的加速为例。套接字是应用程序最常用的通信原语，但由于操作系统的多种开销，其性能不令人满意。第 6 章设计并实现了一个高性能的用户态套接字系统，与现有应用程序完全兼容，并且对主机内进程间通信和跨主机的通信都能达到接近硬件极限的低延迟和高吞吐量。系统由可编程网卡上的可靠通信协议和主机 CPU 上运行的用户态库、用户态守护进程三部分构成。对于跨主机的通信，数据面由可编程网卡和用户态库构成，可编程网卡负责多路复用和可靠传输等低层语义，提供远程直接内存访问（RDMA）原语；用户态库负责把 RDMA 原语封装成 Linux 虚拟文件系统（Virtual File System）的套接字语义，提供无锁消息队列、缓冲区管理、等待事件、零拷贝内存页面重映射等高层语义。对于主机内的通信，数据面由 CPU 的硬件内存一致性协议（coherence protocol）和用户态库构成。用户态库在进程间建立共享内存队列，并依靠 CPU 的内存一致性协议自动同步。用户态库的功能与跨主机通信类似。用户态守护进程负责控制面，即初始化、进程创建和退出、连接建立和关闭、与 RDMA 网卡建立队列、在进程之间建立共享内存队列等。

在第 6 章的设计中，客户应用程序通过 SocksDirect 运行库，直接访问可编程网卡中的 RDMA 功能，不需要经过操作系统内核和虚拟机监控器，可编程网卡也就不需要支持 SR-IOV 硬件虚拟化。

3.4　可编程网卡

在结束基于可编程网卡的数据中心系统架构介绍后，本节介绍可编程网卡内部的软硬件架构。如图 3.8 所示，可编程网卡内的逻辑主要由第 4 章的编程框架、第 5 章的基础服务中间件以及第 5 章和第 6 章的应用层构成。主机 CPU 上配套的软件包括第 4 章的 FPGA 通信库和驱动程序、第 5 章的键值操作库和第 6 章兼容 Linux 操作系统的套接字通信库。

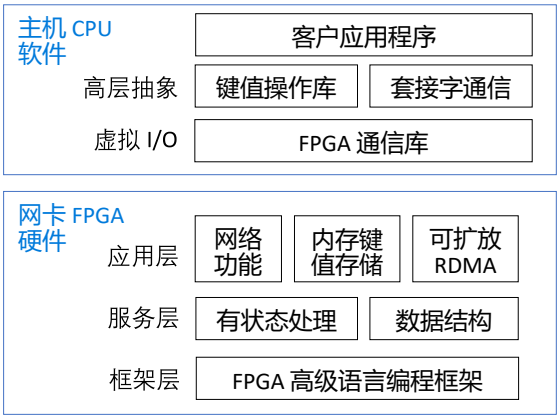


图 3.8　软硬件协同设计的可编程网卡架构。

图 3.9 显示了本文使用的 Catapult 可编程网卡[48] 硬件架构。Catapult 可编程网卡由 Stratix V FPGA 和 Mellanox ConnectX-3 商用网卡构成。FPGA 有两个连接 40 Gbps 以太网络的 QSFP 接口，一个连接数据中心交换机，另一个连接可编程网卡内的商用网卡。由于本文使用的 FPGA 没有 PCIe Gen3 x16 硬核，FPGA 与主机之间通过两个 PCIe Gen3 x8 接口连接，它们共享一个 PCIe Gen3 x16 物理插槽。商用网卡有两个 40 Gbps 以太网接口，一个连接 FPGA，另一个闲置。商用网卡与主机之间通过 PCIe Gen3 x16 接口连接。

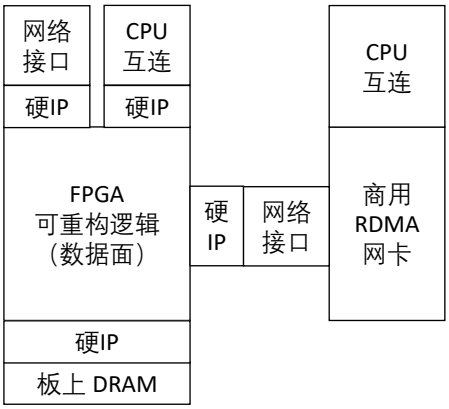


图 3.9　本文使用的 Catapult 可编程网卡。

本文使用的 FPGA 有 172,600 个逻辑元件（ALM），2,014 个 20 Kb 大小的 M20K 片上内存（BRAM）以及 1,590 个可以执行 16 位乘法的数字信号处理单元（DSP）。FPGA 板上还有 4 GB 大小的 DRAM，通过一个 DDR3 通道与 FPGA 相连。

本文第 4 章和第 5 章使用其中的 FPGA 可重构逻辑来实现网络功能和数据结构处理；第 6 章使用其中的商用 RDMA 网卡实现套接字原语中的硬件传输协议部分。来自数据中心网络的数据包从 FPGA 左上角的网络接口被可编程网卡接收。如果它是网络功能或者数据结构处理需求，就直接在 FPGA 可重构逻辑中进行处理，处理过程中 FPGA 需要使用板上 DRAM 和通过 CPU 互连（如 PCIe）访问主机 DRAM。如果数据包是用于第 6 章中的套接字通信，它将在 FPGA 中进行虚拟网络的解封装，并通过 FPGA 与商用 RDMA 网卡之间的网络接口发送给商用 RDMA 网卡。商用 RDMA 网卡会把数据包的内容通过 CPU 互连（如 PCIe）DMA 发送给主机上的用户态套接字库。

接下来的三章将依次详细讨论本文的三个主要创新点，即基于可编程网卡的网络功能（ClickNP）、存储数据结构（KV-Direct）和操作系统通信原语（SocksDirect）加速。

第 4 章　ClickNP 网络功能加速

4.1　引言

本章的主题是网络虚拟化和网络功能加速，在本文中的位置如图 4.1 所示。

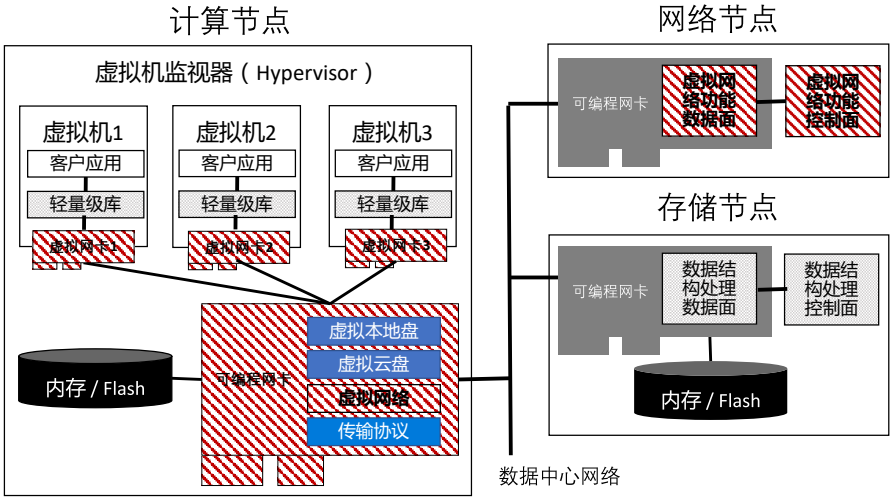


图 4.1　本章主题：网络虚拟化和网络功能加速，用粗斜线背景的方框标出。

本章作为全文的基础，将提出一个 FPGA 高级语言编程框架，以及主机 CPU 上相应的运行时，并在此基础上实现硬件加速的网络功能，如图 4.2 所示。

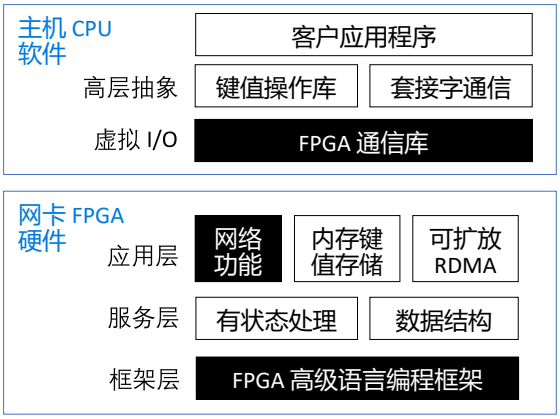


图 4.2　本章在可编程网卡软硬件架构中的位置。

本章介绍 ClickNP，一个用于在商用服务器上进行高度灵活和高性能的网络功能处理的 FPGA 加速平台。ClickNP 通过三个步骤解决 FPGA 的编程挑战。首先，提供了一个模块化架构，类似于第 2.2.2 节介绍的 Click 模型[116]，复杂的网络功能可以使用简单的元件组成。① 其次，ClickNP 元件是用高级 C 语言编写的，并且是跨平台的。ClickNP 元件可以通过利用商业高层次综合（High-Level

①这也是系统名称 *Click Network Processor*（ClickNP）的来源。

Synthesis，HLS）工具[49,57-58] 在 FPGA 上编译成硬件描述语言和硬件模块，或在 CPU 上使用标准 C++ 编译器编译成机器指令。最后，高性能 PCIE I/O 通道可在 CPU 和 FPGA 上运行的元件之间提供高吞吐量和低延迟通信。PCIE I/O 通道不仅可以实现 CPU-FPGA 的联合处理 – 允许程序员自由地在 CPU 和 FPGA 间划分任务，而且对调试也有很大的帮助，因为程序员可以在主机上轻松地运行有问题的元件，并使用熟悉的软件调试工具。

ClickNP 使用一系列优化技术来有效地利用 FPGA 中的大规模并行性。首先，ClickNP 将每个元件组织成 FPGA 中的逻辑块，并将它们与先进先出（FIFO）缓冲区连接起来。因此，所有这些元件块都可以完全并行运行。对于每个元件，本章仔细编写处理函数以最小化操作之间的依赖关系，从而允许高层次综合工具生成最大并行逻辑。此外，开发了*延迟写入*和*内存散射*技术来解决读写依赖性和伪内存依赖性，这些问题是现有高层次综合工具无法解决的。最后，通过在不同阶段仔细平衡操作并匹配其处理速度，可以最大化管道的总体吞吐量。通过所有这些优化，ClickNP 实现了每秒高达 2 亿个数据包的数据包吞吐量 ①，并具有超低延迟（对大多数数据包大小，延迟小于 2μs）。与 GPU 和 CPU 上最先进的软件网络功能相比，这大约是 10 倍和 2.5 倍的吞吐量增益[36]，同时将延迟分别降低 10 倍和 100 倍。

本章实现了 ClickNP 工具链，它可以与各种商业高层次综合工具集成[49,57]，包括 Intel FPGA OpenCL SDK 和 Xilinx SDAccel。本章还实现了大约 200 个常用元件，其中 20% 与 Click 中的对应元件功能相同，参考 Click 的代码重新实现。本章将使用这些元件构建五个演示网络功能：（1）高速数据包发包和抓包工具，（2）支持精确匹配和通配符匹配的防火墙，（3）IPSec 网关，（4）一个可以处理 3200 万个并发流的四层负载均衡器，（5）pFabric 调度器[167] 执行严格优先级流调度（flow scheduling），具有 40 亿个优先级。评估结果表明，所有这些网络功能都可以通过 FPGA 大大加速，并且可以在任何数据包大小下使 40Gbps 的线速饱和，同时具有极低的延迟和可忽略的 CPU 开销。

总之，本章的贡献是：（1）ClickNP 语言和工具链的设计与实现；（2）在 FPGA 上高效运行的高性能数据包处理模块的设计和实现；（3）FPGA 加速网络功能的设计和评估。据作者所知，ClickNP 是第一个用于通用网络功能、完全用高级语言编写并能实现 40 Gbps 线速的 FPGA 加速数据包处理平台。

① ClickNP 网络功能的实际吞吐量可能受以太网端口数据速率限制。

4.2　背景

4.2.1　软件虚拟网络与网络功能的性能挑战

虚拟网络通常用虚拟交换机软件实现，如 Open vSwitch[168]。为了提高虚拟网络的性能，并满足可编程性要求，云服务商重新设计了软件虚拟交换机。通过利用 DPDK[14] 等高速网络处理包技术，并使用轮询模式运行 CPU 核，可以绕过 OS 网络协议栈来显著降低数据包处理成本。但是，即使是不做任何处理的简单网络数据包转发，每个 CPU 核每秒也只能处理 10 M 至 20 M 个数据包[117-118]，对于 60 M 个数据包每秒的 40 Gbps 线速，仍然需要 3 至 6 个 CPU 核。

传统的网络功能由部署在数据中心特定位置的专用设备实现，如 F5 负载均衡器[113]。这些专用网络功能设备不仅价格高昂，也不够灵活，不足以支持云服务中的多租户。为了支持灵活的网络功能，云服务商还部署了软件实现的虚拟网络功能。例如，Ananta[7] 是一个部署在微软数据中心的软件负载均衡器，用于提供云规模的负载均衡服务。为了在一台服务器上支持多种网络功能，RouteBricks[8]、xOMB[169] 和 COMB[170] 采用 Click 模块化路由器[116] 的编程模型，把每个网络功能实现成一个 C++ 的类，让每个数据包在一个 CPU 核上依次被各个网络功能处理，即所谓“运行到结束（run-to-completion）”模型。这些工作表明，在实际网络功能下，基于多核 x86 CPU 的每台服务器转发数据包的速度可达 10 Gbps，并且可以通过多核和搭建更多网络节点的集群来扩展容量。为了实现网络功能之间的隔离，NetVM[171]、ClickOS[117]、HyperSwitch[172]、mSwitch[173] 等工作把每个网络功能放在一个（轻量级）虚拟机里，并让虚拟交换机把数据包依次分发给这些虚拟机处理，即所谓“流水线”模型。流水线模型中，数据包在多核之间反复传递，开销较高。NetBricks[118] 回到了“运行到结束”模型，但用高级语言实现网络功能，并在编译器和运行框架的层面上保证网络功能之间的隔离性。NFP[174] 利用多个并行的网络功能流水线提高数据包处理的性能。

虽然软件实现的虚拟交换机和网络功能可以使用更多数量的 CPU 核和更大的网络节点集群来支持更高的性能，但这样做会增加相当大的资产和运营成本[7,9]。云服务商在 IaaS 业务中的盈利能力是客户为虚拟机支付的价格与托管虚拟机的成本之间的差异。由于每台服务器的资产和运营成本在上架部署时就已基本确定，因此降低托管虚拟机成本的最佳方法是在每台计算节点服务器上打包更多的客户虚拟机，并减少网络和存储节点的服务器数量。目前，物理 CPU 核（2 个超线程，即 2 个 vCPU）的售价为 0.1 美元/小时左右，亦即最大潜在收入约为 900 美元/年[10]。在数据中心，服务器通常服役 3 到 5 年，因此一个物理 CPU 核在服务器生命周期内的售价最高可达 4500 美元[10]。即使考虑到总有一部分 CPU 核没有被售出，并且云经常为大客户提供折扣，与专用硬件相比，即

使专门分配一个物理 CPU 核用于虚拟网络也是相当昂贵的。

为了加速虚拟网络和网络功能，以前的工作已经提出使用 GPU[36]，网络处理器（Network Processor，NP)[37-38] 和硬件网络交换机[9]。GPU 早期主要用于图形处理，近年来扩展到具有海量数据并行性的其他应用程序。PacketShader[36] 表明使用 GPU 可以实现 40Gbps 的分组交换速度。GPU 适合批量操作，但是，批量操作会导致高延迟。例如，PacketShader[36] 报告的转发延迟约为 200μs，比 ClickNP 高两个数量级。如第 2.3.4 节讨论的，与 GPU 相比，FPGA 可以充分利用流水线并行、数据并行和请求并行，实现低延迟、高吞吐量的数据包处理。网络处理器专门用于处理网络流量并且具有许多硬连线（hard-wired）的网络加速器。NP-Click[175] 在网络处理器上实现了 Click 编程框架。Click 模块化路由器[116] 提出的第二年，NP-Click[175] 就在网络处理器上实现了 Click 编程框架。如第 2.3.2 节讨论的，网络处理器的主要问题是单流（single flow）吞吐量受单核性能局限。如第 2.2.2 节讨论的，硬件网络交换机的主要问题是灵活性和查找表容量不足[9]。

如第 2.3.4 节讨论的，使用 FPGA 来做网络功能处理存在一系列挑战，本文关注利用大规模并行性和编程工具链两个方面。与 CPU 或 GPU 相比，FPGA 通常具有更低的时钟频率和更小的存储器带宽。例如，FPGA 的典型时钟频率约为 200MHz，比 CPU 慢一个数量级（2 至 3 GHz）。同样，FPGA 的单块存储器或外部 DRAM 的带宽通常为 2 至 10 GBps，而 Intel Xeon CPU 的 DRAM 带宽约为 60 GBps，GPU 的 GDDR5 或 HBM 带宽可高达数百 GBps。但是，CPU 或 GPU 只有有限的内核，这限制了并行性。FPGA 内置了大量的并行性。现代 FPGA 可能拥有数百万个 LE，数百个 K 位寄存器，数十个 M 位 BRAM 和数千个 DSP 模块。从理论上讲，它们中的每一个都可以并行工作。因此，FPGA 芯片内部可能会同时运行数千个并行的 “核”。虽然单个 BRAM 的带宽可能有限，但如果并行访问数千个 BRAM，则总内存带宽可达数 TBps！因此，为了实现高性能，程序员必须充分利用这种大规模的并行性。

传统上，FPGA 使用诸如 Verilog 和 VHDL 之类的硬件描述语言进行编程。这些语言水平太低，难以学习，编程也很复杂。因此，大型软件程序员社区已经远离 FPGA 多年了[54]。为了简化 FPGA 编程，工业界和学术界开发了许多高级综合工具和系统，旨在将高级语言（主要是 C）的程序转换为硬件描述语言。但是，正如下一小节将讨论的，它们都不适合网络功能处理，这是本工作的重点。

4.2.2　基于 FPGA 的网络功能编程

本章的目标是利用 FPGA 加速构建一个多功能，高性能的网络功能平台。这样的平台应满足以下要求。

灵活性。平台应该完全使用高级语言编程。开发人员应当能使用高级抽象和

熟悉的工具编程，就像在多核处理器上编程一样。这是使大多数软件程序员可以使用 FPGA 的必要条件。

模块化。网络功能平台应该支持模块化架构进行数据包处理。以前关于虚拟化网络功能的经验表明，正确的模块化架构可以很好地捕获数据包处理中的许多常见功能[116-117]，使它们易于在各种网络功能中重用。

高性能、低延迟。数据中心的网络功能应该以 40 / 100 Gbps 的线路速率处理大量数据包，具有超低延迟。以前的工作已经显示[176]，即使网络功能添加的几百微秒的延迟也会对服务体验产生负面影响。

支持 CPU/FPGA 联合数据包处理。 FPGA 不是万能药。正如第 2.3.4 节所讨论的，并非所有任务都适用于 FPGA。FPGA 中也无法容纳较大的逻辑。因此，应该支持 CPU 和 FPGA 之间的细粒度处理分离。这需要 CPU 和 FPGA 之间的高性能通信。

FPGA 长期被用于实现网络路由器和交换机。NetFPGA[177] 提出了一个在 FPGA 上实现路由器的开放硬件平台。近年来，FPGA 也被用于加速网络功能[178-179]。早在 Click 模块化路由器[116] 编程框架被提出的第二年，Xilinx 就提出了用 FPGA 实现 Click 的 Cliff[180]，需要硬件开发人员使用硬件描述语言把一个 Click 元件实现为一个硬件模块。此后，CUSP[181] 和 Chimpp[178] 提出了一系列改进，简化了硬件模块的互连，提高了软硬件协同处理、软件仿真的能力。然而，上述工作使用 Verilog、VHDL 等硬件描述语言编程 FPGA。众所周知，硬件描述语言难以调试、编写和修改，给软件人员使用 FPGA 带来了很大挑战。

为了提高 FPGA 的开发效率，FPGA 厂商提供了高层次综合（HLS）工具[49-50]，可以把受限的 C 代码编译成硬件模块。但这些工具只是硬件开发工具链的补充，程序员仍然需要手动将从 C 语言生成的硬件模块插入到硬件描述语言的项目中，且 FPGA 与主机 CPU 之间的通信也需要自行处理。学术界和工业界提出了 Bluespec[51]、Lime[52] 和 Chisel[53] 等高效硬件开发语言[54-56]，但它们需要开发者具有足够的硬件设计知识。Gorilla[179] 为 FPGA 上的数据包交换提出了一种领域特定的高级语言。高层次综合工具和高效硬件开发语言可以提高硬件开发人员的工作效率，但仍然不足以让软件开发人员使用 FPGA。

Click2NetFPGA[182] 利用高层次综合工具，直接将 Click 模块化路由器[116] 的 C++ 代码编译到 FPGA 来提供模块化架构。然而，Click2NetFPGA 的性能的系统设计存在若干瓶颈（例如，内存和数据包 I/O），它们也没有优化代码以确保完全流水线处理，因此比本文达到的性能低两个数量级。此外，Click2NetFPGA 不支持 FPGA / CPU 联合处理，因此无法在数据平面运行时更新配置或读取状态。

近年来，为了让软件开发人员使用 FPGA，FPGA 厂商提出了基于 OpenCL 的编程工具链[57-58]，提供了类似 GPU 的编程模型，如图 4.3 所示。软件开发人

员可以把用 OpenCL 语言编写的核（kernel）卸载到 FPGA 上。

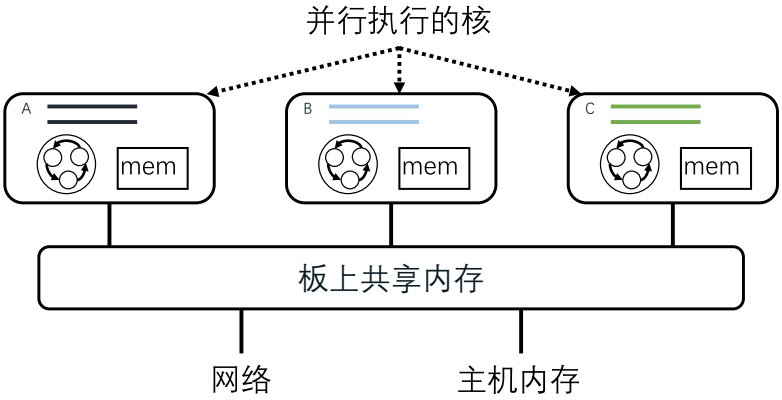


图 4.3　OpenCL 核之间以及核与网络、主机的通信方式：共享内存。

但是，这种方法中多个并行执行的核间需要通过板上共享内存进行通信，而 FPGA 上的 DRAM 吞吐量和延迟都不理想，共享内存还会成为通信瓶颈。其次，FPGA 与 CPU 之间的通信模型是类似 GPU 的批处理模型，主机程序和 FPGA 内核之间的通信必须始终通过板载 DDR 内存。这使得处理延迟较高（约 1 毫秒），不适用于需要微秒级延迟的网络数据包处理。第三，OpenCL 内核函数需要宿主机上的软件程序显式调用。在内核终止之前，主机程序无法控制内核行为，例如设置新参数，也不能读取任何内核状态。但是网络功能面临着连续的数据包流，应该始终在运行。第四，OpenCL 不支持 CPU 和 FPGA 之间的联合数据包处理，CPU 上的数据包处理只能在 OpenCL 框架以外进行。

下文将介绍 ClickNP，一种新颖的 FPGA 加速网络功能平台，满足灵活性、模块化、高性能、低延迟和 CPU/FPGA 联合处理的要求。

网络数据包处理属于流式处理。ST-Accel[183] 指出，在 FPGA 中通过 FIFO 进行流式处理的效率比共享内存更高，可以达到更低的延迟和更高的吞吐量。为此，在 ClickNP 框架内，FPGA 的处理逻辑模块以及网络和主机之间也应当通过 FIFO 管道来通信，如图 4.4 所示。

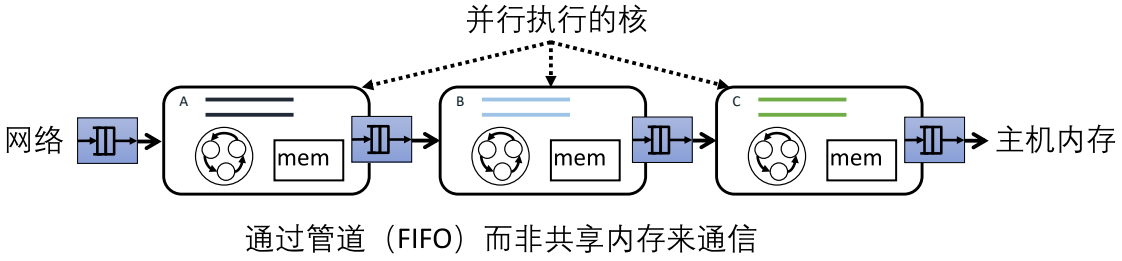


图 4.4　ClickNP 核之间以及核与网络、主机的通信方式：管道（FIFO）。

4.3　系统架构

4.3.1　ClickNP 开发工具链

图 4.5 显示了 ClickNP 的体系结构。ClickNP 基于 Catapult Shell 架构构建[48]。Catapult *shell* 包含许多对所有应用程序通用的可重用逻辑模块。Shell 将它们抽象为一组明确定义的接口，例如，PCIe、直接内存访问（DMA）、DRAM 内存管理单元（MMU）和以太网 MAC。用 ClickNP 编写的 FPGA 程序被编译为 Catapult *用户逻辑*（role）。用户逻辑调用 shell 提供的接口访问外部资源。由于 ClickNP 依赖于商业高层次综合工具链来生成 FPGA 硬件描述语言，需要一个*高层次综合的特定运行时*（Board Specific Package，BSP），用于执行高层次综合特定接口与 shell 接口之间的转换。

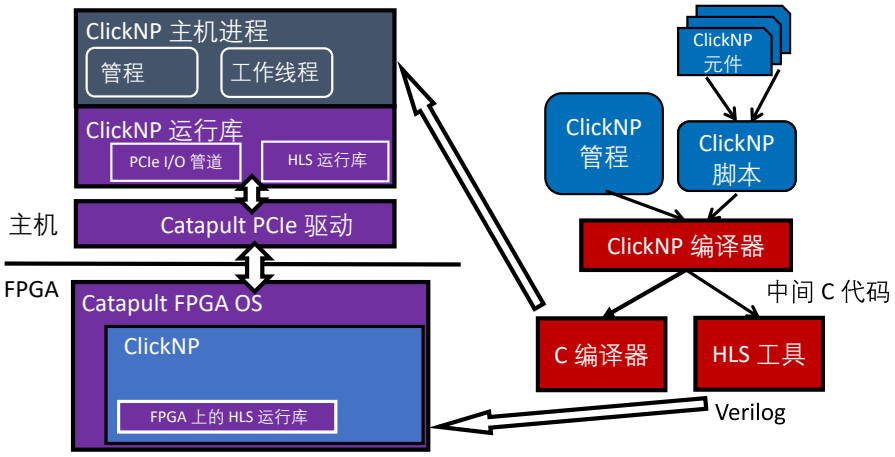


图 4.5　ClickNP 架构。

ClickNP 主机（host）进程通过 ClickNP 运行库与 ClickNP 用户逻辑进行通信，这进一步依赖于 Catapult PCIe 驱动程序中的服务与 FPGA 硬件进行交互。ClickNP 运行库实现了两个重要的功能：（1）它暴露了一个 PCIe 通道 API，以实现 ClickNP host 进程和角色之间的高速和低延迟通信；（2）它调用几个高层次综合特定库将初始参数传递给角色中的模块，并控制这些模块的启动/停止/复位。ClickNP 主机进程有一个管理器线程和零个或多个工作线程。管理器线程将 FPGA 映像加载到硬件中，启动工作线程，根据配置初始化 FPGA 和 CPU 中的 ClickNP 元件，并通过在运行时向元件发送*信号*（signal）来控制它们的行为。如果将每个工作线程分配给 CPU，则每个工作线程可以处理一个或多个模块。

4.3.2　ClickNP 编程

1. 抽象

ClickNP 提供模块化架构，基本处理模块称为*元件*（element）。如图 4.6 所示，ClickNP 元件具有以下属性：

- 本地状态。每个元件都可以定义一组只能在元件内部访问的局部变量。
- 输入和输出端口。元件可以具有任意数量的输入或输出端口。
- 处理程序功能。元件有三个处理函数：（1）初始化处理程序，在元件启动时调用一次，（2）处理处理程序，连续调用以检查输入端口和处理可用数据，以及（3）信号处理程序，它从主机程序中的管理器线程接收和处理命令（信号）。

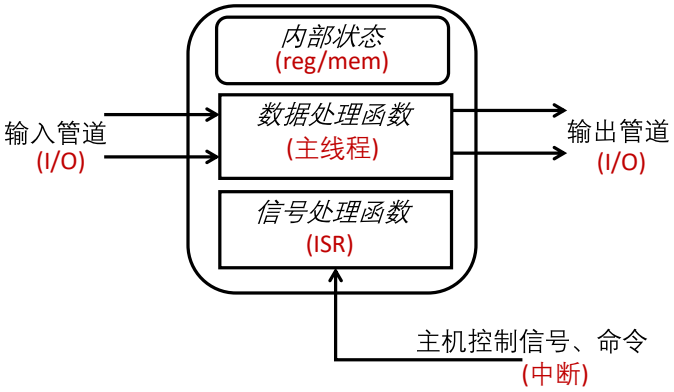


图 4.6　ClickNP 元件的构成。

元件的输出端口可以通过管道（channel）连接到另一个元件的输入端口，如图4.7（a）所示。在 ClickNP 中，通道基本上是一个 FIFO 缓冲区，写入一端并从另一端读取。对通道的读/写操作的数据单元称为 *flit*，其具有 64 字节的固定大小。flit 的格式如图 4.7（b）所示。每个 flit 包含元数据的标头和 32 字节的有效负载。当在 ClickNP 元件之间流动时，大量数据（例如，完整大小的数据包）被分成多个 flits。第一个 flit 标有 sop（数据包开始），最后一个 flit 标有 eop（数据包结束）。如果数据块的大小不是 32，则最后一个 flit 的 pad 字段表示已填充到有效负载的字节数。通过硬件描述语言综合工具优化了 flit 中的保留字段。将大数据分解为 flits 不仅可以减少延迟，还可以在不同元件处同时处理分组的不同片段，以增加并行性。最后，为了实现网络功能，可以将多个 ClickNP 元件互连以形成定向处理图，称为 ClickNP 配置。

显然，ClickNP 编程抽象很像 Click 软件路由器[116]。但是，有三个基本差异使得 ClickNP 更适合 FPGA 实现：（1）在 Click 中，元件之间的边是 C ++ 函数调用，并且需要 *queue* 元件来存储数据包。但是，在 ClickNP 中，边实际上表示可以保存实际数据的 FIFO 缓冲区。此外，ClickNP 管道可以打破元件之间的数据依赖关系，并允许它们并行运行。（2）与 Click 不同，其中每个输入/输出端口可以写入（推）或读取（拉）数据，而 ClickNP 统一了这些操作：一个元件只能写入（推）到输出端口，而输入端口只能执行读取（拉）操作。（3）Click 允许元件直接调用另一个元件的方法（通过基于流的路由器上下文），在 ClickNP 中元件之间的协调是基于消息，例如，请求者向响应者发送请求消息并通过另一个消息

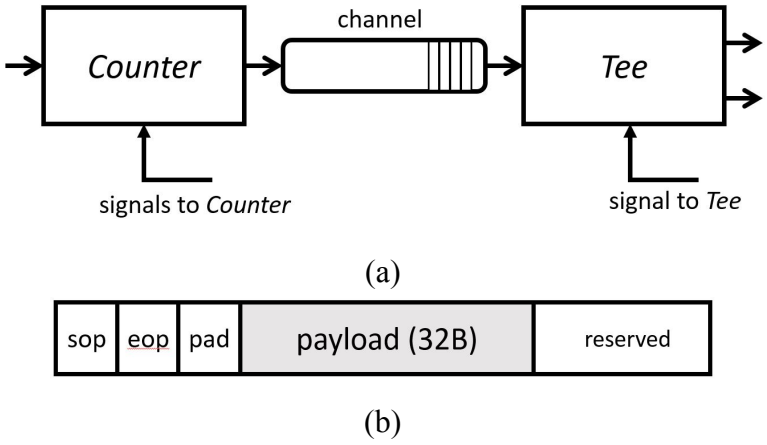


图 4.7　(a) 两个用管道相连接的 ClickNP 元件。(b) Flit 的格式。

获得响应。与通过共享内存进行协调相比，基于消息的协调允许更多的并行性，并且在 FPGA 中更高效，因为对共享内存的访问可能成为瓶颈。

2. 语言

ClickNP 元件（element）可以声明为面向对象语言（如 C++）中的一个对象。不幸的是，许多现有的高层次综合工具仅支持 C 语言。为了利用商业高层次综合工具，可以编写一个编译器，将面向对象的语言（例如 C++）转换为 C，但这种努力并非易事。本文提出一个基于 C 语言子集的领域专用语言（DSL），以支持元件声明。

图 4.8 显示了元件 *Counter* 的代码片段，它只计算传递了多少个数据包。元件由 .element 关键字定义，后跟元件名称和输入/输出端口声明。关键字 .state 定义元件的状态变量，.init、.handler 和 .signal 指定初始化、数据处理和信号处理函数元件。表 4.1 列出了操作输入和输出端口的内置函数。

与 Click 类似，ClickNP 也使用简单脚本来指定网络功能的配置，如图 4.9 所示。配置有两部分：声明和 连接，遵循 Click 语言的类似语法[116]。值得注意的是，在 ClickNP 中，可以使用关键字 host 来注释元件，这将导致元件被编译为 CPU 二进制文件并在 CPU 上运行。

对于一些高层次综合工具难以生成高效硬件逻辑的元件，ClickNP 支持用硬件描述语言编写的 Verilog 元件。为了把 Verilog 元件集成到系统中，开发者需要编写一个接口相同的空元件作为占位符（或功能相同的高级语言元件用于 CPU 调试与测试），并用 verilog 关键字在 ClickNP 配置文件中声明。编译工具链会将高层次综合工具为占位符元件生成的 Verilog 模块替换为开发者的实现。

3. ClickNP 工具链

ClickNP 工具链包含一个 ClickNP 编译器作为前端，一个 C / C++ 编译器（例如，Visual Studio 或 GCC）和一个支持 OpenCL 语言的高层次综合工具（例如，

```
.element Count (flit in -> flit out) {
  .state {
    ulong count;
  }
  .init {
    count = 0;
  }
  .handler {
    if (test_input_port(in)) {
      flit x;
      x = read_input_port(in);
      if (x.flit.fd.sop)
        count = count + 1;
      set_output_port(out, x);
    }
  }
  .signal {
    ClSignal p;
    p.Sig.LParam[0] = count;
    set_signal(p);
  }
}
```

图 4.8　数据包计数器元件的代码。

```
Count :: cnt @
Tee :: tee
host PktLogger :: logger

from_tor -> cnt -> tee [1] -> to_tor
tee [2] -> logger
```

图 4.9　抓包工具元件间互连的 ClickNP 配置文件。使用 host 关键字注释的元件在 CPU 上编译和执行。用 “@” 注释的元件需要从管理器线程接收控制信号。“From_tor” 和 “to_tor” 是两个内置元件，代表 FPGA 上以太网端口的输入和输出。

表 4.1　ClickNP 管道上的内置操作。

uint get_input_port()	获取具有可用数据的所有输入端口的位图。
bool test_input_port(uint id)	测试 id 指示的输入端口。
flit read_input_port(uint id)	读取 id 指示的输入端口。
flit peek_input_port(uint id)	获取 id 指示的输入端口数据，但不取走。
void set_output_port(uint id, flit x)	将 flit 设置为输出端口。处理程序返回时，flit 将写入管道。
ClSignal read_signal()	从信号端口读取信号。
void set_signal(ClSignal p)	在信号端口上设置输出信号。
return (uint bitmap)	.handler 的返回值指定下一次迭代时要读取的输入端口的位图。

Altera OpenCL SDK 或 Xilinx SDAccel）作为后端。如图 4.5 所示，要编写 ClickNP 程序，开发人员需要将代码分为三部分：（1）一组元件，每个元件实现概念上简单的操作，（2）指定这些元件之间连接的配置文件，以及（3）主机管理器，它初始化每个元件并在运行时控制它们的行为，例如，根据管理员的输入。这三部分源代码被送入 ClickNP 编译器并转换为主程序和 FPGA 程序的中间源文件。主程序可以由普通的 C / C++ 编译器直接编译，而 FPGA 程序则使用商业高层次综合工具合成。现有的商用高层次综合工具可以通过时序分析确定每个元件的最大时钟频率。然后，ClickNP 处理图的时钟受到计算流图中最慢元件的约束。另外，高层次综合工具还可以生成优化报告，该报告显示元件中的操作之间的依赖性。如果解析了所有依赖关系并且元件通过在每个时钟周期中处理一个 flit 来实现最佳吞吐量，则元件是*完全流水线化*的。

4.4　FPGA 内部并行化

充分利用 FPGA 内部的并行性对性能至关重要。ClickNP 充分挖掘元件间和元件内的 FPGA 并行性。

4.4.1　元件间并行化

ClickNP 的模块化体系结构使得在不同元件之间利用并行性变得很自然。ClickNP 工具链将每个元件映射到 FPGA 中的硬件模块。这些硬件模块通过 FIFO 缓冲区互连，可以完全并行工作。因此，可以将 ClickNP 配置中的每个元件视为具有自定义逻辑的微小独立核心。数据包沿着*处理管道*从一个元件流向另一个

元件。这种类型的并行性称为*流水线并行*。此外，如果单个处理流水线没有足够的处理能力，可以在 FPGA 中复制多个这样的流水线，并使用负载平衡元件将数据划分到这些流水线中，即利用*数据并行*。对于网络流量，存在数据并行性（在数据包级或流级别）和流水线并行性，可用于加速处理。ClickNP 非常灵活，可以轻松配置两种类型的并行，如图 4.10。

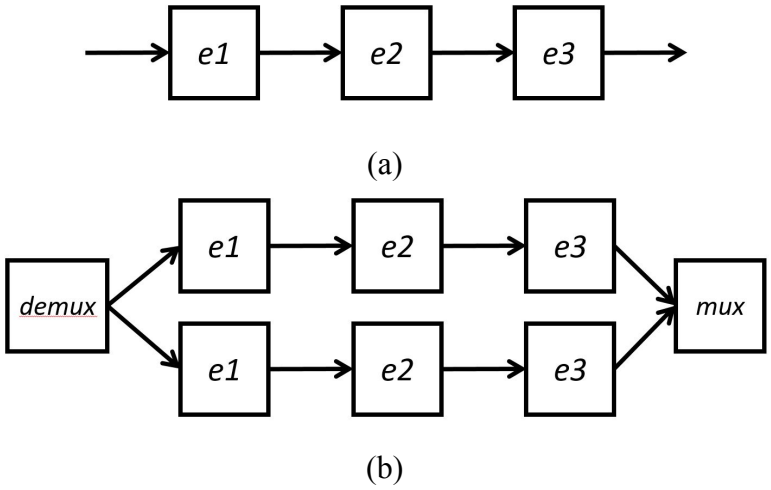


图 4.10　(a) 元件间并行。(b) 元件内并行。

开发者可以手工指定元件间并行的次数（即某个元件重复的次数），也可以指定整个网络功能流水线或某个元件的吞吐量或面积目标，由 ClickNP 工具链自动计算各元件的并行次数。ClickNP 根据高层次综合工具输出的依赖分析报告得到平均每时钟周期最多从输入管道读取的数据量，并根据 FPGA 综合后的时钟频率估计元件的吞吐量[①]；根据 FPGA 的综合结果得到面积。ClickNP 工具链自动平衡各个元件的复制次数，使得流水线中各个元件的处理吞吐量大致平衡。

4.4.2　元件内并行

与在内存中以有限并行性执行指令的 CPU 不同，FPGA 将操作合成为硬件逻辑，因此可以消除指令加载开销。如果数据在一个处理函数中需要多个相关操作，则高层次综合工具将以同步方式将这些操作调度到管道阶段。在每个时钟，一个阶段的结果移动到下一个阶段，同时，一个新的数据被输入到这个阶段，如图 4.11（a）所示。这样，处理函数可以在每个时钟周期处理数据并实现最大吞吐量。但是，实际上，流水线处理可能会在两种情况下效率降低：（1）操作中存在*内存依赖*；（2）有*不平衡*的流水线阶段。以下两个小节将详细讨论这两个问题，并提出解决方案。

[①]假定元件内部无阻塞，即每次迭代都能从输入管道中读入数据。如果元件不能做到，开发者可以指定每次读入数据所需的平均迭代次数。

1. 减少内存依赖

如果两个操作访问相同的内存位置，并且其中至少有一个是 *写操作*，这两个操作被称为相互依赖[184]。因为每个内存访问都有一个周期延迟，并且程序的语义正确性很大程度上取决于操作的顺序，具有 *内存依赖*的操作无法同时处理。如图 4.11（b），S1 和 S2 相互依赖：S2 必须延迟到 S1 结束，只有在 S2 完成后，S1 才能对新的输入数据进行操作。因此，该函数将需要两个周期来处理一个数据。对于某些数据包处理算法，内存依赖性可能相当复杂，但是由于 ClickNP 的模块化体系结构，大多数元件只执行简单的任务，而*读写操作*之间的内存依赖关系是最常见的情况，如图 4.11（b）所示。

消除此内存依赖性的一种方法是仅将数据存储在寄存器中。由于寄存器足够快，可以在一个周期内执行读取，计算和写回，只要计算过程可在一个时钟周期内完成，根本就不存在*读写*依赖。与 CPU 相比，FPGA 的寄存器数量要大得多，如 Altera Stratix V 有 697Kbit 的寄存器，因此可以使用寄存器尽可能减少内存依赖性。当变量是标量，或者变量是数组但所有访问的偏移量均为常量且数组大小不超过阈值时，ClickNP 编译器将该变量在寄存器中实现。程序员可以使用“register” 或 “local / global” 关键字来明确地指示编译器放置一个变量（也可以是一个数组）在寄存器，BRAM 或板载 DDR 内存。

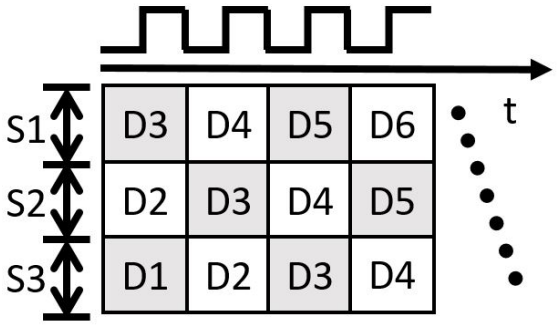
对于较大的数据，它们必须存储在 BRAM 或 DDR 内存中。元件内声明的变量默认存储在 BRAM 中。幸运的是，仍然可以使用一种名为*延迟写入*的技术缓解图 4.11（b）中*读写操作*导致的内存依赖 ①。延迟写入的核心思想是通过增加临时存储来缓解内存依赖。延迟写入将待写入的新数据缓冲在寄存器中，直到下一次读操作 ②。如果下一次读取访问相同的位置，它将从缓冲寄存器中直接读取值。这样，读操作和写操作可以并行执行，因为读写操作一定访问不同的内存位置。③ 图 4.11（c）显示了延迟写入的代码片段。由于代码中不再存在内存依赖性，元件可以每个周期处理一个数据，实现完全流水线化。默认情况下，ClickNP 编译器会对存在读写依赖的数组自动应用 *延迟写入*，生成类似图 4.11（b）的代码。如果存在读写依赖的数组有多处读操作，ClickNP 将为每处读操作分别生成图 4.11（c）中 P1 所示的代码。如果数组有多处写操作，且这些写操作位于互斥的分支条件中，ClickNP 将生成中间寄存器变量，转化成仅有一处写操作的情形。

①图 4.11（b）中写操作的结果可能在下一次循环迭代中被读操作使用。展开相邻的两次循环迭代后，读后写和写后读本质上是同种依赖。

②*延迟写入*得名于变换后的代码模式，在物理上写入到内存的操作并没有被延迟。延迟写入技术生成的硬件逻辑是流水线处理器设计中常用的寄存器转发（register forwarding）模式。

③FPGA 中的大多数 BRAM 都有两个端口，一个用于读操作，一个用于写操作，即每个时钟周期可以执行一次随机读和一次随机写，读写操作分别有一个时钟周期的延迟。

如果数组有多处不互斥的写操作，目前 ClickNP 不能自动生成延迟写入。[①]



(a)

```
1       r = read_input_port (in);
2  S1:  y = mem[r.x]+1;
3  S2:  mem[r.x] = y;
4       set_output_port (out, y);
```

(b)

```
1       r = read_input_port (in);
2  P1:  if ( r.x == buf_addr ) {
3           y_temp = buf_val;
4       } else {
5           y_temp = mem[r.x];
6       }
7       mem[buf_addr] = buf_val;
8  S1:  y = y_temp + 1;
9  S2:  buf_addr = r.x;
10      buf_val  = y;
11      set_output_port (out, y);
```

(c)

图 4.11　内存依赖的例子。(a) 没有依赖性。S*n* 表示流水线的一个阶段，D*n* 是一个数据。(b) 当状态存储在内存中并需要更新时，会发生内存依赖性。(c) 使用延迟写入解决内存依赖性。

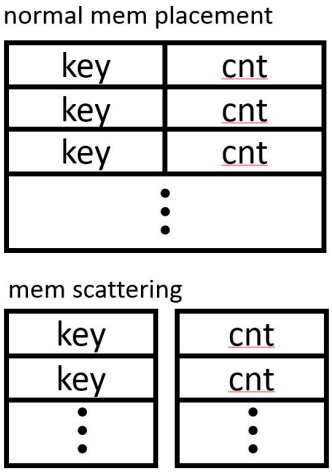
使用 struct 数据结构时会出现一个微妙的问题。图 4.12（a）显示了这样一个例子，其中哈希表用于维护每个流的计数。S2 与 S1 间将具有内存依赖关系，尽管它们正在访问 struct 的不同字段。原因是几乎所有当前的高层次综合工具

[①]理论上可以如下实现：如果有 *N* 处写操作，可以用 *N* 个寄存器保存这些写入的值，并在读取时比较所有 *N* 个寄存器。如果读写操作之间互相穿插，而生成的内存读写指令集中于一处，还需要增加寄存器以保存中间状态。

都会将 struct 数据结构视为具有较大位宽的单个数据 – 等于 struct 的大小，并且只使用一个仲裁器控制访问。这种类型的内存依赖性称为*伪依赖*。在物理上，两个字段 *key* 和 *cnt* 可以位于不同的内存位置。为了解决这个问题，ClickNP 采用了一种名为*内存散射*的技术，它自动将 struct 数组转换为几个独立的数组，每个数组用于存储 struct 中的一个字段。每个数组分配不同的 BRAM，因此可以并行访问（图 4.12（b））。使用*内存散射*后，S1 不再依赖于 S2，从而消除了伪依赖。一般地，如果一个数组的所有访问可被划分为若干互不相交的等价类，每个等价类中的访问地址范围互不相交，就可以应用*内存散射*，将每个等价类所访问的地址范围转换为一个独立的数组[①]。值得注意的是，内存散射仅适用于 FPGA 中的元件，如果元件在主机 CPU 上运行则禁用。

```
1  struct hash_entry
2  {
3    ulong key;
4    ulong cnt;
5  } A[100];
6
7  .handler {
8    ...
9    idx = hash (h);
10 S1: if (A[idx].key==k)
11   {
12 S2: A[idx].cnt ++;
13   }
14   ...
15 }
```

(a)



(b)

图 4.12　内存散射。

当 struct 与 union 共用时，如数据包头有时按字段处理，有时按字节处理，上述内存散射技术不再适用。为了消除数据依赖，ClickNP 采用*数据重排*技术，将 union 中每种类型的数据转换为分立的数组，并根据数据流在数组间同步数据。一般地，如果一个数组的访问按照程序语句的顺序可被划分为若干个阶段，其中每个阶段的数组访问均符合应用*内存散射*技术的条件，ClickNP 将尝试在阶段间插入数据重排代码，并权衡数据重排的开销与内存散射的收益，选择最优的阶段划分[②]。数据重排仅适用于 FPGA 上的元件。

[①]例如图 4.12 的例子中，S1 访问除 16 余 0 ～7 的地址，S2 访问除 16 余 8 ～15 的地址。

[②]例如图 4.12 中，如果 A 数组是从输入管道中读入的，则数据读入阶段和 S1 之间需要插入数据重排代

上述技术可能无法解决所有内存依赖性。一些情况下，尽管理论上数据访问可能存在依赖，但程序的输入或计算逻辑隐式保证了不会发生读写冲突。此时程序员可以通过 `#pragma ivdep` 来指定某段代码并不存在依赖，高层次综合工具将不再分析这些依赖关系。在许多情况下，为了确保在 FPGA 中完全流水线化，程序员需要将代码重写为访问不相交内存区域的多个元件。

2. 平衡流水级

理想情况下，一个处理管道中的每个阶段应当具有相同的速度。即，在一个时钟周期处理数据。但是，如果每个阶段的过程不平衡并且某些阶段需要比其他阶段更多的时钟周期，则这些阶段将限制管道的整个吞吐量。例如，在图 4.13（a）中，S1 是一个循环操作。由于每次迭代需要一个周期（S2），整个循环将需要 N 周期才能完成，从而显着降低了管道吞吐量。图 4.13（b）显示了另一个示例，它为 DDR 中的全局表（*gmem*）实现了一个 BRAM 缓存。虽然 “else” 分支很少命中，但它会在管道中产生一个胖阶段（需要数百个时钟周期）。我们使用的高层次综合编译器为每个阶段预留最坏情况的时钟周期数量，因此，即使胖阶段很少被用到，也会大大影响整个流水线的处理速度。

ClickNP 使用两种技术来平衡管道内的各个阶段。首先，尽可能地展开（*unroll*）循环。循环展开可以将循环分解为一系列可并行或流水线执行的小操作。值得注意的是，展开循环将复制循环体中的操作，从而增加面积成本。因此，它可能仅适用于具有简单循环体和少量迭代的循环。在网络功能中，这种小循环是相当常见的，例如计算校验和，对数据包有效负载进行移位，或者迭代枚举各种可能的配置。ClickNP 编译器提供 unroll 指令来展开循环。

虽然许多高层次综合工具支持展开已知迭代次数的循环，但很多现实应用的循环迭代次数是不固定的。然而，在网络功能中，往往可以确定循环迭代次数的上界，如数据包的最大长度。由于循环的迭代器常常用来作为数组索引，这种情况下可以根据数组的大小确定迭代器的上下界 ①。另有一些循环的迭代器上界是常量或其他循环迭代器组成的简单表达式，这种情况下可以计算上界表达式的最大值。对于编译器不能自动确定循环迭代次数的情况和没有显式迭代器的 `while` 循环，ClickNP 允许程序员通过 `pragma` 指定循环次数的上界。循环次数上界确定后，ClickNP 将循环体用表示循环条件的 `if` 语句包裹，替换循环体中的 `continue` 和 `break` 语句，然后复制多份。

对图 4.13（a）所示的计算，循环展开后还需要解决内存依赖，因为变量 `sum` 被多次写入和读取。ClickNP 在循环展开后，对标量变量展开为静态单赋值（static

码，将 A 数组的内容分别搬移到内存散射后 S1 和 S2 对应的数组。显然，只有当每次读入数据后 S1 和 S2 执行的次数足够多时，数据重排才有收益。

①ClickNP 不支持动态内存分配，因此数组的大小都是编译时可以静态确定的。

```
1 .handler {
2     r = read_input_port (in);
3     ushort *p = (ushort*) &r.fd.data;
4 S1: for (i = 0; i<N; i++) {
5 S2:   sum += p[i];
6     }
7     set_output_port (out, sum);
8 }
```

(a)

```
1 .handler {
2     r = read_input_port (in);
3     idx = hash (r.x);
4 S1: if ( cache[idx].key == r.x ) {
5       o = cache[idx].val;
6 S2: } else {
7       o = gmem[r.x];
8       k = cache[idx].key;
9       gmem[k] = cache[idx].val;
10      cache[idx].key = r.x;
11      cache[idx].val = o;
12    }
13    set_output_port (out, o);
14 }
```

(b)

图 4.13　不平衡的流水级。

single assignment）形式，使得每个变量仅被赋值一次，即可消除这种内存依赖。本质上，静态单赋值与延迟写入都是用增加内存空间的方式提高并行性。

第二种技术是，如果元件同时具有快速和慢速操作，可以尝试将单个元件中的每种类型的操作分开。例如，在缓存元件的实现中，如图 4.13（b）所示，较慢的 “else” 分支被移动到另一个元件中，使得快速路径和慢速路径异步运行。如果缓存不命中率很低，则整个元件的处理速度由快速路径决定。如图 4.14，ClickNP 编译器提供 “async” 原语，用户可以在 handler 中插入 async { } 包裹的代码块，其中的代码将被编译成一个新元件，通过管道与原来的元件相连。原来的元件将异步元件中所用到的变量序列化并发送给异步元件，然后等待异步元件结束。异步元件执行完毕后，将原来元件仍将用到的写入变量发送回原来元件。

```
.handler {
  r = read_input_port (in);
  idx = hash (r.x);
  if (cache[idx].key == r.x) {
    o = cache[idx].val;
  } else {
    k = cache[idx].key;
    v = cache[idx].val;
    .async {
      o = gmem[r.x];
      gmem[k] = v;
    }
    cache[idx].key = r.x;
    cache[idx].val = o;
  }
  set_output_port (out, o);
}
```

图 4.14　Async 原语示例。

4.5　系统实现

4.5.1　ClickNP 工具链与硬件平台

本章实现了一个 ClickNP 编译器，作为 ClickNP 工具链的前端（§3）。对于宿主程序，使用 Visual C++ 作为后端。进一步集成了 Altera OpenCL SDK（即 Intel FPGA SDK for OpenCL）[57] 和 Xilinx SDAccel[49] 作为 FPGA 程序的后端。ClickNP 编译器的核心部分包含约 2 万行 C++、flex 和 bison 代码，它们解析配置文件和元件声明，执行第 4.4 节所述的优化，并生成特定于每个商业高层次综合工具的代码。

对于在 FPGA 上运行的元件，每个元件被编译成中间 C 代码，进而通过高层次综合工具编译成一个逻辑模块。使用 Altera OpenCL 高层次综合工具时，每个 ClickNP 元件都被编译成 *kernel*，元件之间的连接被编译为 Altera 扩展通道（channel），进而用 Avalon ST 接口实现；元件与板上 DRAM（即全局内存）之间使用 Avalon MM 接口通信。使用 Xilinx SDAccel 高层次综合工具时，将每个元件编译成 IP 核（IP core），并使用 AXI 流（stream）来实现元件之间的连接，实现 AXI 内存映射接口访问板上 DRAM。在主机上运行的元件被编译为 CPU 二进制文件，管理进程将为每个主机元件创建一个工作线程。主机和 FPGA 元件之间的每条管道都映射到 PCIe I/O 通道的插槽（§4.5.3）。

本文的硬件平台基于 Altera Stratix V FPGA 和 Catapult shell[48]。Catapult shell 还包含一个 OpenCL 特定的运行时（BSP）。ClickNP 用户逻辑通过 BSP 与 shell 通信。ClickNP 用户逻辑运行在独立的时钟域内，BSP 将 shell 中处于不同时钟域的 PCIe DMA、DRAM 等接口通过异步 FIFO 转换为用户逻辑所在的时钟域。BSP 还提供了 OpenCL 内核启动、停止等管理功能。撰写本文时，作者还没有获得 Xilinx 硬件平台。因此，系统评估主要基于使用 ClickNP+ OpenCL 的 Altera 平台，使用 Vivado HLS 生成的报告（例如频率和面积成本）来了解 ClickNP+ Vivado 的性能。

1. 适合高层次综合工具的中间 C 代码

当主机上电或 FPGA 被在线重配置后，各个 kernel 或 IP 核就开始并行运行。如图 4.15 所示，每个 kernel 首先执行初始化（init）函数，然后进入永不退出的循环，检查输入管道并执行事件处理（handler）函数，检查信号并执行事件处理函数。

```
void kernel() {
  调用 init 函数;
  声明并初始化输入输出缓冲区;
  while (true) {
    if (输入缓冲区有空闲 and 输入管道非空) {
      从输入管道中搬移数据到输入缓冲区;
    }
    调用 handler 函数;
    if (输出管道有空闲 or 输出缓冲区已满) {
      从输出缓冲区搬移数据到输出管道;
    }
    if (输入事件管道非空) {
      从输入事件管道读取输入事件;
      调用 signal 函数;
      将事件处理响应写入输出事件管道;
    }
  }
}
```

图 4.15　Kernel 中间 C 代码的伪代码。

高层次综合工具将中间 C 代码变换成硬件描述语言，中间 C 代码中的每层循环或者被完全展开（unroll）成流水线化的逻辑模块，或者被实现成状态机，每个时钟周期执行循环的一次迭代。循环展开仅适用于循环次数在编译时静态可知，且循环次数较小的情况。对于实现成状态机的循环，由于循环的各次迭代之

间可能存在数据依赖，且一次迭代的流水线逻辑可能需要若干个时钟周期才能执行完毕，相邻两次循环迭代间可能需要若干时钟周期的间隔。高层次综合工具需要根据依赖关系和循环体的流水线延迟信息，计算相邻两次迭代的最小间隔（initiation interval，II）。II 越小，吞吐量越高。因此，我们希望尽量减小 II。

目前，高层次综合工具为计算密集型操作设计。计算密集型操作往往由多层嵌套循环构成，编译器使用的循环变换方法通常适用于控制流编译时静态可知（static control part）、完美嵌套循环（perfectly nested loop）的程序。然而，图 4.15 的程序中 while 循环的执行次数在编译时不可知，且由于输入输出缓冲区搬移代码的存在，handler 和 signal 函数中的循环均不是完美嵌套循环。早期版本的高层次综合工具由于完备性问题，不仅对较为复杂的源程序可能出现编译失败、编译时间过长等错误，还可能分析出过多不必要的内存依赖，导致 II 过大，甚至导致 II 静态分析失败，内层循环无法并行。

本文希望绕过现有高层次综合工具的嵌套循环优化。注意到网络功能中的元件计算逻辑相对简单，每个时钟周期处理的数据量也很有限（如一个 flit），很多循环的执行次数可以在编译时静态确定（例如处理一个 flit 的每个字节）。因此，ClickNP 将 handler 和 signal 函数中的所有循环进行展开（unroll）或压平（flatten），使生成的中间 C 代码仅有 while 一层循环，而不再有嵌套循环，如图 4.16 所示。ClickNP 的默认策略是展开循环次数可在编译时确定的循环，并压平所有其他循环。用户也可以通过嵌入在源程序中的编译选项（pragma）指定展开和压平的策略。这样，高层次综合工具只需分析单层循环，减小了出错的可能性。

展开循环体还有一个重要的好处：便于编译优化。传统编译器往往很难优化用循环表示的向量操作。ClickNP 进行循环展开后向量操作被分解为逐点操作，高层次综合工具就可以进行常量传播、死代码消除等一系列优化。此外，循环展开后，ClickNP 可以进行静态单赋值变换，还可以将访问地址由循环变量确定的数组展开为若干个离散的寄存器，从而消除内存依赖。

ClickNP 可以生成性能分析报告。在每个元件内，分析报告包括每个变量的存储方式，每个循环的展开或压平策略，相邻两次迭代的最小间隔（II）以及导致 II 瓶颈的依赖关系链。在计算图层次上，分析报告包括每个元件的延迟、吞吐量和时钟频率。

2. 编译速度优化

FPGA 编程的一个限制是编译时间相当长。一个网络数据包简单转发的功能就需要 3 小时来编译。编译时间主要由高层次综合、IP 核生成、硬件描述语言逻辑综合、FPGA 布局布线、时序分析等几个阶段构成。本章采用几项技术来缩短 FPGA 编译时间。第一，OpenCL 编程框架会把高层次综合生成的 Verilog 模块生成 IP 核，插入到 shell 部分中，这需要复制大量 IP 核和 shell 部分的代码。注意到

```
while (true) {
  Packet pkt = read_input_port(in);
  uchar checksum = 0;
  #pragma unroll 2
  for (int i = 0; i < pkt.num_flits(); i++) {
    flit f = pkt.filt(i);
    for (int j = 0; j < FLIT_BYTES; j++)
      checksum ^= f.bytes[j];
  }
  write_output_port(out, checksum);
}
```

(a) 原始的 C 中间代码（示意代码）。

```
uchar checksum = 0;
Packet pkt;
int i = 0;
while (true) {
  if (i == 0)
    pkt = read_input_port(in);
  if (i < pkt.num_flits()) {
    flit f = pkt.filt(i);
    checksum ^= f.bytes[0]; checksum ^= f.bytes[1];
    ...
    checksum ^= f.bytes[FLIT_BYTES - 1];
    i++;
  }
  if (i < pkt.num_flits()) {
    flit f = pkt.filt(i);
    checksum ^= f.bytes[0]; checksum ^= f.bytes[1];
    ...
    checksum ^= f.bytes[FLIT_BYTES - 1];
    i++;
  }
  if (i == pkt.num_flits()) {
    write_output_port(out, checksum);
    i = 0;
  }
}
```

(b) 循环展开和压平后的结果。

图 4.16　循环展开和压平。对 i 循环按照并行度 2 进行展开，并压平；j 循环完全展开：变换的结果是每个时钟周期计算 2 个 flit 的所有字节。

用户逻辑的外围接口是固定的，本文预先生成 IP 核，只需将高层次综合 Verilog 模块文件替换进项目中；对于 shell 部分代码，使用文件引用而非拷贝。第二，为了缩短逻辑综合时间，将固定的 shell 部分通过逻辑综合生成网表（netlist）文件，利用设计分区（design partition）保留综合结果，可以将 shell 部分的逻辑综合时间减少约 35 分钟。第三，为了加快 FPGA 布局算法的收敛速度，在初始编译完成后，增加 shell 部分各模块的布局约束，将其布局基本上固定下来。shell 部分的模块大多与芯片固定位置的硬核（hard IP）交互，固定布局在硬核附近是

合理的。但是为了更好的性能，本文没有使用设计分区将 shell 部分的布局布线完全固定下来。这项优化可节约大约 20 分钟。第四，区分调试（debug）和发布（release）编译模式。调试模式旨在验证逻辑上的正确性，而不追求性能。在调试模式下，把用户逻辑的时钟频率固定为 50 MHz，大大降低了布局布线的难度；使用设计分区固化 shell 部分的布局布线，这部分高频逻辑的布局布线需要较长的时间。调试模式相比发布模式可节约 25 分钟编译时间，在用户逻辑更复杂时节约的时间将更多。第五，删除不必要的时序分析模型。OpenCL 框架默认在四种情形下分析时序约束，但只要满足其中最苛刻的一种，其余三种也能满足，因此我们仅保留最苛刻的时序约束模型。第六，OpenCL 框架为了尽可能提高性能，首先使用较高的用户逻辑时钟频率（如 250 MHz）编译，然后根据时序分析结果计算出最长延迟和能正确工作的最高时钟频率，再用该时钟频率进行二次布局布线和时序分析。这对计算密集型工作负载可以达到尽可能高的吞吐量。然而，本文关注网络数据包处理，只需达到网络线速的处理能力，因此可以固定时钟频率为 180 MHz，大多数用户逻辑均可达到此频率，因此无需重新布局布线。

表 4.2　FPGA 编译加速技术。

编译阶段	优化方法	优化前编译时间（分）	优化后编译时间（分）
ClickNP 编译	–	0.1	0.1
高层次综合	–	1	1
生成 IP 核	预先生成 IP 核；使用文件引用而非拷贝	10	0
逻辑综合	保留 shell 的综合结果	50	15
布局布线	增加 shell 的布局约束；调试模式下降低用户逻辑的时钟频率，保留 shell 的布局布线结果	60	40 (15)*
时序分析	删除不必要的时序分析模型	15	5 (0)*
二次布局布线	固定时钟频率，无需重新布局布线	30	0
二次时序分析	删除	15	0
总计	–	180	60 (30)*

* 括号内是调试模式下的编译时间。

表 4.2 总结了上述编译加速技术。优化前，开发者每个工作日只能进行 3 轮调试，而优化后在调试模式下可调试 10 轮左右，需要性能的系统测试也可进行约 6 轮，大大提高了工作效率。

4.5.2　ClickNP 元件库

本文实现了一个包含近 200 个元件的 ClickNP 元件库。其中一部分（约 20 %）直接来自 Click Modular Router，但使用 ClickNP 编程框架重新编写。这些元件涵盖了网络功能的大量基本操作，包括数据包解析，校验和计算，封装/解封，哈希表，最长前缀匹配（LPM），速率限制，加密和数据包调度。由于 ClickNP 有模块化的体系结构，每个元件的代码大小都是适中的。元件的平均代码行数（Line of Code，LoC）是 80，最复杂的元件 *PacketBuffer* 有 196 行 C 代码[①]。

表 4.3 显示了在 ClickNP 中实现的一组选定的关键元件。除了元件名称，还标记了使用该元件的演示网络功能（在 §4.6 中讨论）。之前在 §4.4.2 中讨论的优化技术最小化了内存依赖性，并平衡了流水线阶段。第 3 列总结了每个元件使用的优化技术。对于表 4.3 顶部的元件，元件中的处理逻辑需要访问数据包的每个字节。其中吞吐量以 Gbps 显示。但是，表格底部的元件只处理数据包头（元数据）。因此，使用每秒数据包来测量吞吐量更有意义。值得注意的是，表 4.3 中测量的吞吐量是相应元件可以实现的最大吞吐量。当它们在真正的网络功能中工作时，其他组件，例如以太网端口，可能是瓶颈。作为参考，表 4.3 比较了优化后的 FPGA 版本、不应用 §4.4 所述技术的 FPGA 简单实现以及 CPU 实现。很明显，经过优化后，所有这些元件都可以非常高效地处理数据包，与初始的 FPGA 实现相比，速度提高了 7 至 117 倍，与一个 CPU 内核上的软件实现相比，速度提高了 21 %。这种性能提升来自于利用 FPGA 中的巨大并行性的能力。考虑到 FPGA 的功耗（大约 30W）和 CPU（每个核心大约 5W），ClickNP 元件的能耗效率比 CPU 高 4 到 120 倍。

表 4.3 还显示了每个元件的处理延迟。可以看到，这个延迟很低：平均值是 $0.19\mu s$，最大值仅为 $0.8\mu s$（LPM _Tree）。最后两列总结了每个元件的资源利用率。利用率归一化为 FPGA 芯片的容量。大多数元件只使用少量的逻辑元件。这是合理的，因为大多数数据包操作都很简单。HashTCAM 和 RateLimiter 具有适度的逻辑资源使用，因为这些元件具有更大的仲裁逻辑。但是，BRAM 的使用很大程度上依赖于元件的配置。例如，BRAM 使用率随流表中支持的条目数呈线性增长。总而言之，本文使用的 FPGA 芯片有足够的容量来支持包含少量元件的有意义的网络功能。

4.5.3　PCIE I/O 通道

如前所述，ClickNP 的一个关键属性是支持联合 CPU / FPGA 处理。本章通过设计高吞吐量，低延迟的 PCIe I / O 通道来实现这一目标。ClickNP 支持灵活

[①]代码行数是指 ClickNP 元件代码，不包括主机控制程序和测试代码。

表 4.3 ClickNP 中的一些网络元件。

元件	配置	优化	性能				资源占用 (%)	
			最高频率 (MHz)	峰值吞吐量	加速比 (FPGA/CPU)	延迟（时钟周期）	LE	BRAM
L4_Parser (A1-5)	N/A	REG	221.93	113.6 Gbps	31.2x / 41.8x	11	0.8%	0.2%
IPChecksum (A1-4)	N/A	UL	226.8	116.1 Gbps	33.1x / 55.1x	18	2.3%	1.3%
NVGRE_Encap (A1,4)	N/A	REG, UL	221.8	113.6 Gbps	35.5x / 42.9x	9	1.5%	0.6%
AES_CTR (A3)	16 字节块大小	UL	217.0	27.8 Gbps	79.9x / 255x	70	4.0%	23.1%
SHA1 (A3)	64B 字节块大小	MS, UL	220.8	113.0 Gbps	157.5x / 83.1x	105	7.9%	6.6%
CuckooHash (A2)	128K 个条目	MS, UL, DW	209.7	209.7 Mpps	43.6x / 57.5x	38	2.0%	65.5%
HashTCAM (A2)	16 x 1K 个条目	MS, UL, DW	207.4	207.4 Mpps	155.9x / 696x	48	18.7%	22.0%
LPM_Tree (A2)	16K 个条目	MS, UL, DW	221.8	221.8 Mpps	34.5x / 45.2x	181	4.3%	13.2%
FlowCache (A4)	4 路组相连，16K 条目	MS, DW	105.6	105.6 Mpps	55.8x / 21.5x	27	5.6%	46.9%
SRPrioQueue (A5)	32 个数据包的缓冲区	REG, UL	214.5	214.5 Mpps	150.3x / 28.6x	41	2.6%	0.6%
RateLimiter (A1,5)	16K flows	MS, DW	141.5	141.5 Mpps	7.0x / 65.3x	14	16.9%	14.1%

优化方法。 REG=Using Registers 使用寄存器; MS=Memory Scattering 内存散射; UL=Unroll Loop 循环展开; DW=Delay Write 延迟写入。

性能提升一列对比了应用第 §4.4 节的优化后和优化前的 FPGA 性能。也列出了与 CPU 实现间的性能对比。

的 I/O 操作。如图 4.17 所示，ClickNP 提供了基于槽位（slot）和工作队列（work queue）两种与主机 CPU 通信的抽象，还提供了裸 DMA 操作的接口。

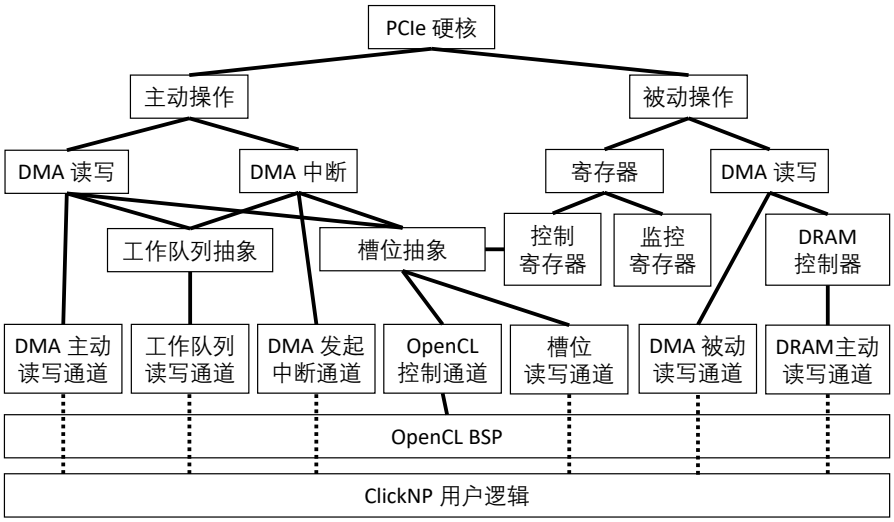


图 4.17　PCIe I/O 通道的架构。

在基于槽位的抽象中，PCIe 物理链路划分为 64 个逻辑子通道，即 *slots*（插槽）。每个插槽都有一对用于 DMA 操作的发送和接收缓冲区。在 64 个插槽中，33 个由 OpenCL BSP 用于管理 ClickNP 内核和访问板载 DDR（即 OpenCL 控制通道），一个插槽用于将信号传递给 ClickNP 元件。剩余的 30 个插槽用于 FPGA 和 CPU 元件之间的数据通信。CPU 上的槽位抽象可以在中断或轮询模式下工作。

槽位抽象中每发送一个数据都需要至少 4 次 DMA 操作[①]，且需要等待对面的设备处理完成才能在同一槽位发送下一个数据。为分摊 DMA 开销、提高消息发送的并发度，工作队列是槽位抽象的扩展，每个槽位不再仅有一对缓冲区，而是有一对用于发送和接收的环形缓冲区队列。每条环形缓冲区队列有 128 个槽位，按照先进先出的方式访问。发送端发现环形缓冲区队列中还有尚未被取走的数据时，就无需通知对端，节约了 CPU 通过 PCIe MMIO 发送门铃（doorbell）和 FPGA 发送中断的开销。

除了槽位和工作队列，FPGA 和 CPU 之间还需要更灵活的通信方式。首先，第 5 章的键值存储中，FPGA 需要直接读写主机内存中的键值，而无需主机 CPU 参与，这就需要 FPGA 能够直接发出裸的 PCIe DMA 读写请求。其次，在基于可编程网卡的内存解聚中，FPGA 将远程内存通过 PCIe MMIO 直接映射到主机内

[①]从主机 CPU 发送数据到 FPGA 的过程是：主机 CPU 写 FPGA 中的下行控制寄存器（又称门铃，doorbell）；FPGA 从主机内存中 DMA 读取数据。当 FPGA 处理完槽位中的数据后，写主机内存中的下行完成寄存器，并向主机 CPU 发送中断。从 FPGA 向主机 CPU 发送数据的过程是：FPGA 读内部的上行控制寄存器，判断为空；FPGA 向主机内存 DMA 写入数据，并向主机 CPU 发送中断。主机接收 FPGA 发来的数据过程是：读 FPGA 中的上行控制寄存器，判断非空；读取主机内存中的数据；写 FPGA 中的上行控制寄存器，表示处理完毕。

存空间，主机 CPU 直接访问此内存空间，生成 PCIe DMA 读写请求发送到 FPGA。FPGA 中的用户逻辑需要处理这些 DMA 被动读写操作。最后，一些应用（如传统 OpenCL 应用）可能希望主机 CPU 与 FPGA 之间采用 FPGA 板上的 DRAM 作为共享内存，因此 FPGA 板上的 DRAM 通过 PCIe MMIO 映射到主机内存空间，由 shell 中的 PCIe 被动读写逻辑发送到 DRAM 控制器。由于主机 PCIe MMIO 读写大块数据的效率较低，也支持主机 CPU 通过控制寄存器，让 FPGA shell 主动发起 DMA 在板上 DRAM 和主机内存间搬运数据。

图 4.18 显示了具有不同插槽数和批量大小的 PCIe I/O 通道的基准测试结果。作为基线，还测量了 OpenCL 全局内存（global memory）操作的性能 – 到目前为止，这是 OpenCL[185] 中 CPU 与 FPGA 间通信提供的唯一方法。在图 4.18 中可以看到，单个插槽的最大吞吐量约为 8.4 Gbps。通过 4 个插槽，PCIe I/O 通道的总吞吐量可达 25.6 Gbps ①。但是，OpenCL 的吞吐量低得惊人，甚至低于 1 Gbps。这是因为全局内存 API 旨在传输 GB 级的大量数据。这可能适用于具有大数据集的应用程序，但不适用于需要强流处理（stream prpocessing）功能的网络功能。同样，图 4.18（b）显示了通信延迟。由于 OpenCL 未针对流处理进行优化，因此 OpenCL 延迟高达 1 ms，通常是网络功能无法接受的。相比之下，PCIe I/O 通道在轮询模式下具有非常低的 1 μs 的延迟（一个 CPU 核持续轮询状态寄存器），而在中断模式下的延迟为 9 μs（几乎没有 CPU 开销）。

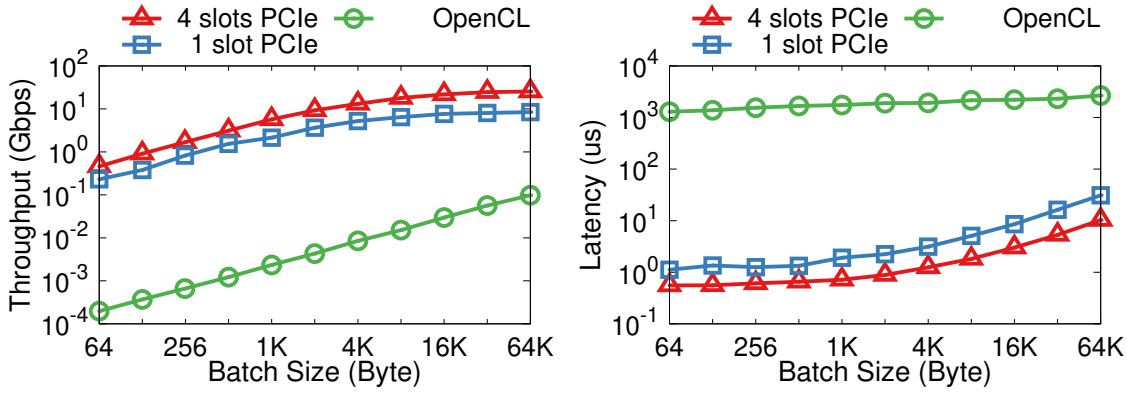


图 4.18　PCIe I/O 通道的性能。Y 轴是对数坐标系。

为了给 FPGA 元件发送信号，ClickNP 编译器自动在 FPGA 中生成一个名为 *CmdHub* 的特殊元件。*CmdHub* 将主机管理程序下发的控制信号通过管道分发到 FPGA 元件，FPGA 元件再将信号处理函数的结果通过管道返回给 *CmdHub*，进而返回给主机管理程序。为了避免一对多管道连接带来的布局布线困难，*CmdHub*

①这是本章写作时使用的 PCIe Gen2 x8 硬核的实际最高性能。事实上该 FPGA 支持 PCIe Gen3 x8 硬核，第 5 章通过替换硬核和优化 shell，实现了 2 倍的 PCIe I/O 通道吞吐量。

与所有元件构成一条菊花链，从 *CmdHub* 开始，按照元件连接图的拓扑排序依次穿过所有元件，最后回到 *CmdHub*。为了在菊花链中识别目标元件，信号消息中嵌入了元件 ID，每个元件仅处理匹配元件 ID 的信号消息，而对其他信号消息直接转发。

4.5.4　调试

ClickNP 提供两种方法用于调试。

CPU 功能仿真。 ClickNP 元件使用类 C 的高级语言编写，因此一个元件可以编译成 CPU 上运行的一个线程，而管道就是线程之间的队列。开发者可以使用熟悉的软件调试工具进行功能仿真。

实际 FPGA 运行。 CPU 功能仿真存在局限。首先，元件之间的通信管道有死锁的可能，在 CPU 功能仿真时由于时序与硬件逻辑不一致，不一定能发现死锁问题；其次，CPU 仿真速度较慢，不能反映实际性能，且难以测试与 PCIe DMA、网络之间的交互；最后，功能仿真不能发现编译器中的错误。为此，实际应用在功能仿真通过后，一般用实际 FPGA 运行的方法调试。

由于 ClickNP 语言中的变量名与 Verilog 中的不匹配，SignalTap 等用于 FPGA 在线调试的工具并不适用。因此，ClickNP 需要定制的调试机制。首先，在调试模式下，ClickNP 可以记录每条管道的输入输出日志，并通过 PCIe I/O 管道传输到主机内存上。第二，ClickNP 允许用户在元件代码中插入 printf 语句，通过管道将包含变量值的调试信息发送到主机端。第三，ClickNP 支持用户在编译时插入断点，以便调试交互式网络协议或模拟队列阻塞导致的死锁。断点被编译成管道的写操作（通知主机断点命中）和阻塞读操作（等待主机发送断点继续命令）。第四，ClickNP 允许在运行过程中（包括断点命中时）随时查询或修改某个变量的值。当用户查询某个变量的值时，通过 signal 发送查询或修改命令，并返回变量的值。

4.5.5　元件热迁移和高可用

热迁移是数据中心网络功能需要支持的重要特性。当虚拟机热迁移时，计算节点对应的网卡内部状态需要热迁移到新节点，不然就需要在新节点上重新初始化网卡状态，带来软件的复杂性和迁移延迟。类似的，当网络、存储节点因升级、扩容等进行热迁移时，网卡状态也需要热迁移到新节点。此外，为了实现服务不间断的网络功能升级，需要将内部状态热迁移到另一节点，再将原节点下线进行升级。为了实现高可用性，当增加备份节点时，为了同步源节点和新增备份节点的内部状态，也需要热迁移技术。

当热迁移开始时，首先在交换机上进行配置，停止向旧 FPGA 发送数据包，

而是将这些数据包缓冲在交换机内。然后通过第 4.5.4 节的断点机制停止 FPGA 内各元件的运行，再将所有元件内变量的值、管道内的数据和全局内存的值导出到主机。接下来，在新 FPGA 上运行相同的 ClickNP 程序，通过调试机制导入上述 FPGA 内部状态，并恢复各元件运行。最后，通知交换机修改路由表，将旧 FPGA 的地址指向新 FPGA 所在端口，发送交换机内缓冲的数据包，热迁移结束。

为了实现网络功能的高可用，ClickNP 采用状态机复制的方法。两个 FPGA 接收到相同的数据包序列，只要元件内部没有随机化或与时间相关的处理逻辑，也不接受主机的控制信号，就能保证两个 FPGA 的内部状态和发出的数据包序列相同。当检测到备份节点故障时，只需启动一个新的备份节点，然后执行状态热迁移。当检测到主节点故障时，需要切换到备份节点，此时可能出现少量输入数据包丢失或输出数据包重复，TCP 可以安全地处理这些情况。

4.6　应用与性能评估

为了评估 ClickNP 的灵活性，本文基于 ClickNP 开发了几个常见的网络功能，可以在本文的测试床中运行。表 4.4 总结了每个网络功能中包含的元件数量和总代码行数，包括所有元件规范和配置文件。经验证实，ClickNP 的模块化架构极大地改善了代码重用并简化了新网络功能的构建。如表 4.3 所示，在许多应用程序中有很多机会重用一个元件，例如，本文中所有的网络功能都使用了 L4_Parser。每个网络功能可能需要 1 个左右的时间才能让一个程序员进行开发和调试。联合 CPU / FPGA 处理的能力也将极大地帮助调试，因为可以将有问题的元件移动到 CPU，以便轻松地打印日志来跟踪问题。

本章在 16 台 Dell R720 服务器的测试台中评估 ClickNP。对于每个 FPGA 板，两个以太网端口都连接到架顶式（Top-of-Rack）Dell S6000 交换机[186]。所有 ClickNP 网络功能都在 Windows Server 2012 R2 上运行。本章比较 ClickNP 与其他最先进的软件网络功能。对于在 Linux 上运行的那些网络功能，使用内核版本为 3.10 的 CentOS 7.2。测试使用 PktGen 发包工具以不同的速率生成具有不同数据包大小的测试流量（64B 数据包，最高吞吐量为 56.4 Mpps）。为了测量网络功能处理延迟，在每个测试数据包中嵌入了一个生成时间戳。当数据包通过网络功能时，它们被循环回到 PktCap 抓包工具，该 PktCap 与 PktGen 位于同一 FPGA 中。然后可以通过从数据包的接收时间中减去生成时间戳来确定延迟。由 PktGen 和 PktCap 引起的延迟通过直接环回（无网络功能）进行预校准，并从数据中删除。

下面依次介绍基于 ClickNP 的网络功能。

4.6.1　数据包生成器和抓包工具

数据包生成器（PktGen）可以根据不同的配置文件生成各种流量模式。它可以生成不同大小的流，并根据给定的分布安排它们在不同的时间开始。生成的流可以进一步通过不同的流量整形器来控制流速及其突发性（burstiness）。

抓包工具（PktCap）将收到的所有数据包重定向到 *logger* 元件，这些元件通常位于主机中。图 4.19 展示了抓包工具的元件结构。由于单个抓包元件无法充分利用 PCIe I/O 通道容量，因此 PktCap 在 FPGA 中实现了接收端缩放（RSS）元件，以根据流 5 元组的哈希值将数据包分发到多个抓包元件。由于 PCIe 通道的吞吐量小于 40G 网卡的吞吐量，添加了一个提取器（extractor）元件，它只提取数据包的重要字段（例如，如果有 5 个元组，DSCP 和 VLAN 标记），并通过 PCIe 转发这些字段（总共 16B）以及时间戳（4B）。PktCap 就是一个展示联合 CPU / FPGA 处理重要性的例子。与 FPGA 相比，CPU 具有更多的内存用于缓冲，并且可以轻松访问其他存储，例如，文献[187] 中的 HDD / SSD 驱动器，因此在 CPU 上运行记录器更有意义。

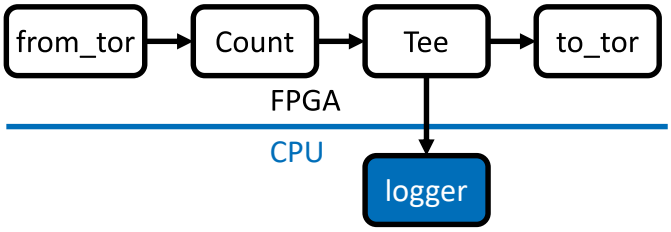


图 4.19　抓包工具的元件结构。

4.6.2　OpenFlow 防火墙

Openflow[97] 防火墙支持流的精确匹配、前缀匹配和通配符模糊匹配。精确匹配表是使用 Cuckoo Hashing[188] 实现的，可容纳精确匹配流 5 元组的 128K 条规则。前缀匹配表使用决策树实现，树的每一层存储在一个 BRAM 中，从而可以实现流水线处理。前缀匹配的最大延迟即为决策树的深度。

模糊匹配需要 TCAM。但是，TCAM 在 FPGA 中很难高效实现，可容纳 512 条规则[①]的简单 TCAM 实现会占用 FPGA 的 65 % 逻辑资源。为了降低面积开销，本章采用基于 BRAM 的 TCAM[189]。基于 BRAM 的 TCAM 将待匹配的键分解为多个较小的键。每个键与一个 BRAM 关联，用键作为 BRAM 地址，查找结果为一个宽度为规则条数的比特位图（bitmap），表示每条规则是否匹配。BRAM 的内容在规则修改时由主机管理程序预处理。最终的匹配结果即为各 BRAM 输出的比特位图的按位 “与” 中第一个有效位的位置。基于 BRAM 的 TCAM 用更

[①]每条规则匹配五元组的 104 位，下同。

多的内存空间来降低逻辑资源占用。具有 2K 条规则的 BRAM TCAM 需要占用 14％的逻辑资源和 43％的 BRAM。

为了进一步降低逻辑资源和内存空间开销，注意到流表中的许多条目共享相同的位掩码这一事实，本文设计了 HashTCAM。HashTCAM 将表空间划分为许多较小的哈希表，每个哈希表都与一个可配置的位掩码相关联，可以容纳位掩码相同的一组 TCAM 规则。输入的流 5 元组首先与位掩码执行“与”操作，再查找哈希表。表中的每个条目与优先级相关联。仲裁器（multiplexer）选择具有最高优先级的匹配条目。HashTCAM 在规则条数和面积成本之间有更好的权衡。具有 16K 条规则和 16 个不同位掩码的 HashTCAM（类似于 Broadcom Trident II[190]）只需要占用板上 19％的逻辑资源和 22％的 BRAM。主机管理程序尝试根据其位掩码对规则进行分组，并将具有大多数规则的组放入 HashTCAM，然后将不适合 HashTCAM 中任何组的其余规则放入基于 BRAM 的 TCAM 中。

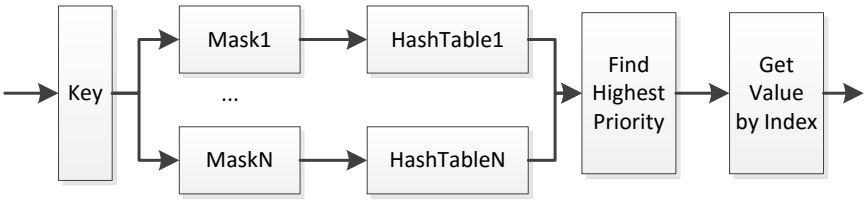


图 4.20　HashTCAM。

本实验比较 OpenFlow 防火墙与 Linux 防火墙以及 Click + DPDK [191]。对于 Linux，使用 IPSet 来处理完全匹配规则，同时将 IPTable 用于通配符规则。作为参考的还包括 Dell S6000 交换机的性能，该交换机具有有限的防火墙功能并支持 1.7K 通配符规则。值得注意的是，最初的 Click + DPDK [191] 不支持接收端缩放（RSS）。本章修复了这个问题，并发现当使用 4 个内核时，Click + DPDK 已经达到了最佳性能。但对于 Linux，使用尽可能多的内核（由于 RSS 限制，最多 8 个内核）可以获得最佳性能。

图 4.21（a）显示了具有不同数量的通配符规则的不同防火墙的数据包处理速率。数据包大小为 64B。可以看到，ClickNP 和 S6000 都可以达到 56.4 Mpps 的最大速度。Click + DPDK 可以达到约 18 Mpps。由于 Click 使用静态分类树来实现通配符匹配，因此处理速度不会随插入规则的数量而变化。Linux IPTables 具有 2.67 Mpps 的低处理速度，并且随着规则数量的增加速度降低。这是因为 IPTables 为通配符规则执行线性匹配。

图 4.21（b）显示了使用小数据包（64B）和 8K 规则的不同负载下的处理延迟。由于每个防火墙具有显着不同的容量，因此将负载因子标准化为每个系统的最大处理速度。在所有负载水平下，FPGA（ClickNP）和 ASIC（S6000）解决

方案具有 μs 级别的延迟（ClickNP 为 1.23 μs，S6000 为 0.62 μs），方差非常小（ClickNP 为 1.26 μs，对于 S6000 为 95%（百分位数）为 0.63 μs）。但是，软件解决方案具有更大的延迟，并且方差也更大。例如，使用 Click + DPDK，当负载很高时，延迟可能高达 50μs。图 4.21（c）显示了不同数据包大小和 8K 规则的处理延迟。使用软件解决方案，延迟随数据包大小的增加而增加，这主要是由于要复制的内存较大。相比之下，FPGA 和 ASIC 保持相同的延迟，而与数据包大小无关。在所有实验中，ClickNP 防火墙的 CPU 使用率非常低（单个 CPU 核的 < 5%）。

最后，图 4.21（d）显示了已有 8K 规则时的规则插入延迟。Click 的静态分类树需要事先了解所有规则，而生成 8K 规则的树需要一分钟。IPTables 规则插入需要 12 ms，这与表中现有规则的数量成比例。Dell S6000 中的规则插入需要 83.7 μs。对于 ClickNP，在 HashTCAM 表中插入一个规则需要 6.3 至 9.5μs 用于 2 至 3 次 PCIe 往返，而 SRAM TCAM 表平均需要 44.9 μs 来更新 13 个查找表。ClickNP 数据平面吞吐量在规则插入期间不会降低。结论是 ClickNP 防火墙在数据包处理方面具有与 ASIC 类似的性能，但相比 ASIC 具有更好的灵活性和可重构性。

4.6.3　IPSec 网关

软件网络功能的一个问题是，当数据包需要一些计算密集型处理时，CPU 很快就会成为瓶颈，例如，IPSec[36]。IPSec 数据面需要使用 AES-256-CTR 加密和 SHA-1 身份验证处理 IPSec 数据包。如 §4.5.2 所示，单个 AES_CTR 元件只能实现 27.8 Gbps 的吞吐量。因此，需要两个 AES_CTR 元件并行运行以实现线速率。然而，SHA-1 很棘手。SHA-1 将数据包分成较小的数据块（64B）。虽然一个数据块中的计算可以流水线化，但是一个 IP 数据包内的连续块之间存在依赖关系 - 下一个块的计算无法在前一个块完成之前开始！如果按顺序处理这些数据块，吞吐量将低至 1.07 Gbps。幸运的是，可以利用不同数据包之间的并行性。虽然当前数据包的数据块的处理仍在进行，但提供了不同数据包的数据块。由于这两个数据块没有依赖关系，因此可以并行处理它们。为了实现这一点，本文设计了一个名为 *reservo* 的新元件（保留站的简称），它可以缓冲多达 64 个数据包，并为 SHA-1 元件调度独立的块。在计算了一个数据包的签名之后，*reservo* 元件将它发送到将 SHA-1 HMAC 附加到数据包的下一个元件。

还有一件棘手的事情。虽然 SHA-1 元件具有固定的延迟，但数据包的总延迟是不同的，即与数据包大小成比例。当在 SHA-1 计算中调度多个数据包时，这些数据包可能发生重排，例如，大数据包后面的较小数据包可能更早地完成。为了使输出的数据包与输入顺序相同，在 SHA-1 元件之后进一步添加了一个 *reorder*

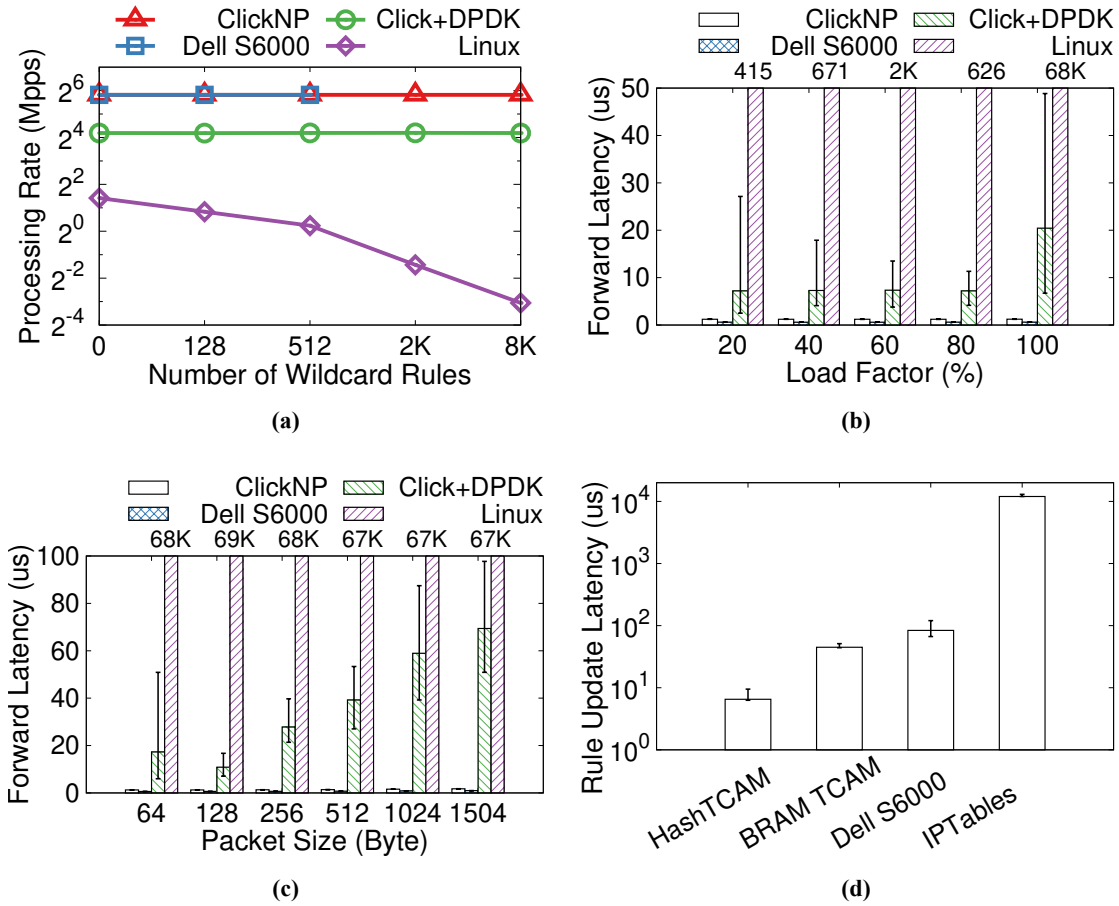


图 4.21　防火墙。误差条（error bar）表示 5% 和 95% 分位的延迟。在 (a) 和 (b) 图中，数据包大小为 64 字节。

buffer 元件，该元件存储乱序的数据包并根据数据包的序列号恢复原始顺序。图 4.22 显示了 IPSec 网关的元件结构。

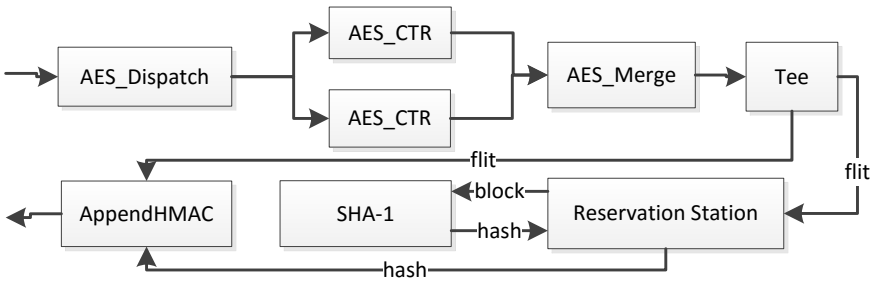


图 4.22　IPSec 网关的元件架构。

下面比较 IPSec 网关和 StrongSwan [192]，使用相同的密码套件 AES-256-CTR 和 SHA1。在单个 IPSec 隧道的情况下，图 4.23（a）显示了不同数据包大小的吞吐量。对于所有规模，IPSecGW 实现了线路速率，即 64B 数据包为 28.8 Gbps，1500B 数据包为 37.8 Gbps。然而，StrongSwan 最多只能达到 628 Mbps，随着数据包变小，吞吐量也会降低。这是因为尺寸越小，需要处理的数据包数量就越

多，因此系统需要计算更多的 SHA1 签名。图 4.23（b）显示了不同负载因子下的延迟。同样，IPSecGW 产生的恒定延迟为 13 μs，但是 StrongSwan 会产生更大的延迟和更高的方差，最长可达 5 ms!

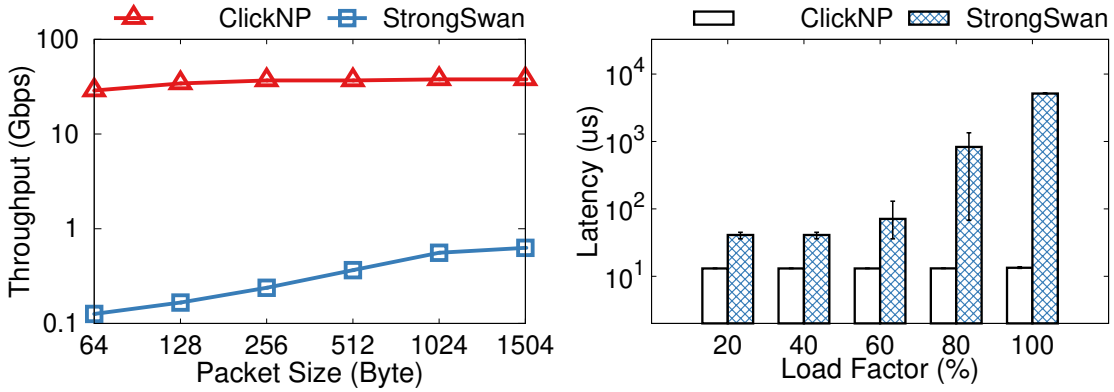


图 4.23　IPSec 网关。

4.6.4　L4 负载平衡器

L4 负载平衡器根据 Ananta [7] 中的 *multiplexer*（MUX）实现。MUX 服务器基本上查看数据包标头，并查看是否已为该流分配了直接地址（DIP）。如果是，则通过 NVGRE 隧道将分组转发到由 DIP 指示的服务器。否则，MUX 服务器将调用本地控制器为流分配 DIP。MUX 服务器中需要按流状态。由于存在故障并且避免黑洞需要即时更改后端服务器列表，因此无法使用基于散列的 ECMP。此外，高级 LB 还可能需要负载感知平衡。流表用于记录流向其 DIP 的映射。为了处理数据中心的大流量，它需要 L4LB 在流表中支持多达 3200 万个流。显然，如此大的流量表不能适应 FPGA 的 BRAM，必须存储在板载 DDR 存储器中。但是，访问 DDR 内存很慢。为了提高性能，在 BRAM 中创建了一个带有 16K 高速缓存行的 4 路关联流缓存。最近最少使用（LRU）算法用于替换流缓存中的条目。

如图 4.24，传入的数据包首先传递一个解析器（parser）元件，该元件提取 5 元组并将它们发送到流缓存（flow cache）元件。如果在流缓存中找不到流，则将数据包的元数据转发到全局流表，该表读取 DDR 中的完整表。如果仍然没有匹配的条目，则该数据包是流的第一个数据包，并且请求被发送到 *DIPAlloc* 元件，以根据负载平衡策略为该流分配 DIP。确定 DIP 后，将一个条目插入到流表中。

在确定分组的 DIP 之后，封装元件将检索下一跳信息，例如 IP 地址和 VNET ID，并相应地生成 NVGRE 封装的分组。对于流的剩余分组，从流状态提取 DIP。如果收到 FIN 数据包或发生超时，则流条目将无效在从流中接收任何新数据包

之前。在确定 DIP 之后，从 BRAM 检索下一跳元数据并封装 NVGRE 头以将分组引导到分配的 DIP。

除了 *DIPAlloc* 元件之外，所有元件都放在 FPGA 中。由于只有流的第一个数据包可能会出现 *DIPAlloc* 并且分配策略也可能很复杂，因此更适合运行 CPU 上的 *DIPAlloc*，是联合 CPU-FPGA 处理的另一个例子。

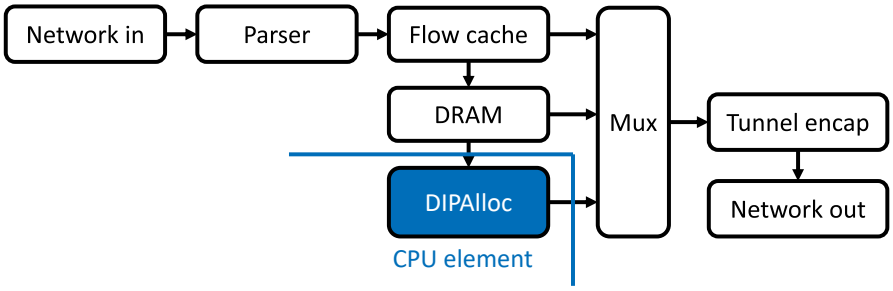


图 4.24　L4 负载均衡器的元件架构。

下面比较 L4LB 与 Linux 虚拟服务器（LVS）[193]。为了对系统进行压力测试，使用 64B 数据包生成大量并发 UDP 流，目标是一个虚拟 IP（VIP）。图 4.25（a）显示了具有不同并发流数的处理速率。当并发流量小于 8K 时，L4LB 达到 51.2Mpps 的线路速率。但是，当并发流的数量变大时，处理速率开始下降。这是因为 L4LB 中的流缓存未命中。当流缓存中缺少流时，L4LB 必须访问板载 DDR 内存，这会导致性能下降。当流量太多时，例如 32M，缓存未命中占主导地位且对于大多数数据包而言，L4LB 需要一次访问 DDR 内存。因此处理速度降低到 11Mpps。在任何情况下，LVS 的处理速率都很低。由于 LVS 将 VIP 关联到仅一个 CPU 核心，因此其处理速率必须达到 200Kpps。

图 4.25（b）显示了不同负载条件下的延迟。在这个实验中，将并发流的数量固定为 100 万。可以看到，L4LB 实现了 4 μs 的非常低的延迟。然而，LVS 会导致约 50 μs 的延迟。当吞吐量负载高于 100Kpps 时，排队延迟会迅速上升，超过 LVS 的处理能力。

最后，图 4.25（c）比较 L4LB 和 LVS 接受新流的能力。这个实验指示 PktGen 生成尽可能多的单包微流（micro-flow）。可以看到，L4LB 每秒可以接受高达 10M 条新连接。由于单个 PCIe 插槽每秒可以传输 16.5M 次数据，因此瓶颈仍然是 DDR 访问。为简单起见，本文中 *DIPAlloc* 元件轮流（round-robin）分配 DIP。对于复杂的分配算法，*DIPAlloc* 的 CPU 核将成为瓶颈，并且可以通过在更多 CPU 核心上复制 *DIPAlloc* 元件来提高性能。对于 LVS，由于数据包处理能力有限，它每秒最多只能接受 75K 个新连接。

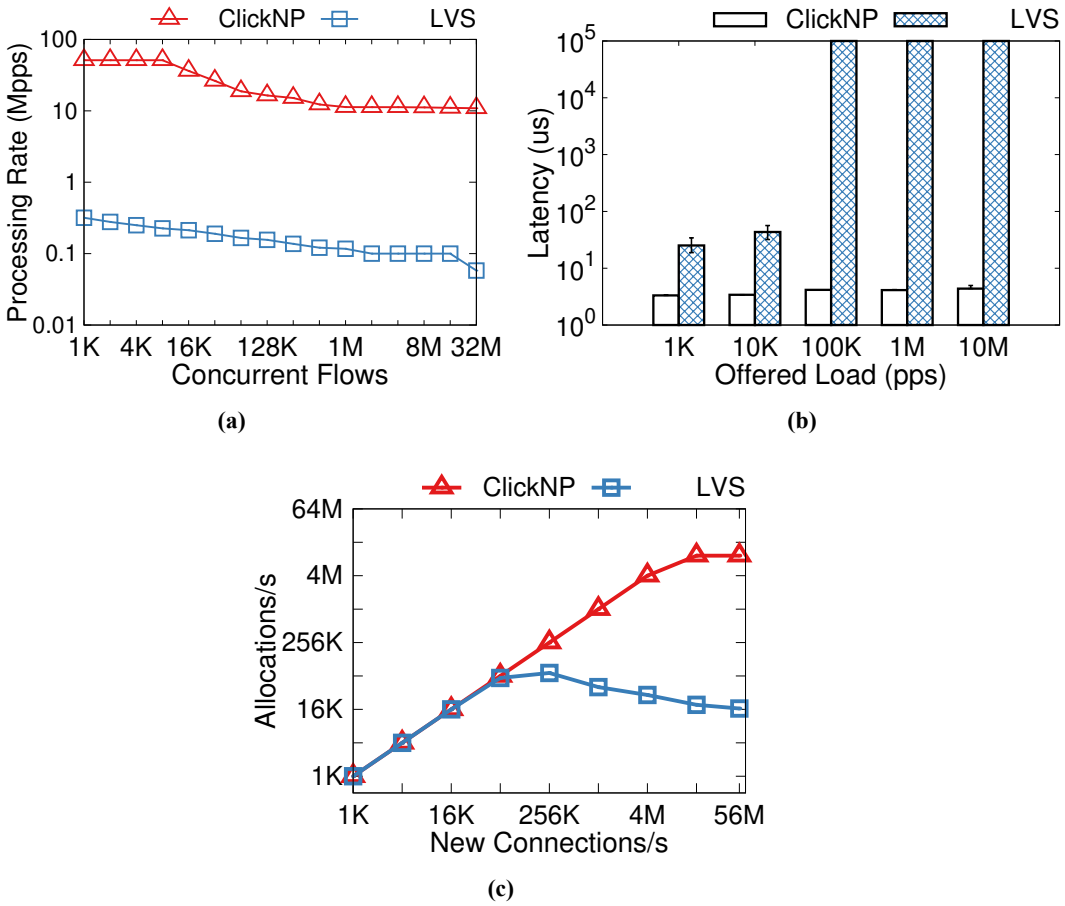


图 4.25　L4 负载均衡器的性能评估。

4.6.5　pFabric 流调度器

ClickNP 也是网络研究的好工具。由于灵活性和高性能，ClickNP 可以快速制作最新研究的原型并将其应用于真实环境。

本节使用 ClickNP 来实现一个最近提出的数据包调度规则–pFabric[167]。pFabric 调度很简单。它只保留浅缓冲区（32 个数据包），并始终使具有最高优先级的数据包出列。当缓冲区已满时，优先级最低的数据包将被丢弃。pFabric 在数据中心中实现了接近最优的流完成时间。在原始论文中，作者提出使用二进制比较树（Binary Comparison Tree，BCT）来选择具有最高优先级的数据包。但是，虽然 BCT 只需要 $O(log_2 N)$ 个周期来计算最高优先级的数据包，但在相邻两次选择过程之间存在依赖关系。这是因为只有当前一个选择完成后才能知道最高优先级的数据包，然后才能可靠地启动下一个选择过程。这种限制要求时钟频率至少为 300MHz 才能实现 40Gbps 的线速，这对当前的 FPGA 平台来说是不可能的。

本文使用不同的方式来实现 pFabric 调度程序，它更容易并行化。该方案基于*移位寄存器优先级队列*（shift register priority queue）[194]。如图 4.26，条目以非

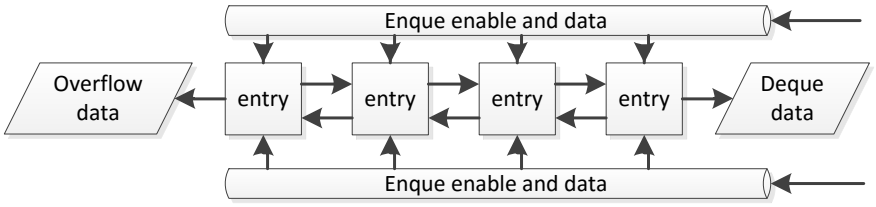


图 4.26　移位寄存器优先级队列。

增加优先级顺序保存在 *K* 个寄存器中。出列时，所有条目都向右移动并弹出头部。这只需要 1 个周期。对于入队操作，新数据包的元数据将转发到所有条目。现在，对于每个条目，可以在条目中的分组，新分组和相邻条目中的分组之间执行本地比较。由于所有局部比较都可以并行进行，因此入队操作也可以在 1 个周期内完成。入队和出队操作可以进一步并行化。因此，可以在一个周期中处理分组。图 4.27 显示了 pFabric 流调度器的元件架构。

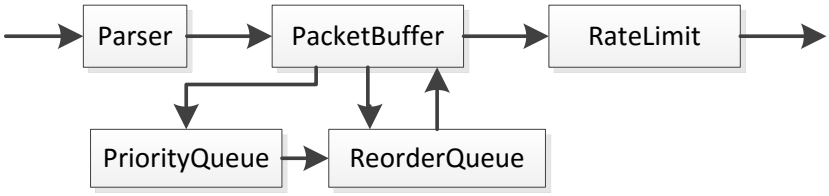


图 4.27　pFabric 流调度器的元件架构。

本实验修改了一个软件 TCP 流生成器[195]，以便在数据包有效负载中放置流优先级，即流的总大小。本实验根据[167] 中的数据挖掘工作负载生成流，并使用 *RateLimit* 元件将限制出口端口进一步设置为 10 Gbps。应用 pFabric 根据流优先级调度出口缓冲区中的流量。图 4.28 显示了 pFabric 和具有 Droptail 队列的 TCP 的平均流完成时间（FCT）和理想值。该实验验证了 pFabric 在这种简单的场景中实现了接近最优的 FCT。

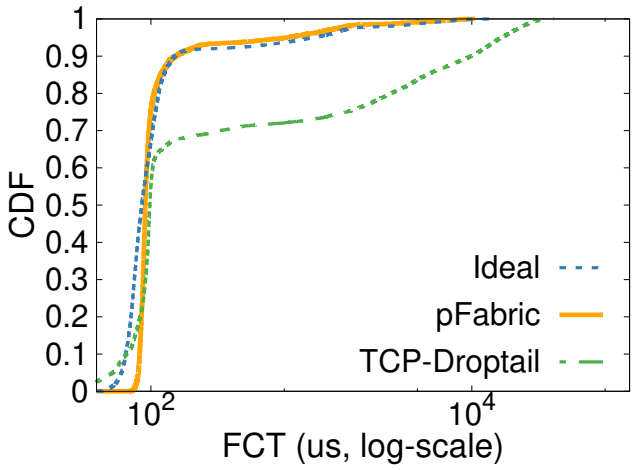


图 4.28　pFabric 的验证。

4.6.6　容错的 EPC SPGW

LTE 核心网（EPC）的数据面使用 S-GW 和 P-GW 处理网络数据包，其处理流程与 IPSec 网关类似，如图 4.29 所示。EPC SPGW 需要为每个 bearer（即用户）保存状态，每个数据包经过时都会修改 bearer 的状态。EPC SPGW 不仅需要高吞吐量、低延迟，还需要高容错性，即硬件故障对用户不可感知，且状态不会丢失。

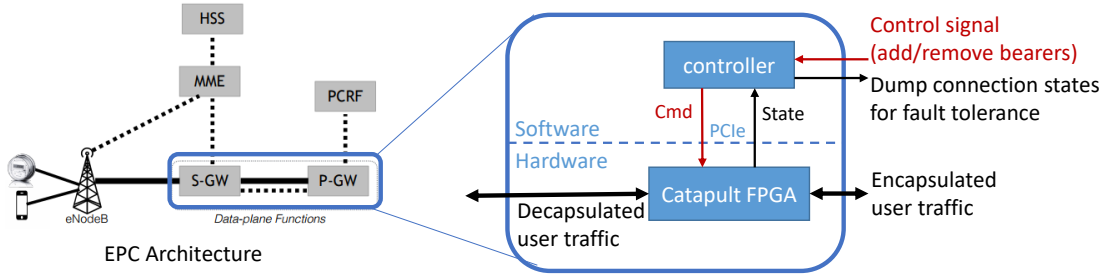


图 4.29　LTE 核心网（EPC）加速架构。

高容错性采用第 4.5.5 节的状态机复制方法实现，元件结构如图 4.30 所示。每个用户需要约 300 字节的状态，FPGA 的片上内存可以缓存约 4K 用户的状态，而板上 BRAM 可以保存 10M 用户的状态，因此一块 FPGA 就可以支持最多 10M 个并发连接。在用户访问服从幂律分布的通常情况下，FPGA 的吞吐量可达 40 M 数据包每秒（pps）；在最坏情况下，由于缓存不命中，吞吐量可达 20 M pps。在正常情况下，FPGA 的 95% 延迟为 4 μs。控制平面通过 PCIe I/O 管道实现，95% 控制平面延迟为 1 μs，吞吐量为每秒 1M 次添加或删除用户操作。当添加一个新的备份节点时，需要将原有节点的用户状态备份到新节点，在网络空载时，该状态迁移过程可以充分利用 40 Gbps 带宽，仅需 0.8 秒即可实现状态复制。

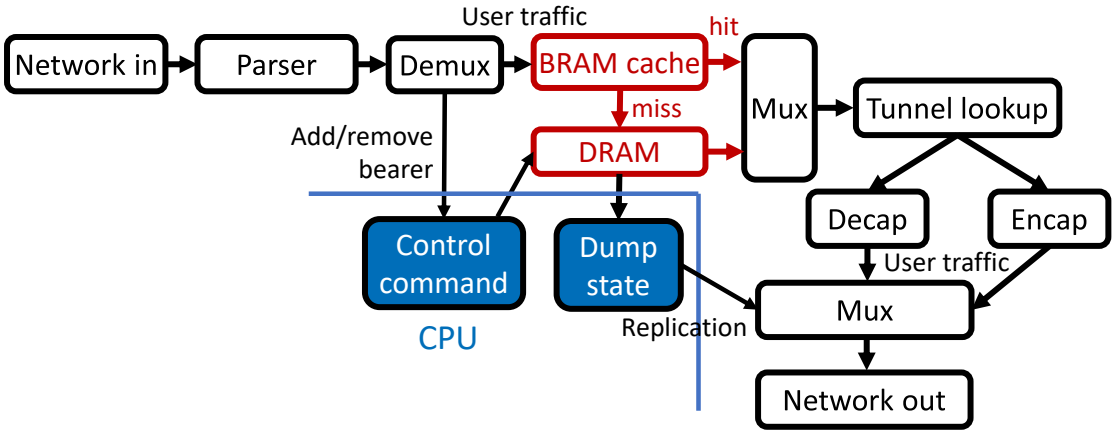


图 4.30　LTE SPGW 的元件结构。

4.7　讨论：资源利用率

本节将评估 ClickNP 网络功能的资源利用率。表 4.4 总结了结果。除了使用大多数 BRAM 来保存码表（code book）的 IPSec 网关之外，所有其他网络功能仅使用中等数量的资源（5 至 50 %），仍有空间容纳更多的网络功能。

表 4.4　ClickNP 网络功能汇总。

Network Function	LoC†	#Elements	LE	BRAM
Pkt generator	665	6	16%	12%
Pkt capture	250	11	8%	5%
OpenFlow firewall	538	7	32%	54%
IPSec gateway	695	10	35%	74%
L4 load balancer	860	13	36%	38%
pFabric scheduler	584	7	11%	15%

† 所有元件描述语言的代码行数与配置文件的行数之和。

接下来研究 ClickNP 的细粒度模块化的开销。由于每个元件都将生成逻辑块边界并仅使用 FIFO 缓冲区与其他块进行通信，因此应该存在开销。为了衡量这种开销，创建一个只将数据从一个输入端口传递到输出端口的简单“空”元件。此空元件的资源利用率可以很好地体现模块化的开销。不同的高层次综合工具可能使用不同数量的资源，但都很低，最小值为 0.15 %，最大值为 0.4 %。因此，细粒度模块化引入的开销很小。未来的工作还可以探索将计算流图中相邻的元件进行融合，以进一步降低元件的模块化开销。

最后研究 ClickNP 与手写硬件描述语言相比生成的硬件描述语言代码的效率。为此，使用 NetFPGA[138] 作为参考。首先，提取 NetFPGA 中的关键模块，这些模块由经验丰富的 Verilog 程序员进行了优化，并在 ClickNP 中实现具有相同功能的对应元件。然后，使用不同的高层次综合工具作为后端，比较这两种实现之间的相对面积成本。结果总结在表 4.5 中。由于不同的工具可能具有不同的面积成本，因此记录最大值和最小值。可以看到，与手工优化代码相比，自动生成的硬件描述语言代码使用更多区域。然而，差异并不是很大。对于复杂模块（如表格顶部所示），相对面积成本小于 2 倍。对于微小模块（如表格底部所示），相对面积成本看起来更大，但绝对资源使用量很小。这是因为现有的高层次综合工具都会为每个元件生成固定的控制逻辑，产生面积开销。

总之，ClickNP 可以为 FPGA 生成高效的硬件描述语言，只需要适度的面积成本，可以构建实用的网络功能。展望未来，FPGA 技术仍在迅速发展。例如，Intel 的 Arria 10 FPGA 和最新的 Stratix 10 FPGA 的面积分别是本文使用的芯片（Stratix V）的 2.5 倍和 10 倍。因此，高层次综合的面积成本将来将受到较少的

表 4.5　相比 NetFPGA 的面积开销。

NetFPGA 功能	逻辑查找表（LUT）最小 / 最大	寄存器 最小 / 最大	内存（BRAM）最小 / 最大
输入选择器	2.1x / 3.4x	1.8x / 2.8x	0.9x / 1.3x
输出队列	1.4x / 2.0x	2.0x / 3.2x	0.9x / 1.2x
数据包头解析器	0.9x / 3.2x	2.1x / 3.2x	N/A
Openflow 查找表	0.9x / 1.6x	1.6x / 2.3x	1.1x / 1.2x
IP 校验和计算	4.3x / 12.1x	9.7x / 32.5x	N/A
隧道封装	0.9x / 5.2x	1.1x / 10.3x	N/A

关注。

4.8　扩展：计算密集型应用

一些网络功能是计算密集的，例如第 4.6.3 节的 IPSec 网关需要对每个数据包的内容进行加密和签名。由于一个数据包的计算量过大，将整个计算流图完全展开成一条流水线的面积将超过 FPGA 的容量。因此需要*用时间换空间*，对计算流图进行切分。由于计算过程中存在依赖关系，第 4.6.3 节展示了如何利用*保留站*将数据包的计算分为多个阶段，保存中间状态，并充分利用不同连接间的并行性。本节讨论两个计算量更大的应用：HTTPS RSA 加速和神经网络推断，以展示实现计算密集型应用的一般方法。

4.8.1　HTTPS RSA 加速

HTTPS 是用于与 Web 服务进行安全连接的协议。随着用户对隐私的日益关注，越来越多的 Web 服务通过 HTTPS 提供访问。2010 年以来，HTTPS 流量的比例每 6 个月增长 40%，2016 年已经占到所有网络连接的 40% 以上。

HTTPS 提供三种机制来确保安全性。首先，当连接建立时，它使用验证 Web 服务器的身份，并创建双方的共享密钥。其次，它加密用户和 Web 服务器之间的数据传输。第三，它检查数据的完整性。在这三种机制中，连接建立是计算密集程度最高的部分，因为它需要进行非对称密钥操作（如 RSA 算法）。

如果不启用 HTTPS，单个 CPU 核心每秒可处理超过 7,000 个 HTTP 请求。使用 HTTPS 时，由于连接设置中的 TLS 握手，吞吐量下降到 1/35。TLS 握手的高计算开销一直是高流量网站部署 HTTPS 以确保安全性的主要障碍。

在 TLS 握手中，Web 服务器使用 RSA 私钥执行解密。如图 4.31，RSA 解密在数学上是大整数的幂取模运算，其中底数、指数和模都是大整数。通过对指数进行二进制分解，大整数幂取模运算可用迭代多次的乘法取模运算实现。乘法

取模运算又可以通过 Montgomery 算法转化为 3 次乘法、1 次加法和 1 次减法运算。对于 2048 位私钥，解密需要大约 800 万次 16 位整数乘法运算。尽管 TLS 证书解析和协议处理也很复杂，但需要的计算次数比解密少得多。因此，我们修改 OpenSSL 库以将 RSA 解密操作卸载到 FPGA，并将其他部分保留在 CPU 上。

显然，将整个 RSA 解密的过程完全展开成数字逻辑将占用过大的芯片面积，因此需要用时间换空间，构建较小规模的乘法和加法阵列，重复多次计算。对于 2048 位密钥，即使是最内层的 1024 位大整数乘法器都已经超过了 FPGA 的 DSP 数量限制，因此需要将大整数乘法进一步切分。在 Montgomery 算法中，3 次乘法显然可以复用同一乘法器阵列。① 图 4.31 中的模乘（ModMulti）控制器在乘法器、加法器和减法器之间搬移数据；二进制幂运算（Binary Exp）控制器在多次模乘运算之间搬移数据。此外，由于运算部件存在流水线延迟，而一次 RSA 解密的计算间存在依赖，如果仅进行一次 RSA 解密，很多运算部件将闲置。为了充分利用 FPGA 的并行性，需要多个 RSA 解密任务并发处理。模乘和二进制幂运算控制器需要管理并发任务的中间数据和执行状态，并调度不相关的子任务到计算部件。

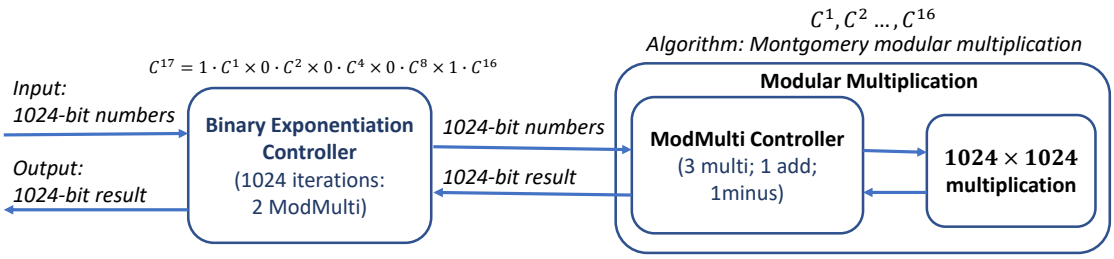


图 4.31　HTTPS 加速器的元件结构（简化图）。

显然，手动实现控制器的缓存管理、任务切分、并发控制、数据搬移等逻辑是非常繁琐的。为此，我们希望从图 4.32 所示的源代码自动生成控制器代码。

首先，ClickNP 从最内层循环开始，尝试展开尽可能多层的循环，以并行执行尽可能多的计算。在有多个并列循环的情况下（如图 4.32 中的乘法和加法），各循环的展开次数需要与计算次数匹配②，使得各计算模块的吞吐量匹配。开发者可以为每个循环分别指定展开次数；对于循环次数在编译期静态可知的程序，也可以指定一个全局的展开次数作为计算强度最大循环的展开次数，由 ClickNP 自动匹配其他循环的展开次数。③ 展开循环次数的上界取决于 FPGA 的硬件容

①如果使用三个分离的乘法器阵列，则在总面积一定的条件下，每个乘法器阵列的并行度将变为 1/3，而由于 3 次乘法之间是有数据依赖的，RSA 解密运算的延迟将增加到约 3 倍。因此，当并行度可调时，大多数情况下将多个不能并行执行的元件合并为一个并行度更大的元件是有益的。

②如两个循环分别有 512 和 1024 次，则两个循环的展开次数比例为 1:2。

③对于循环次数不确定的程序，开发者可以将其手工划分为若干个静态区域，并为每个区域指定所需的吞吐量或面积目标。

```
template<T> uint##(T*2) Karatsuba(uint##T a, b) {
    if (T > 256) { // karatsuba multiplication
        const T' = T/2;
        uint##T' a0 = a[T-1:T'], a1 = a[T'-1:0];
        uint##T' b0 = b[T-1:T'], b1 = b[T'-1:0];
        uint##T m0 = Karatsuba<T'>(a0, b0);
        uint##T m1 = Karatsuba<T'>(a1, b1);
        uint##T m2 = Karatsuba<T'>(a0 + a1, b0 + b1);
        return (m0 << T) + ((m2 - m0 - m1) << T') + m1;
    }
    else { // simple school book multiplication
        uint32 t[T*2/16];
        for (i=0; i<T; i+=16)
            for (j=0; j<T; j+=16)
                t[(i + j) / 16] += a[i+15:i] * b[j+15:j];
        uint##(T*2) result;
        for (i=0; i<T*2; i+=16) {
            t[i+1] += t[i][31:16];
            result[i+15:i] = t[i][15:0];
        }
    }
}
uint1024 ModMulti(uint1024 a, b, m, m') {
    uint2048 t = Karatsuba<1024>(a, b);
    uint1024 n = Karatsuba<1024>(t[1023:0], m')[1023:0];
    uint2048 sum = Add<2048>(t, Karatsuba<1024>(m * n));
    uint1024 s = sum[2047:1024];
    int1024 diff = Subtract<1024>(s, n);
    return IsPositive<1024>(diff) ? diff : s;
}
uint1024 ModExp(uint1024 a, e, m, m') {
    uint1024 square = a, result = 1;
    for (i = 0; i < 1024; i++) {
        if (e[i])
            result = ModMulti(result, square, m);
        square = ModMulti(square, square, m);
    }
    return result;
}
```

图 4.32　2048 位 RSA 解密所用大整数幂取模运算的 ClickNP 示意代码。ClickNP 模板（template）是用于生成递归结构代码的语法糖，在编译时会被展开，并消除无效代码。没有被声明为 inline 的函数被实现成独立的元件，就像 async 原语。

量，开发者可以根据高层次综合工具报告的资源估计或者综合、布局布线后的结果来配置。一些程序的多层循环顺序可交换，切分不同层次循环所生成代码的数据局部性不同，从而所需的数据搬移数量不同。开发者在编写 ClickNP 代码时，应将并行度最高、数据局部性最强的循环放到最内层，这样编译器展开内层循环时，可以降低数据搬移开销。

接下来，ClickNP 生成控制器代码。为了生成延迟隐藏的并发执行代码，ClickNP 用高层次综合工具编译计算部件（即展开后的循环）以及片上内存与计算部件之间数据搬移的代码，以计算其延迟和吞吐量。并发执行的任务数量即

为延迟乘以吞吐量。控制器将每个任务的当前执行状态和中间数据保存在片上内存中，并向计算部件调度拆分后的子计算任务。

表 4.6　2048 位 RSA 解密的性能。

	CPU	Stratix V FPGA	Arria 10 FPGA
吞吐量	1.2K	6K	12K
延迟	0.85ms	3.5ms	1.75ms

如表 4.6，对 2048 位 RSA 解密运算，FPGA 是单个 CPU 核性能的 5 至 10 倍，但延迟是 CPU 的 2 至 4 倍。这是因为 RSA 解密运算中的一大部分难以并行化，FPGA 的主频远低于 CPU。但 FPGA 可以充分利用任务级并行来提高吞吐量。

在使用 FPGA 加速前，单个 CPU 核的 Nginx 服务器每秒可以处理约 900 次 HTTPS 握手。利用 FPGA 加速后，单个 CPU 核每秒可以处理约 3.5K 次 HTTPS 握手。

4.8.2　神经网络推断

神经网络推断是数据中心越来越重要的计算密集型场景。传统上，由于 FPGA 的开发成本高，基于 FPGA 的神经网络加速器往往使用固定的硬件逻辑，不同的神经网络映射到不同的微指令程序。然而，这种方法没有发挥 FPGA 可重构的优势。为多样的神经网络定制 FPGA 硬件逻辑的关键是高层次综合技术，它可以从算法描述自动生成硬件逻辑。事实上，由于 FPGA 可以实现几乎是任意的数据通路、流水线和计算单元，在计算不规则或存在复杂依赖的场景下，定制化的硬件逻辑比编译到基于 SIMD 的定制化指令更高效。

一个典型的推理神经网络由若干稀疏层和稠密层组成。稀疏层主要是从较大的特征数组中随机读取，而稠密层主要进行矩阵或向量乘法。每个稀疏层和稠密层用一个 ClickNP 元件表示，而神经网络的计算流图就是 ClickNP 的元件间连接。如果将每个元件都实现为单独的硬件逻辑，则片上资源将会碎片化。对于一个特定的任务，同一时刻仅有少量计算元件在运行，从而单个任务的延迟将较长。因此，需要从不同的元件中提取公共计算部件。对网络功能，ClickNP 很难进行这项优化，因为网络功能的各种元件所进行的计算类型不同，难以提取公共部分。

提取公共计算部件的方法是比较不同元件中循环的同构性，即是否可以通过替换所访问数组名和循环变量名的方式将两个循环的抽象语法树变得完全相同。如果若干不同元件中的循环同构，就可以将这些重构的循环抽取成一个新的 `async` 元件。例如，各稀疏层中访问全局内存可被提取成一个元件，各稠密层中的矩阵与向量乘法可被提取成一个元件，各稠密层中的 relu 激活函数可被提取

成另一个元件。接下来，与 RSA 解密类似，ClickNP 尽可能展开内层循环并生成控制器代码。由图 4.7 可见，循环展开的并行度越高，神经网络推理的延迟越低。

表 4.7　不同稠密层并行度下神经网络推理的延迟。所用的神经网络由三个稀疏层和四个稠密层组成，三个并行执行的稀疏层获取的特征与输入的其他特征拼接在一起后进入稠密层。四个稠密层组成流水线，其中每层由矩阵与向量乘法、激活函数、归一化函数组成。

稠密层并行度	每样本延迟 (μs)
8	188.2
16	98.2
32	49.3
64	26.1
128	17.5
256	资源不足

最后需要指出，对计算密集型任务，ClickNP 所采用的高层次综合方法尽管开发效率比手动编写底层 FPGA 代码大大提高，其面积开销可能显著高于手动优化的 FPGA 代码。幸运的是，网络数据包处理任务的性能主要受限于网络带宽，多数情况下 FPGA 面积不是重要的考虑。

4.9　本章小结

本章介绍了 ClickNP，这是一个 FPGA 加速平台，用于商用服务器中高度灵活和高性能的网络功能。ClickNP 使用高级语言完全可编程，并提供网络领域软件程序员熟悉的模块化架构。ClickNP 支持联合 CPU / FPGA 数据包处理并具有高性能。评估表明，与最先进的软件网络功能相比，ClickNP 将网络功能的吞吐量提高了 10 倍，将延迟降低了 10 倍。本章提出了一个具体案例，表明 FPGA 能够加速数据中心的网络功能。本章证实了在 FPGA 上进行高级语言编程实际上是可行和实用的。

第 5 章　KV-Direct 数据结构加速

5.1　引言

本章的主题是存储虚拟化与数据结构处理加速，在全文中的位置如图 5.1 所示。其中数据结构处理加速是本文的重点，存储虚拟化仅在第 5.6.3 节中略作讨论。

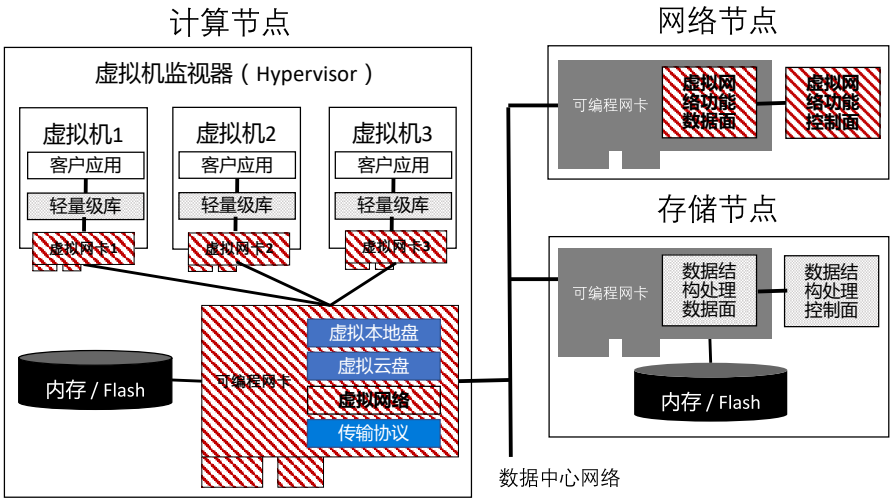


图 5.1　本章主题：存储虚拟化与数据结构处理加速，用粗斜线背景的方框标出。

在可编程网卡的编程方面，本章基于上一章提出的 ClickNP 编程框架，搭建了服务层的基础，提出了有状态处理和数据结构处理的框架，并基于此实现了内存键值存储，如图 5.2 所示。

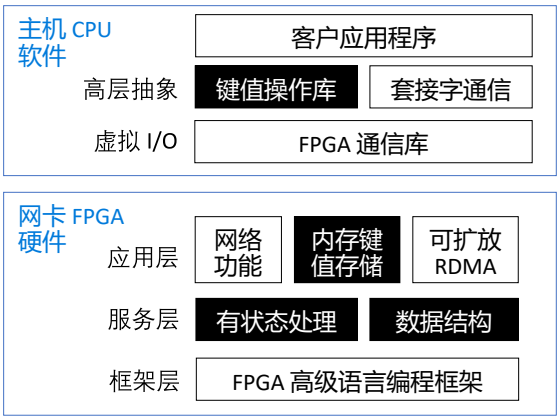


图 5.2　本章在可编程网卡软硬件架构中的位置。

内存键值存储是数据中心的关键分布式系统组件。本章提出 KV-Direct，一个基于可编程网卡的内存键值系统。顾名思义，KV-Direct 的可编程网卡从网络上接收并处理键值操作请求，并直接在主机内存中应用更新，绕过主机 CPU。KV-Direct 将 RDMA 原语从内存操作（读和写）扩展到键值操作（GET，PUT，

DELETE 和原子操作）。此外，为了支持基于向量的操作并减少网络流量，KV-Direct 还提供了新的向量原语 UPDATE，REDUCE 和 FILTER，允许用户定义活动消息（active message）[196] 并将某些计算委托给可编程网卡。

在可编程网卡内进行键值处理的设计重点是优化网卡和主机内存之间的 PCIe 流量。KV-Direct 采用一系列优化来充分利用 PCIe 带宽和隐藏延迟。首先，KV-Direct 设计了一个新的哈希表和内存分配器，以利用 FPGA 的并行性，并最大限度地减少 PCIe DMA 请求的数量。平均而言，KV-Direct 每次 GET 操作仅使用接近一次 PCIe DMA 操作，每次 PUT 操作仅使用两次 PCIe DMA 操作。其次，为了保证键值存储的一致性，KV-Direct 设计了一个乱序执行引擎来跟踪操作依赖性，同时最大化独立请求的吞吐量。第三，KV-Direct 在 FPGA 中实现了基于硬件的负载分派程序和缓存组件，以充分利用板载 DRAM 带宽和容量。

基于上述优化，单网卡 KV-Direct 系统能够实现每秒高达 180 M 次键值操作，相当于 36 个 CPU 核心的吞吐量[31]。与最先进的 CPU 键值存储系统相比，KV-Direct 可将尾延迟降低至 10 μs，同时将能耗效率提高 3 倍。而且，KV-Direct 可以通过多个网卡实现接近线性的可扩展性。通过在单台商品服务器中使用 10 个可编程网卡，性能可达每秒 12.2 亿键值操作，这比现有系统提高了一个数量级。

KV-Direct 还支持高达 180 Mops 的通用原子操作，明显优于目前最先进的基于 RDMA 的系统中报告的性能：2.24 Mops[46]。原子操作的高性能主要归功于乱序执行引擎。乱序执行引擎可以高效地跟踪键值操作之间的依赖性，而不会阻塞流水线。

本章的其余部分安排如下。第 5.2 节介绍背景，并阐明设计目标和挑战。第 5.3 节描述 KV-Direct 的设计。第 5.5 节评估 KV-Direct 的性能。第 5.6 和 5.7 节讨论本文的扩展工作。第 5.8 节讨论相关工作。第 5.9 节总结。

5.2　背景

5.2.1　键值存储的概念

顾名思义，键值存储（key-value storage）保存了若干键值对（key-value pair）组成的无序集合。其中的键（key）和值（value）都是可变长度的任意字符串。在一个键值存储中，相同的键只能出现一次。键值存储的基本操作是 GET 和 PUT。GET 操作输入一个键，输出该键对应的值。PUT 操作输入一个键值对，将其保存到键值存储中。如果有相同的键，则将原有键值对删除，再保存新的键值对。为了高效地支持读写操作，键值存储通常基于哈希表实现。不论是 GET 还是 PUT 操作，都首先计算键的哈希值，并在哈希表中查找之。对于 PUT 操作，可能需

要为新的键值对分配内存，并释放同一个键的原有键值对占用的内存。在分布式系统中，键值存储通常作为服务，接收来自网络客户端的 GET 和 PUT 操作，并将处理结果通过网络发送回客户端。

5.2.2　键值存储的工作负载变化

从历史上看，诸如 Memcached[197] 的键值存储作为 Web 服务的对象缓存系统获得了普及。在内存计算的时代，键值存储超越了缓存，成为在分布式系统中存储共享数据结构的基础架构服务。许多数据结构可以在键值哈希表中表示，例如，NoSQL 数据库中的数据索引[65]，机器学习中的模型参数[81]，图计算中的节点和边[198-199] 和分布式同步中的序列号发生器[47,200]。未来，内存键值存储还可为无服务器计算（serverless computing）提供高性能的临时存储[94]。

工作负载从对象缓存转移到通用数据结构存储意味着键值存储的几个新的设计目标。

小键值的高批量吞吐量。内存计算通常以大批量访问小键值对，例如线性回归中的稀疏参数[34,199] 或图遍历中的所有邻居节点[198]，因此键值存储能够受益于批处理和流水线操作。

可预测的低延迟。对于许多数据并行计算任务，迭代的延迟由最慢的操作决定[29]。因此，控制键值存储的尾延迟非常重要。基于 CPU 的设计往往需要通过调整批量大小来解决延迟和吞吐量间的平衡[31]。此外，由于操作系统不规则的调度、难以预测的硬件中断和缓存不命中，CPU 处理时间在高负载下可能会有很大的波动[156]。

写入密集型工作负载下的高效率。对于缓存工作负载，键值存储的读取数量通常比写入更多[201]，但图计算[69]、参数服务器[81] 等分布式计算工作负载不再是这种情况。对于图中的 PageRank 计算[69] 或参数服务器中的梯度下降[81]，每个迭代周期，都要读取和写入每个节点或参数一次。键值存储需要提供相同数量的 GET（读）和 PUT（写）操作。序列号发生器（sequencer）[47] 需要原子的增量操作而不是只读操作。这些工作负载需要可以同时高效处理读写操作的哈希表结构。

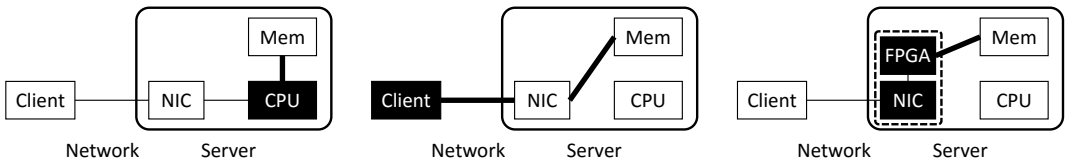
快速原子操作。几个非常流行的应用程序需要原子操作，例如集中式调度程序[202]，序列号发生器[47,200]，计数器[203] 和 Web 应用程序中的临时值[201]。这需要单个键上的高吞吐量原子操作。

向量操作。机器学习和图计算工作负载[81,198-199] 通常需要对向量中的每个元素进行操作，例如，将一个标量加到向量中的每个元素，或将向量归约为其元素的总和。没有向量支持的键值存储要求客户端为向量中的每个元素发出一个键值存储操作，或者将整个向量作为一个大键值对，取回客户端并执行操作。键

值存储如果支持向量数据类型和操作，就可以大大减少网络通信和 CPU 计算开销。

5.2.3　现有键值存储系统的性能瓶颈

构建高性能键值存储需要全局优化计算机系统中各种软件和硬件组件。按照数据结构在哪里处理，目前最先进的高性能键值存储系统基本上分为三类：在键值存储服务器的 CPU 上（图 5.3a）、在键值存储客户端上（图 5.3b）或者在硬件加速器上（图 5.3c）。



(a) 软件 / 双侧 RDMA。　**(b)** 单侧 RDMA。　**(c)** KV-Direct。

图 5.3　键值存储数据通路和处理装置的设计空间。行表示数据路径。一个键值操作（细线）可能需要多个基于地址的存储器访问（粗线）。黑框表示键值处理发生的位置。

当网络开销被缩减到极限时，高性能键值存储系统的吞吐量瓶颈可归因于键值操作中的计算和随机存储器访问中的延迟。基于 CPU 的键值存储需要花费 CPU 周期来进行键比较和哈希槽计算。此外，键值存储哈希表比 CPU 高速缓存的容量大几个数量级，因此内存访问延迟主要由实际访问模式下的缓存不命中延迟决定。

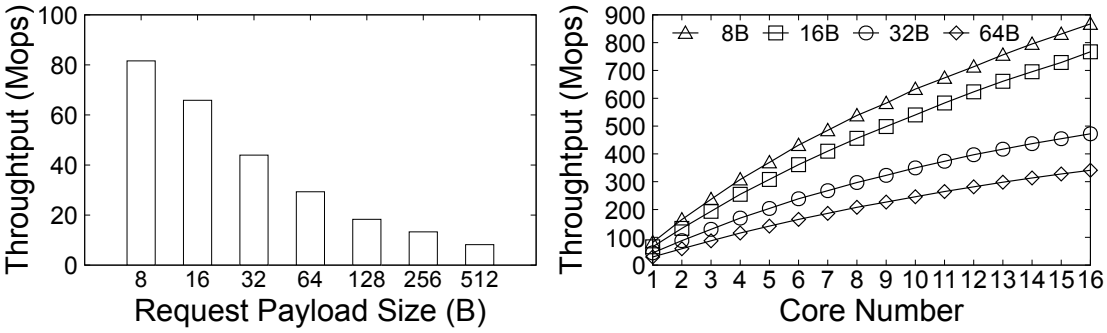
测量表明，当代计算机的 64 字节随机读取延迟大约为 100 ns ①。CPU 核心可以同时发出多个内存访问指令，受核心中加载存储单元（load-store unit）数量的限制（如 3 至 4 个）[204-206] ②。如图 5.4 和表 5.1 所示，在本文实验使用的 CPU 中，每个核心每秒最大吞吐量为 29.3 M 随机 64B 访问。另一方面，访问 64 字节键值对的操作通常需要大约 100 ns 计算或大约 500 条指令，这是无法放进指令窗口的 ③。当随机访存与计算交错时，由于指令窗口不足以覆盖访存延迟，CPU 内核的性能降低到每秒 5.5 M 键值操作（Mops）。一种优化方法是在一次发出内存访问之前，将多个键值存储操作的计算汇聚在一起，批量进行内存访问[31,207]。

①此随机读取延迟假定使用 4 KiB 正常页面大小，把 TLB 不命中和数据缓存行（cache line）不命中的延迟都考虑在内。

②尽管 CPU 微体系结构中每个核心可能有数十个加载存储单元，但 64 字节随机访存会产生多次 TLB 不命中和数据缓存行不命中，因此本文的实际测量中，一个访存延迟内只能完成 3 至 4 个随机访存操作。

③指令窗口是 CPU 乱序执行引擎可以重排的最大指令数量。如果一个访存指令之后有超过指令窗口数量的计算指令，而访存延迟大于执行指令窗口数量的计算指令的时间，那么由于指令窗口的限制，流水线在执行指令窗口数量的计算指令后就需要暂停（stall），等待访存结果返回，才能继续执行后续的计算指令。在我们使用的 CPU 上，指令窗口的大小的测量值约为 100 至 200。

此优化可以将本文使用的 CPU 的每核心键值操作吞吐量提高到 7.9 MOps，这仍远低于主机 DRAM 的随机 64B 吞吐量。



(a) 单核。　　**(b)** 多核。

图 5.4　CPU 随机内存访问性能。

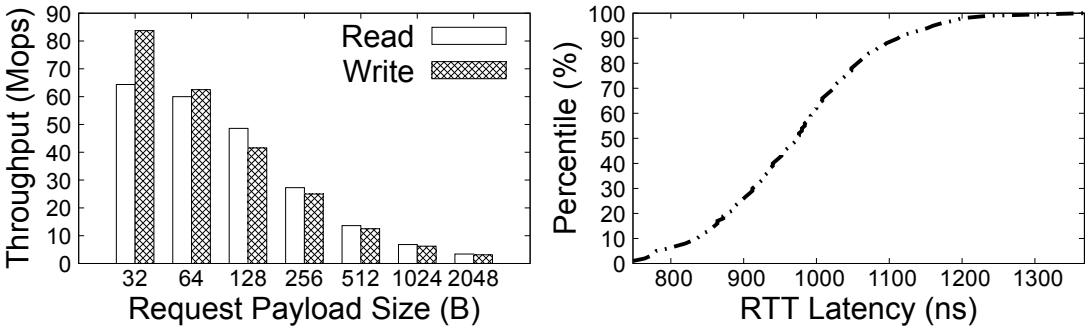
表 5.1　不同工作负载和内存访问粒度下的吞吐量（百万次操作每秒）。

大小（字节）	仅计算	仅访存	同时计算和访存（批处理大小）			
			1	2	3	4
32	24.1	44.0	7.5	11.1	13.1	14.1
64	11.1	29.3	5.5	6.7	7.6	7.9
128	5.4	18.3	3.5	4.1	4.3	4.1
256	2.7	13.2	2.1	2.2	2.2	2.1
512	1.3	8.2	1.2	1.1	1.2	1.1

观察到键值处理中CPU的有限容量，最近的工作利用单边（one-sided）RDMA将键值处理卸载到客户端。单边 RDMA 提供了远程访问共享内存的抽象。服务器端应用程序向本地 RDMA 网卡注册一块内存用于共享内存。客户端应用程序需要读写这块共享内存时，就向本地 RDMA 网卡发送 RDMA 读或写工作请求（work request）。客户端 RDMA 网卡会将工作请求转换成网络数据包发送给服务器端 RDMA 网卡。服务器端 RDMA 网卡将收到的数据包转换成 PCIe DMA 请求，访问共享内存，并将结果返回到客户端 RDMA 网卡。客户端 RDMA 网卡将读到的数据通过 PCIe DMA 发送到应用程序的内存缓冲区，再通过工作完成（work completion）通知应用程序。在这个过程中，服务器端的 RDMA 网卡处理读写请求，完全绕过服务器端的 CPU ①。

尽管 RDMA 网卡提供了高消息吞吐量（8 至 150 Mops[47]），但要找到 RDMA 原语和键值操作之间的高效匹配是一项挑战。对于写入（PUT 或原子）操作，可能

①在现代数据中心服务器体系结构中，这个说法是不严谨的，因为 PCIe 根（root complex）和内存控制器都在主机 CPU 内部，网卡通过 PCIe DMA 访问主机内存必须经过 CPU。本文的 “绕过 CPU” 遵从系统学术界的惯用说法，是逻辑上的含义，即绕过 CPU 核上软件的处理。本文的系统结构图中，CPU 也指代软件处理。下文还将多次出现 “绕过 CPU” 的说法，均取此义。



(a) DMA 吞吐量。　　**(b)** DMA 读延迟。

图 5.5　PCIe 随机 DMA 性能。

需要多个网络往返和多次内存访问来查询哈希索引，处理哈希冲突并分配可变大小的存储器。RDMA 不支持事务。为了保持数据结构的一致性，客户端必须相互同步，使用 RDMA 原子操作或分布式原子广播（distributed atomic broadcast）[208]。这两种方案都会产生通信开销和同步延迟[70,209]。因此，大多数基于 RDMA 的键值存储[46,70,209] 建议仅使用单边 RDMA 进行 GET（只读）操作。对于写入（PUT 或原子）操作，它们会回退到使用服务器 CPU 处理。因此，写入密集型工作负载的吞吐量仍然受限于服务器 CPU 的瓶颈。

5.2.4　远程直接键值访问面临的挑战

KV-Direct 将键值处理从 CPU 移动到服务器中的可编程网卡（图 5.3c）。与 RDMA 相同，KV-Direct 网卡通过 PCIe 访问主机内存。PCIe 是一种分组交换网络，具有大约 500 ns 的往返延迟和每 Gen3 x8 端点 7.87 GB/s 的理论带宽。在延迟方面，由于本文使用的 FPGA 硬核中有大约 300 ns 的额外处理延迟，可编程网卡通过 PCIe DMA 读取已被 CPU 缓存的主机内存，延迟为 800 ns。随机 DMA 读取未被缓存的主机内存时，由于 DRAM 访问、DRAM 刷新和 PCIe DMA 引擎中的 PCIe 响应重新排序，存在额外的 250 ns 平均延迟（图 5.5b）。在吞吐量方面，每个 DMA 读或写操作都需要一个带有 26 字节头的 PCIe 传输层数据包（TLP）和用于 64 位寻址的填充（padding）。对于以 64 字节粒度访问主机内存的 PCIe Gen3 x8 网卡，理论吞吐量因此为 5.6 GB/s 或 87 Mops。

为了使用 64 字节 DMA 请求使 PCIe Gen3 x8 接口的吞吐量饱和，考虑到 1050 ns 的延迟，需要 92 个并发 DMA 请求。然而，有两个实际因素进一步限制了 DMA 请求的并发性。首先，基于 PCIe 信用（credit）的流控（flow control）① 限制了每种 DMA 类型正在处理请求的数量。服务器中的 PCIe root complex（根

①在基于信用的流控中，接收端根据其接收缓冲区的容量，通告一定数量的信用。发送端每发送一定量的数据，就扣除相应的信用。信用不足时不能发送。这保证了接收端的缓冲区不会溢出。

节点）为 DMA 无响应（posted）[①] 操作通告了 88 个 PCIe 传输层数据包（TLP）的信用 [②]，为 DMA 有响应（non-posted）操作通告了 84 个 TLP 的信用。这意味着并发写操作不能超过 88 个，并发读操作不能超过 84 个。其次，DMA 读操作需要分配唯一的 PCIe 标签来识别和重排 DMA 响应 [③]。尽管 PCIe 协议和很多内存控制器支持 256 个 PCIe 标签，本文使用的 FPGA 中的 DMA 引擎仅支持 64 个 PCIe 标签，进一步将 DMA 读操作的并发请求数限制为 64 个。这使得 PCIe DMA 读操作的吞吐量仅能达到 60 Mops（百万次操作每秒），如图 5.5a 所示。另一方面，对于 40 Gbps 网络和 64 字节键值对，如果客户端批量发送这些键值对，网络吞吐量的上限为 78 Mops。本文希望用 GET（读）操作使网络吞吐量饱和。因此，网卡上的键值存储必须充分利用 PCIe 带宽，即每个 GET 操作的平均内存访问次数需要接近 1。这归结为三个挑战：

最小化每键值操作的 DMA 请求。哈希表和内存分配器是键值存储中需要随机内存访问的两个主要组件。以前的工作建议使用布谷鸟（Cuckoo）哈希 [70,210]，即使在高负载因子 [④] 下，每个 GET 操作的内存访问次数也接近 1。但是，布谷鸟哈希是为读操作优化的。在高于 50% 的负载因子下，布谷鸟哈希的每个 PUT 操作平均需要多次存储器访问，且方差很大。这不仅会消耗 PCIe 吞吐量，还会导致写入密集型工作负载的延迟不稳定。

除了哈希表查找之外，还需要动态内存分配来存储无法在哈希表中内联 [⑤] 的可变长度键值。为了在写入密集型小键值工作负载下匹配 PCIe 和网络吞吐量（即将两者的吞吐量同时几乎全部利用），需要使哈希表查找和内存分配的内存访问次数尽可能少。

隐藏 PCIe 延迟，同时保持一致性。一致性（consistency）是分布式事务中的术语，表示并发执行的事务之间逻辑上相互隔离的性质。可编程网卡内的不同硬件模块并行处理，组成一个分布式系统。由于一个键值操作需要多次内存读写访问，多个键值操作并发处理时，每个键值操作可被视为一个分布式事务。本文实现了严格可线性化（strict serializability），即多个键值操作并发执行的结果与

[①] PCIe DMA 操作分为有响应（non-posted）和无响应（posted）两种。有响应操作意味着双向通信，需要接收端返回数据，例如读操作。无响应操作意味着单向通信，例如写操作，接收端写入完成后不再通知发送端。PCIe 流控机制中的有响应和无响应操作是由独立的信用分别控制的。

[②] PCIe 基于信用的流控分为基于数据包和基于有效载荷的两种，数据包的个数和有效载荷的字节数都满足流控要求时才能发送。本文的键值操作访问内存的粒度较小，因此有效载荷字节数不是瓶颈，只需考虑数据包的个数。

[③] 由于 PCIe 网络的传播延迟和 PCIe 终端的处理延迟是可变的，不能保证先发送的操作一定先返回响应。为了配对 DMA 请求和响应，PCIe 终端发送读操作时携带标签，接收端返回的数据也携带相应的标签。显然，并发的读操作必须具有唯一的标签。

[④] 负载因子（load factor）指哈希表中已经存储数据的哈希槽在所有哈希槽中的比例。

[⑤] 内联（inline）是指键值对很小，可以直接放在哈希表内固定大小的槽位中，而无需分配动态内存专门存储。

它们按照网络输入顺序一个接一个执行的结果相同。严格可线性化是分布式事务最强的一致性标准。

在本文的 FPGA 平台上，一个 PCIe DMA 操作的延迟大约是 1 μs。可编程网卡处理键值请求时，如果在等待 DMA 读操作返回的过程中不做任何其他事情，那么键值请求的吞吐量将只有约 1 Mops，这显然不能接受。因此，可编程网卡上的高性能键值存储必须并发执行键值操作和 DMA 请求，以隐藏 PCIe 延迟。但是，键值操作可能具有依赖性，不是所有的键值操作都能并发执行。例如，在同一个键上的 PUT 操作之后的 GET 操作需要返回更新后的值。再如，两个相邻的原子增加操作需要在执行第二个之前等待第一个完成。这需要跟踪正在处理的键值操作，并在发生数据冒险（data hazard）时暂停流水线，或者更好地设计乱序执行器以解决数据依赖性而无需显式暂停流水线。

在网卡 DRAM 和主机内存之间分配负载。一个显而易见的想法是使用网卡上的 DRAM 作为主机内存的缓存，但在网卡中，DRAM 吞吐量（12.8 GB/s）与两个 PCIe Gen3 x8 接口的可实现吞吐量（13.2 GB/s）相当。本文期望在 DRAM 和主机存储器之间分配存储器访问以便利用它们的两个带宽。然而，与主机存储器（64 GiB）相比，板载 DRAM 很小（4 GiB），因此需要混合缓存和负载调度方法。

下文将介绍 KV-Direct，一种基于 FPGA 的新型键值存储系统，满足了上述所有设计目标。

5.3　KV-Direct 操作原语

KV-Direct 将远程直接内存访问（Remote Direct Memory Access，RDMA）原语扩展到*远程直接键值访问*原语，如表 5.2 所述。客户端将 *KV-Direct* 操作发送到键值存储服务器，而可编程网卡处理请求并发送回结果，绕过 CPU。键值存储服务器上的可编程网卡是一个重配置为*键值处理器*的 FPGA。

除了表 5.2 顶部所示的标准键值存储操作外，KV-Direct 还支持两种类型的向量运算：将标量发送到服务器上的网卡，网卡将更新应用于向量中的每个元素；或者向服务器发送一个向量，并且网卡逐个元素地更新原始向量。此外，KV-Direct 支持在原子操作中包含用户定义的函数。用户定义函数需要预先注册并编译为硬件逻辑，运行时使用函数 ID 来索引。使用用户定义函数的键值操作类似于*动态消息*（active message）[196]，从而节省了将键值取回客户端处理的通信和同步成本。

当对一个键执行向量操作更新（update），归约（reduce）或过滤（filter）时，其值被视为固定位宽元素的数组。每个函数 λ 对向量中的一个元素、客户端指定

表 5.2　KV-Direct 操作。

get $(k) \to v$	获取键 k 的值。
put $(k, v) \to$ bool	插入或替换 (k, v) 对。
delete $(k) \to$ bool	删除键 k。
update_scalar2scalar $(k, \Delta, \lambda(v, \Delta) \to v) \to v$	原子更新键 k，使用函数 λ，作用于 Δ 上，返回原始值。
update_scalar2vector $(k, \Delta, \lambda(v, \Delta) \to v) \to [v]$	原子更新键 k 中的所有元素，使用函数 λ 和标量 Δ，返回原始向量。
update_vector2vector $(k, [\Delta], \lambda(v, \Delta) \to v) \to [v]$	原子更新键 k 中的所有元素，使用函数 λ，基于向量 $[\Delta]$ 中的对应元素，并返回原始向量。
reduce $(k, \Sigma, \lambda(v, \Sigma) \to \Sigma) \to \Sigma$	把向量 k 归约成一个标量，使用函数 λ，并返回归约结果 Σ。
filter $(k, \lambda(v) \to$ bool$) \to [v]$	在向量 k 中筛选元素，使用函数 λ，并返回筛选后的向量。

的参数 Δ 和/或初始值 Σ 进行归约操作。基于第 4 章的 KV-Direct 开发工具链将用户定义函数 λ 复制多份，以利用 FPGA 中的并行性，并将计算吞吐量与键值处理器中其他组件的吞吐量相匹配，然后使用高层次综合（HLS）工具[57-58] 将其编译为可重新配置的硬件逻辑。得益于第 4 章的设计，开发工具链自动提取复制函数中的数据依赖性，并生成完全流水线的可编程逻辑。在键值存储客户端开始运行之前，键值存储服务器上的可编程网卡应加载用户定义函数 λ 的硬件逻辑。

利用用户定义函数，可以在向量操作中实现常见的流处理。例如，网络处理应用程序可以将该向量解释为用于网络功能的数据包流（packet stream），或用于数据包事务（packet transaction）的一组状态[211]。甚至可能实现完全在可编程网卡中的单对象事务处理，例如，TPC-C 基准测试中 S_QUANTITY 达到阈值后归零的操作[212]。向量归约操作可以支持 PageRank[69] 中累加邻居节点权重的计算。还可以使用向量过滤操作来获取稀疏向量中的非零值。

5.4　键值处理器

如图 5.6 所示，键值处理器从板载网卡接收数据包，解码向量操作并将键值操作缓冲在保留站 ① 中（第 5.4.3 节）。接下来，乱序执行引擎（第 5.4.3 节）从保留站发送可并发执行的键值操作到键值操作译码器。根据操作类型，键值处理

① 保留站（reservation station）是计算机体系结构中的概念，存储待执行的操作，并调度合适的操作并发执行。

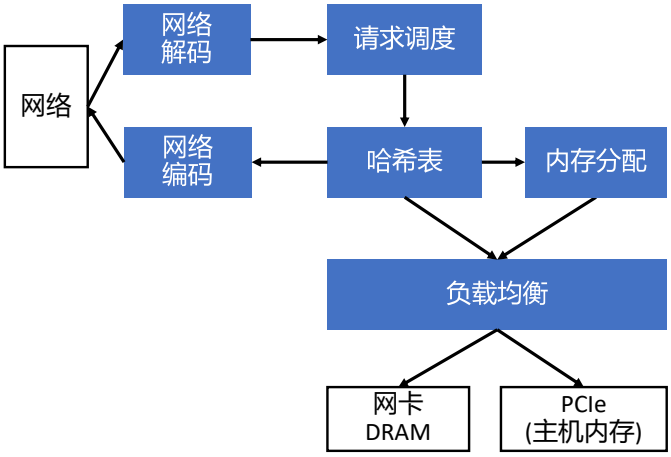


图 5.6　键值处理器架构。

器查找哈希表（第 5.4.1 节）并执行相应的操作。为了最小化内存访问次数，较小的键值对在哈希表中内联（inline）存储，其他的键值对存储在 slab 内存分配器（第 5.4.2 节）的动态分配内存中。哈希索引和 slab 分配的内存都由统一的内存访问引擎（第 5.4.4 节）管理，它通过 PCIe DMA 访问主机内存并将部分主机内存缓存在板载 DRAM 中。键值操作完成后，结果被发送回乱序执行引擎（第 5.4.3 节），在保留站中寻找依赖它的键值操作并执行之。

正如第 5.2.4 节所讨论的，PCIe 吞吐量的稀缺性要求键值处理器节约 DMA 访问。对于 GET 操作，至少需要读取一次内存。对于 PUT 或 DELETE 操作，对于哈希表数据结构，至少需要一次读取和一次写入[①]。基于日志的数据结构可以实现每个 PUT 平摊下来少于一次的写入操作，但它牺牲了 GET 性能。KV-Direct 仔细设计哈希表，以便在每次查找和插入时实现接近理想的 DMA 访问。KV-Direct 也仔细设计了内存分配器，使每次动态内存分配平摊下来，只需不到 0.1 次 DMA 操作。

5.4.1　哈希表

为了存储可变大小的键值，键值存储分为两部分。第一部分是哈希索引（图 5.7），它包含固定数量的*哈希桶*。每个哈希桶包含几个*哈希槽*和一些元数据。内存的其余部分是动态分配的，由 slab 分配器（第 5.4.2 节）管理。初始化时配置的*哈希索引比率*（hash index ratio）是哈希索引的大小占键值存储总内存大小的比例。哈希索引比率的选择将在第 5.4.1 节讨论。

每个哈希槽包括指向动态分配的存储器中的键值数据的指针和*辅助哈希*。辅助哈希是一种优化，采用与主哈希函数独立的另一个哈希函数。由于每个哈希桶中有多个哈希槽，键值处理器需要判断哪个哈希槽对应待查找的键。通过 9 位的

[①]读取操作取出哈希槽内的键，如果槽位为空或与待查找的键相同，且不需要重新分配内存空间，则需要一次写入操作来写回数据

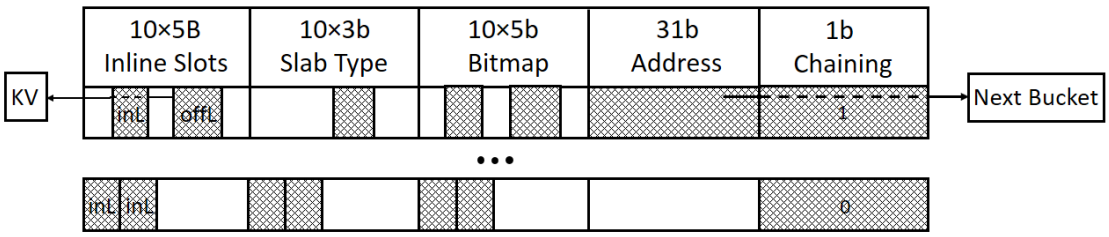


图 5.7　哈希索引结构。每行是一个哈希桶，包含 10 个哈希槽，每个哈希槽包括 3 位的 slab 内存类型，一个位图标记内联键值对的开始和结束，以及指向哈希冲突时下一个链接桶的指针。

辅助哈希，可以 511/512 的概率确定哪个是待查找的键。但为了确保正确性，仍然需要一次额外的内存访问，将键取出，逐字节比较。假设主机内存中的 64 GiB 键值存储和 32 字节分配粒度[①]，指针需要 31 位。每个哈希槽的大小是 5 字节[②]。为了确定哈希桶大小，需要在每个桶的哈希槽数和 DMA 吞吐量之间进行权衡。图 5.5a 表明，低于 64B 粒度时，DMA 读取吞吐量受 DMA 引擎中 PCIe 延迟和并行性的约束。小于 64B 的桶大小将增加哈希冲突的可能性。另一方面，将桶大小增加到 64B 以上会降低哈希查找的吞吐量。因此桶大小选择为 64 字节。

*键值大小*是指键和值的总大小。小于阈值大小的键值在哈希索引中内联（inline）存储，以节约对获取键值数据的额外存储器访问。内联键值可以跨越多个哈希槽，其指针和辅助哈希字段被重新用于存储键值数据。内联所有可装入哈希索引的键值可能不是最佳选择。一个内联的键值可能占用多个哈希槽，减小了哈希表可存储的键值数量。在哈希表容量允许的情况下，内联键值可以减少平均内存访问次数。为此，KV-Direct 根据哈希表被填满的比例来选择*内联阈值*，并内联小于该阈值的键值。传统上，哈希表采用负载因子（load factor）[③]来衡量哈希表被填满的比例，然而这忽略了哈希表的元数据（metadata）和内部碎片（internal fragmentation）带来的开销。为了更科学地比较哈希表不同参数的选择，本章使用*内存利用率*（memory utilization）[④]。如图 5.8 所示，对于某个内联阈值，由于更多的哈希冲突，每个键值操作的平均内存访问次数随内存利用率的增加而增加。较高的内联阈值下，平均内存访问次数的增长曲线更陡峭。因此可以找到最佳内联阈值以最小化在给定内存利用率下的内存访问次数。与哈希索引比率一样，内联阈值也可以在初始化时配置。

当哈希桶中的所有哈希槽都已填满时，有几种解决方案可以解决哈希冲突。

[①]32 字节的分配粒度权衡了内部碎片和用于内存分配的元数据开销。

[②]本文中的设计参数是根据本文所用硬件平台的参数配置的，对于不同容量的内存，哈希槽大小、指针位宽等参数可能改变。

[③]负载因子是已被占用的哈希槽数量与总哈希槽数量之比。

[④]内存利用率是键值存储中所有键值的大小之和与键值存储的总大小之比。由于元数据和内部碎片的存在，内存利用率总是小于 1 的。

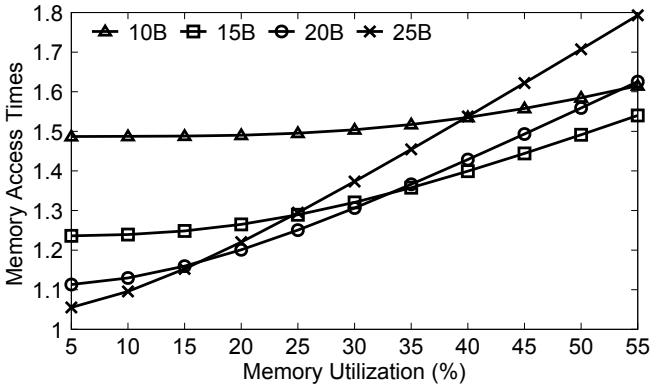


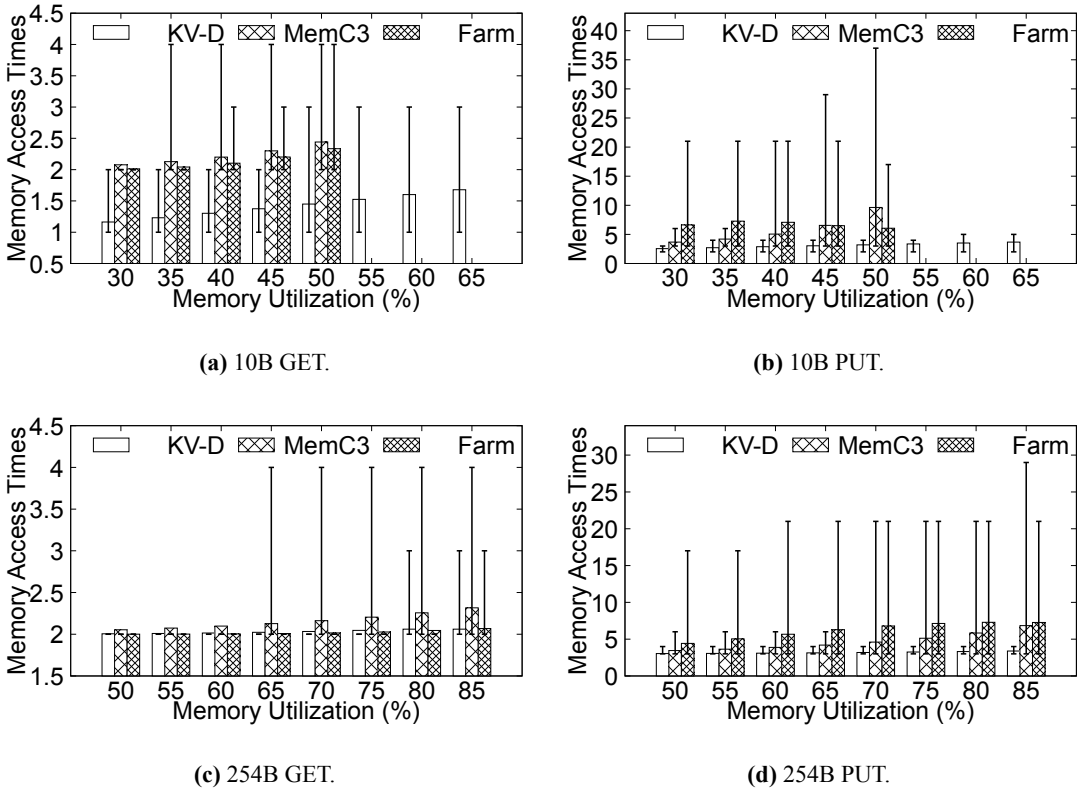
图 5.8　不同内联阈值下的平均内存访问次数和内存利用率。

布谷鸟哈希（Cuckoo Hashing）[213] 和跳房子哈希（Hopscotch Hashing）[214] 通过在插入过程中移动已被占用的哈希槽来保证新插入的键值总是被插入到哈希桶中，从而查找时仅需比较同一哈希桶中恒定数量的哈希槽，实现恒定时间查找。但是，在写入密集型工作负载中，高负载率下的内存访问时间会有较大的波动。在极端情况下，甚至可能出现插入失败，需要哈希表扩容的情况。另一种解决哈希冲突的方法是 *线性探测法*，它可能受到群集（clustering）的影响，因此其性能对哈希函数的均匀性敏感。为此，本文选择 *拉链法*来解决哈希冲突，这平衡了查找和插入的性能，同时对哈希群集更加健壮。

为了比较 KV-Direct 的拉链法、MemC3 中的桶式布谷鸟哈希（bucketized Cuckoo Hash）[33] 和 FaRM[70] 中的链式关联（chain associative）跳房子哈希（Hopscotch Hash），图 5.9 绘制了三种可能的哈希表设计中每个 GET 和 PUT 操作的平均内存访问次数。在 KV-Direct 的实验中，针对给定的键值大小和内存利用率要求，为内联阈值和哈希索引比率做出最佳选择。在布谷鸟和跳房子哈希实验中，假设键是内联的并且可以并行比较，而值存储在动态分配的 slab 存储区中。由于 MemC3 和 FaRM 的哈希表不能对 10B 键值大小支持超过 55％的内存利用率（即它们的元数据和内部碎片占用空间较多），图 5.9a 和图 5.9b 仅显示 KV-Direct 的性能。

对于内联键值，KV-Direct 的每个 GET 操作仅需接近 1 次内存访问，在非极端内存利用率下每个 PUT 也仅需 2 次内存访问。非内联键值的 GET 和 PUT 则有一次额外的内存访问。在高内存利用率下比较 KV-Direct 和链式跳房子哈希，跳房子哈希在 GET 中表现更好，但在 PUT 中表现更差。虽然 KV-Direct 无法保证最坏情况下的 DMA 访问次数，但会在 GET 和 PUT 之间取得平衡。布谷鸟哈希的 GET 操作能保证访问最多两个哈希槽，因此在大多数内存利用率下，KV-Direct 的内存访问次数更多。但是，在高内存利用率下，布谷鸟哈希会导致 PUT 操作的内存访问次数出现大幅波动。

哈希表设计中有两个自由参数：（1）内联阈值，（2）整个内存空间中哈希索

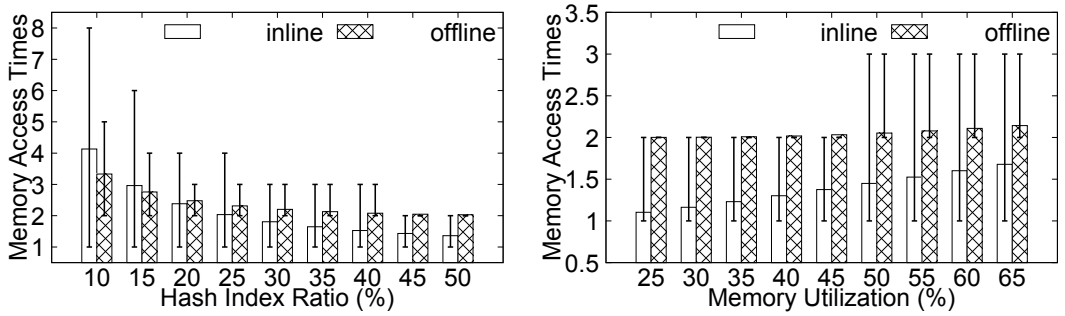


(a) 10B GET.　　**(b)** 10B PUT.

(c) 254B GET.　　**(d)** 254B PUT.

图 5.9　每个键值操作的内存访问次数。

引的比率。如图 5.10a 所示，当哈希索引比率增长时，可以内联存储更多的键值对，从而产生更低的平均内存访问次数。图 5.10b 显示了随着使用更多内存而增加的内存访问次数。



(a) 固定内存利用率 0.5。　　**(b)** 固定哈希索引率 0.5。

图 5.10　不同内存利用率或哈希索引率下的内存占用。

如图 5.11 所示，最大内存利用率在哈希索引比率较高时下降，因为可用于动态分配的内存较少。因此，为了在给定的内存空间中容纳所有需存储的键值，哈希索引比率具有上限。本节选择此上限以获得最小的平均内存访问次数，如图 5.11 中的虚线所示，首先根据目标内存利用率，得到有大的哈希索引比率；然后根据哈希索引比率可以得到理论上 GET 操作查找索引所需的平均内存访问次

数。

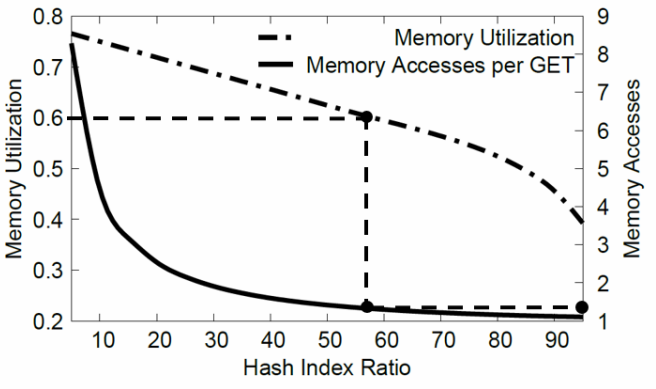


图 5.11　如何在给定内存利用率需求和键值大小的情况下决定最优哈希索引率。

5.4.2　Slab 内存分配器

链式哈希槽和非内联键值需要动态内存分配。为此，本章选择 slab 内存分配器[215] 来实现每次内存分配和释放的 $O(1)$ 平均内存访问次数。主 slab 分配器逻辑在主机 CPU 上运行，并通过 PCIe 与键值处理器通信。Slab 分配器将分配大小四舍五入到最接近的 2 的幂，称为 *slab 大小*。它为每个可能的 slab 大小（32，64，…，512 字节）和全局 *分配位图*（allocation bitmap）维护*空闲 slab 池*，以帮助将小的空闲 slab 合并回更大的 slab。每个空闲 slab 池是一个 *slab 条目*数组，由一个地址字段和一个 slab 类型字段组成。slab 类型字段表示 slab 条目的大小。

可以在网卡上缓存可用的 slab 池，并与主机内存同步。通过批处理的 PCIe DMA 同步操作，每次分配或释放内存平摊下来只需少于 0.07 次 DMA 操作。当一个空闲 slab 池几乎是空的时，需要拆分较大的 slab。因为 slab 类型已经包含在 slab 条目中，所以在 *slab 分割*时，slab 条目只需从较大的池复制到较小的池，而不需要分割出多个小 slab 条目。

在重新分配时，slab 分配器需要检查释放的 slab 是否可以与其邻居合并，需要至少一次读取和写入分配位图。受垃圾收集的启发，主机上的 *懒惰 slab 合并*在一个 slab 池几乎为空，并且没有更大的 slab 池有足够的 slab 来拆分时，批量合并空闲 slab。

如图 5.12 所示，对于每个 slab 大小，网卡上的 slab 缓存使用两个双端堆栈（double-end stack）与主机 DRAM 同步。网卡端的双端堆栈（图 5.12 中的左侧）左端由分配器出栈、解除分配器入栈，右端通过 DMA 对应的 主机端双端堆栈与左端同步。网卡监视网卡堆栈的大小，并根据高水位线和低水位线与主机堆栈同步。主机守护进程定期检查主机端双端堆栈的大小。如果高于高水位线，则触发 slab 合并；低于低水位线时，会触发 slab 分裂。因为双端堆栈的每一端都是

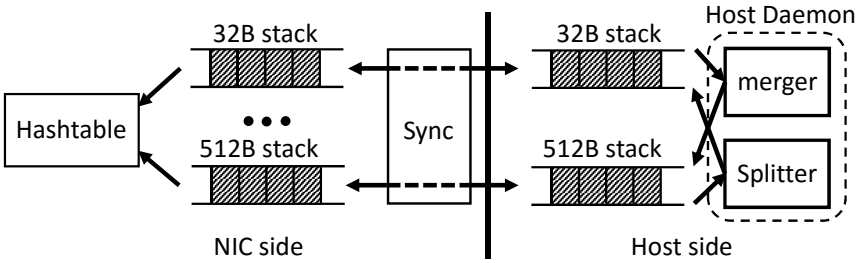


图 5.12　Slab 内存分配器。

仅由网卡或主机之一独占访问的，并且在移动指针之前先搬移数据，所以只要双端堆栈中的数据量大于保护阈值，就不会发生竞争条件。

Slab 内存分配器的通信开销来自网卡访问主机内存中的可用 slab 队列。在本章中，每个 slab 条目是 5 个字节，DMA 粒度是 64 个字节，因此每个 slab 操作的平摊 DMA 开销是 5/64 次 DMA 操作。此外，网卡上新释放的 slab 槽往往可被网卡后续的分配操作重新使用，因此很多情况下根本不需要 DMA 操作。为了维持每秒 180M 操作的最大吞吐量，在最坏的情况下，需要传输 180M 个 slab 条目，消耗 720 MB/s PCIe 吞吐量，即网卡的总 PCIe 吞吐量的 5%。

Slab 内存分配器的计算开销来自主机 CPU 上的 slab 拆分和合并。幸运的是，它们并不经常被调用。对于具有稳定键值大小分布的工作负载，新释放的 slab 槽由后续分配重用，因此不会触发拆分和合并。

Slab 拆分需要将连续的 slab 条目从一个 slab 队列移动到另一个 slab 队列。当工作负载从大键值转换到小键值时，在最坏的情况下，CPU 需要每秒移动 90M 个 slab 条目，这只占核心的 10%，因为它只是连续的内存复制。

将可用 slab 条目合并到较大的 slab 条目是相当耗时的任务，因为这个垃圾回收过程需要用 slab 条目的地址填充分配位图，因此需要随机存储器访问。要对可用 slab 条目的地址进行排序并合并连续的 slab，基数排序[216] 比简单的位图具有更好的多核可扩放性。如图 5.13 所示，将 16 GiB 向量中的所有 40 亿个空闲 slab 槽位合并，在一个 CPU 核上需要 30 秒，而在 32 个核上使用基数排序仅需 1.8 秒[216]。虽然空闲 slab 槽位的垃圾收集需要几秒钟，但它在后台运行而不会停止 slab 分配器，并且实际上仅在工作负载从小键值转换到大键值时触发。

5.4.3　乱序执行引擎

在键值处理器中，相同键的两个键值操作之间的依赖性将导致数据危险（data hazard）和流水线暂停（pipeline stall）。在单键原子操作（single-key atomics）中，这个问题更为显著，因为其中所有操作都是有依赖的，必须逐个处理。这限制了原子操作的吞吐量。本节借用计算机体系结构领域中*动态调度*的概念，并实现*保留站*（reservation station）来跟踪所有正在进行的键值操作及其*执行上下*

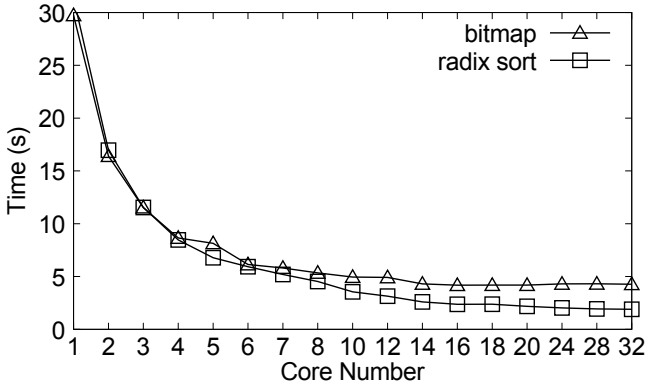


图 5.13　合并 40 亿个 slab 槽位的时间开销。

文。

为了饱和利用 PCIe、DRAM 带宽和处理流水线，需要多达 256 个并发执行的键值操作。但是，并行比较 256 个 16 字节键将占用 FPGA 的 40% 逻辑资源。为了避免并行比较，本节将键值操作存储在片上 BRAM 中的小哈希表中，由键的哈希索引。相同哈希值的键值操作被视为具有依赖关系。不同的键可能具有相同的哈希值，因此可能存在误报的依赖关系，但它永远不会错过依赖关系。具有相同哈希的键值操作组织成链表结构，由键值处理器顺序处理。哈希冲突会增加误报的依赖关系，降低键值处理效率，因此保留站包含 1024 个哈希槽，使哈希冲突可能性低于 25%。

保留站不仅保存因依赖关系而暂时挂起的操作，还缓存了最近被访问的键值以便进行 *数据转发*（data forwarding）。当主处理流水线完成键值操作时，其结果返回给客户端，最新值被转发到保留站。保留站逐个检查相同哈希槽中的待定操作，立即执行具有匹配键的操作并从保留站移除。对于原子操作，计算在专用执行引擎中执行。对于写入操作，将更新缓存的值。执行结果直接返回给客户端。在扫描完成依赖关系链表后，如果保留站缓存的值被更新了，则向主处理流水线发出 PUT 操作，以将高速缓存写回到主存。这种数据转发和快速执行路径使单键原子操作能够在每个时钟周期内处理一次操作[①]，消除了频繁被访问的键的线头阻塞（head-of-line blocking）。保留站确保了数据一致性，因为在同一个键上没有两个操作可以在主处理流水线中同时进行。图 5.14 描述了乱序执行引擎的结构。

下面评估乱序执行的有效性。使用的工作负载包括单键原子操作和长尾分布。对比方法是遇到键冲突就暂停流水线的简单方法。使用单边 RDMA 和双边 RDMA[47] 吞吐量作为基线。

如果没有乱序执行引擎，原子操作需要等待网卡中的 PCIe 延迟和处理延迟，

[①]本章中 FPGA 键值处理器的时钟频率是 180 MHz，因此吞吐量可达 180 M op/s。

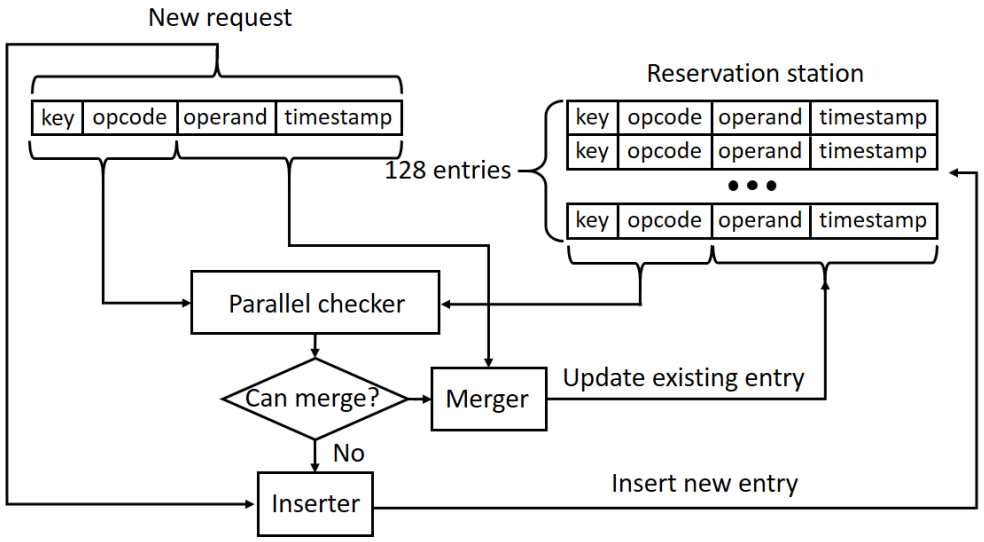
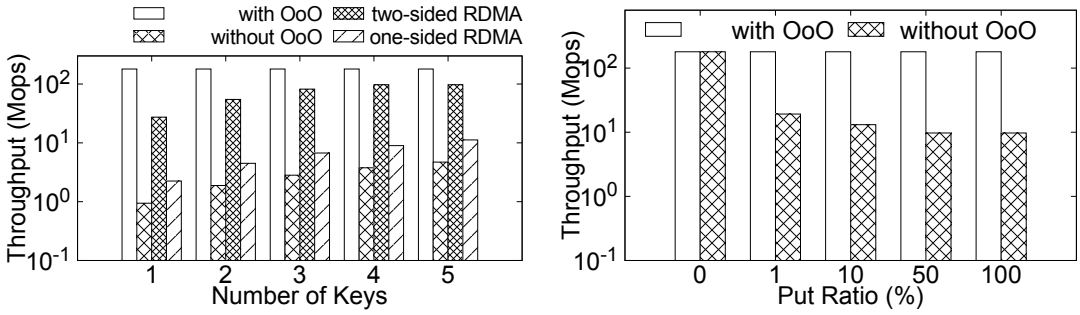


图 5.14　乱序执行引擎。



(a) 原子操作。　　**(b)** 长尾负载。

图 5.15　乱序执行引擎的效率。

在此期间无法执行对同一键的后续原子操作。如图 5.15a，暂停流水线方法的单键原子操作吞吐量为 0.94 Mops，与使用商用 RDMA 网卡测量的 2.24 Mops 接近[47]。商用 RDMA 网卡的更高吞吐量可归因于其更高的时钟频率和更低的处理延迟。利用乱序执行，KV-Direct 的单键原子操作可达峰值吞吐量，即每个时钟周期处理一次键值操作。在 MICA[30] 中，单键原子吞吐量受限于单个 CPU 核的处理能力，无法随多核扩放。事实上，原子增加（atomic increment）操作的性能可以随多核扩放[47]，但它依赖于原子操作之间的可交换性，因此不适用于不可交换的原子操作，如比较-交换（compare and swap）。

利用乱序执行，单键原子吞吐量提高了 191 倍，达到了 180 Mops 的时钟频率极限。当原子操作在多个键之间均匀分布时，单边 RDMA、双边 RDMA 和没有乱序执行的 KV-Direct 吞吐量随着键的数量线性增长，但仍然与 KV-Direct 使用乱序执行后的最佳吞吐量有很大差距。

图 5.15b 显示了长尾工作负载下的吞吐量。当 PUT 操作发现流水线中有任

何正在进行的键相同的操作时，流水线会暂停。长尾工作负载具有多个访问非常频繁的键，因此具有相同键的两个操作很可能几乎同时到达。当 PUT 操作占所有操作的比例更高时，具有相同键的两个操作中更有可能至少一个是 PUT 操作，从而触发流水线暂停。

5.4.4　DRAM 负载分配器

为了进一步减轻 PCIe 的负担，本节在 PCIe 和网卡板载 DRAM 之间调度内存访问。网卡 DRAM 具有 4 GiB 容量和 12.8 GB/s 的吞吐量，比主机 DRAM（64 GiB）上的键值存储存储小一个数量级，并且比 PCIe 链路（14 GB / s）稍慢。一种方法是将固定部分的键值存储放入网卡 DRAM 中。但是，网卡 DRAM 太小，只能容纳整个键值存储的一小部分。另一种方法是使用网卡 DRAM 作为主机内存的缓存，但由于网卡 DRAM 的吞吐量有限（甚至比 PCIe 的吞吐量更低），吞吐量甚至会降低。

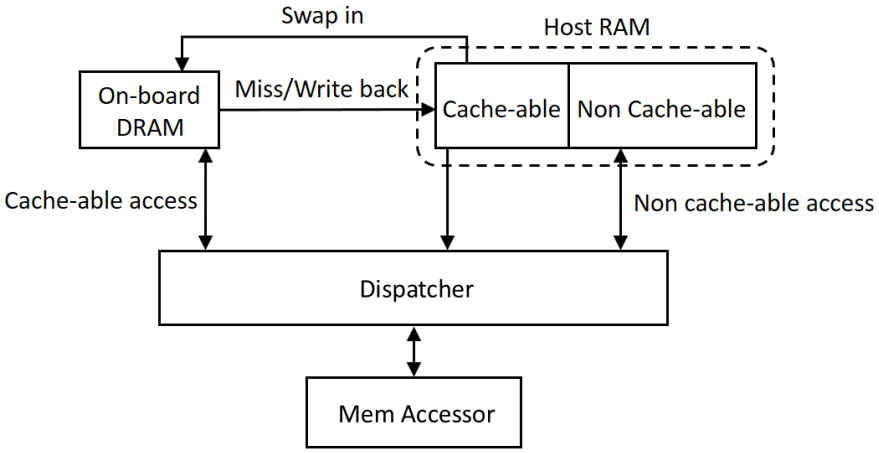


图 5.16　DRAM 负载分配器。

本节采用混合解决方案，将 DRAM 用作主机内存中固定部分键值存储的缓存，如图 5.16 所示。可缓存部分由存储器地址的哈希确定，哈希的粒度为 64 字节（DRAM 访存粒度）。选择哈希函数，使得哈希索引和动态分配的存储器中的地址具有相同的缓存概率。整个键值存储内存中可缓存内存所占的部分称为*负载分配比例*（l）。如果负载分配比例 l 增大，更大比例的负载将被分配给板载 DRAM，并且缓存命中率 $h(l)$ 将增加。为了平衡 PCIe 和板载 DRAM 上的负载，应优化负载调度比 l，使得：

$$\frac{l}{tput_{DRAM}} = \frac{(1-l)+l\cdot(1-h(l))}{tput_{PCIe}}$$

特别的，在均匀（uniform）负载下，令 k 是板上 DRAM 大小和主机键值存储大小之比，则缓存命中率 $h(l) = \frac{\text{cache size}}{\text{cache-able memory size}} = \frac{k}{l}$。当 $k \leqslant l$ 时，均匀负载下的缓存并不高效。在长尾负载（Zipf 分布）下，设 n 是键值的总数，则大致上

$h(l)=\frac{\log(\text{缓存大小})}{\log(\text{可缓存部分大小})}=\frac{\log(kn)}{\log(ln)}$，当 $k \leqslant l$ 时。在长尾工作负载下，1G 的键值存储中 1M 缓存的缓存命中概率高达 0.7。最优的 l 可以得出数值解，这将在第 5.7.1 节讨论。

一个技术挑战是在 DRAM 高速缓存中存储元数据。每 64 字节的一个缓存行（cache line），需要 4 个地址位和一个脏标志位的元数据。因为所有键值存储都由网卡专门访问，不需要缓存有效位。为了存储每个高速缓存行的 5 个元数据位，如果将高速缓存行扩展到 65 个字节，会由于非对齐访问而降低 DRAM 性能；如果将元数据保存在别处，将使内存访问次数倍增。相反，本文利用 ECC（纠错编码）DRAM 中的备用位来进行元数据存储。ECC DRAM 通常每 64 位数据具有 8 个 ECC 位。事实上，使用汉明码来纠正 64 位数据中的一位错误，只需要 7 个额外的校验位。第 8 个 ECC 位是用于检测双位错误的奇偶校验位。当以 64 字节粒度和对齐方式访问 DRAM 时，每 64B 数据有 8 个奇偶校验位。本文将奇偶校验的检查粒度从 64 个数据位增加到 256 个数据位，因此仍然可以检测到双位错误。这节约出了 6 个额外的位，可以用于保存地址位和脏标志这些元数据。

图 5.17 显示了 DRAM 负载调度吞吐量相比仅使用 PCIe 的改进。在均匀工作负载下，DRAM 的缓存效果可以忽略不计，因为它的大小仅为主机键值存储内存的 6%。在长尾工作负载下，大约 30% 的内存访问由 DRAM 缓存提供。总的来说，在 95% 和 100% GET 情况下，总吞吐量达到了 180 Mops 的时钟频率限制。但是，如果简单地将 DRAM 用作高速缓存，则吞吐量将反而降低，因为 DRAM 吞吐量低于 PCIe 吞吐量。

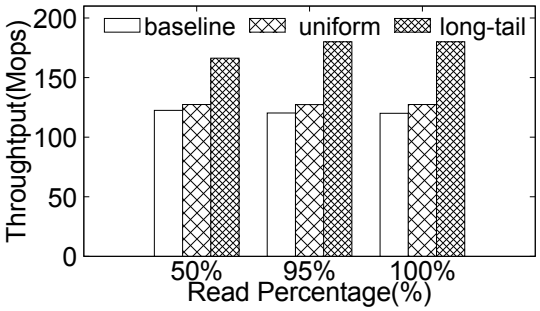


图 5.17　负载分派下的 DMA 吞吐量（固定负载分派比例为 0.5）。

5.4.5　向量操作译码器

整个键值处理器设计将批处理视为通用原则。这包括在一个存储桶中批量获取多个哈希槽，空闲 slab 队列与主机内存的批量同步，懒惰 slab 拆分和合并，以及保留站对有依赖键值操作沿链表批量处理。批处理通过将控制平面开销分摊到多个数据平面的有效负载来提高性能。

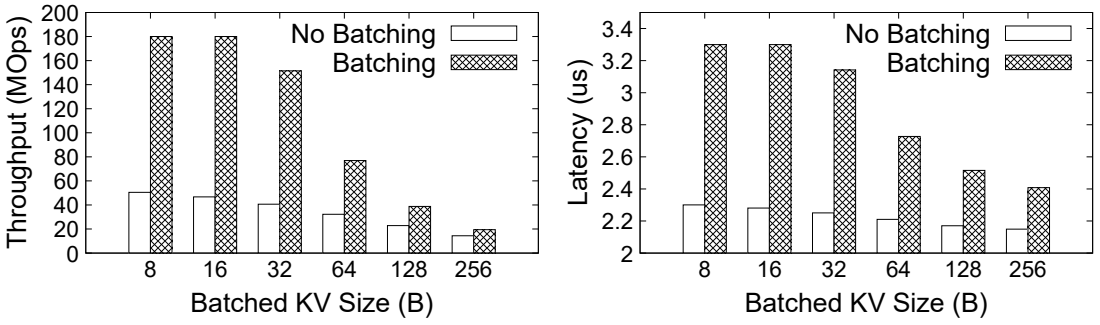
与 PCIe 相比，网络是一种更加稀缺的资源，具有更低的带宽（5 GB / s）和更高的延迟（2 μs）。以太网上的 RDMA 写数据包有 88 字节的包头（header）和填充（padding）开销，而 PCIe TLP 数据包只有 26 字节的开销。这就是以前基于 FPGA 的键值存储[217-218] 没有使 PCIe 带宽饱和的原因，尽管它们的哈希表设计效率低于 KV-Direct。为了充分利用网络带宽，需要在两个方面进行 *客户端批处理*：在一个数据包中批量处理多个键值操作，并支持向量操作以实现更紧凑的表示。为此，在键值引擎中实现了一个解码器，从单个 RDMA 数据包中解压缩多个键值操作。观察到许多键值具有相同的大小或重复值，键值格式包括两个标志位以允许复制键和值大小，或者包中先前键值的值。幸运的是，许多重要的工作负载（*例如图遍历，参数服务器*）的键值操作是可以批量化的。展望未来，如果可以使用更高带宽的网络，则无需批量处理。

为了评估 KV-Direct 中向量操作的效率，表 5.3 将原子向量增加（vector increment）的吞吐量与两种替代方法进行比较：（1）如果每个元素都存储为一个不同的键，则瓶颈是传输键值操作的网络。（2）如果整个向量存储为一个大的不透明值，由客户端取回处理，则通过网络发送向量的开销也很高。此外，表 5.3 中的两个替代方案不能确保多个客户端同时访问时，向量内部的一致性。添加客户端间的同步会产生进一步的开销。

表 5.3　向量操作的吞吐量 (GB/s)。

向量大小 (字节)	64	128	256	512	1024
向量更新（有返回）	11.52	11.52	11.52	11.52	11.52
向量更新（无返回）	4.37	4.53	4.62	4.66	4.68
每个元素一个键	2.09	2.09	2.09	2.09	2.09
取回客户端处理	0.03	0.06	0.12	0.24	0.46

KV-Direct 客户端在网络数据包中打包键值操作，以降低数据包头的开销。图 5.18 表明，网络批处理可将网络吞吐量提高 4 倍，同时保持网络延迟低于 3.5 μs。



(a) 吞吐量。　　**(b)** 延迟。

图 5.18　网络批量化的效率。

5.5　系统性能评估

5.5.1　系统实现

为了提高开发效率，使用英特尔 FPGA SDK for OpenCL [57] 来合成 OpenCL 的硬件逻辑。键值处理器采用 1.1 万行 OpenCL 代码实现，所有内核都完全流水线化，即吞吐量是每个时钟周期一次操作。凭借 180 MHz 的时钟频率，可以在 180 M op/s 下处理键值操作，如果网络，DRAM 或 PCIe 不是瓶颈。

5.5.2　测试床与评估方法

本节在 8 台服务器和 1 台 Arista DCS-7060CX-32S 交换机的测试台上评估 KV-Direct。每台服务器配备两个禁用超线程的 8 核 Xeon E5-2650 v2 CPU，形成两个通过 QPI Link 连接的 NUMA 节点。每个 NUMA 节点都装有 8 个 DIMM 8 GiB 三星 DDR3-1333 ECC RAM，每台服务器上总共有 128 GiB 的主机内存。可编程网卡 [133] 连接到 CPU 0 的 PCIe root complex，其 40 Gbps 以太网端口连接到交换机。可编程网卡在分叉的 Gen3 x16 物理连接器中有两个 PCIe Gen3 x8 链路。经过测试的服务器配备 SuperMicro X9DRG-QF 主板和一个运行 Archlinux 的 120 GB SATA SSD（内核版本 4.11.9-1）。

对于系统基准测试，使用 YCSB 工作负载[219]。对于偏斜（skewed）的 Zipf 工作负载，本文选择偏差（skewness）0.99 并将其称为*长尾*工作负载。

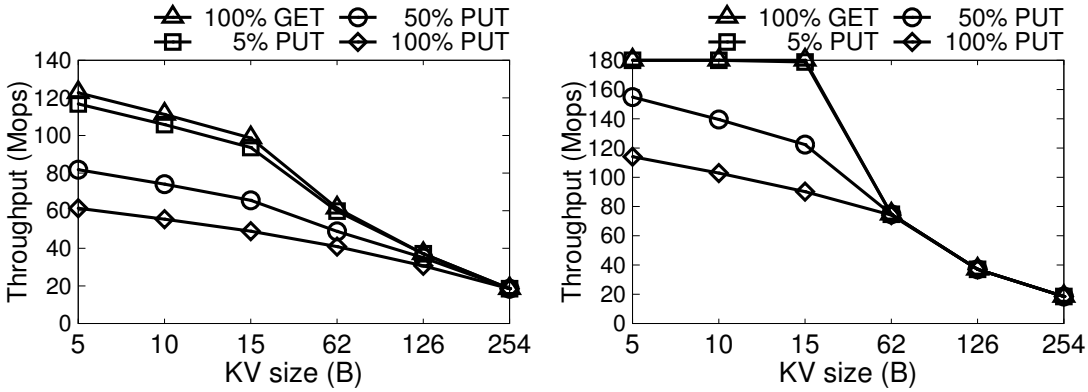
在每个基准测试之前，根据键值大小，访问模式和目标内存利用率来调整哈希索引比率，内联阈值和负载分派比率。然后生成具有给定大小的随机键值对。给定内联键值大小的键大小与 KV-Direct 的性能无关，因为在处理期间将键填充到最长的内联键值大小。为了测试内联案例，使用键值大小作为插槽大小的倍数（当大小为 $\leqslant 50$ 时，即 10 个插槽）。为了测试非内联的情况，使用的键值大小是 2 减 2 字节的幂（对于元数据）。作为准备的最后一步，发出 PUT 操作以将键值对插入空闲键值存储，直到 50% 内存利用率。其他内存利用率下的性能可以从图 5.9中获得。

在基准测试期间，在同一个 ToR 中使用基于 FPGA 的数据包生成器[156] 来生成批量键值操作，将它们发送到键值服务器，接收完成并测量可持续的吞吐量和延迟。分组生成器的处理延迟通过直接环回预先校准，并从延迟测量中移除。误差线表示 5^{th} 和 95^{th} 百分位数。

5.5.3　吞吐量

图 5.19 显示了 YCSB 均匀和长尾（偏斜 Zipf）工作负载下 KV-Direct 的吞吐量。三个因素可能是 KV-Direct 的瓶颈：时钟频率，网络和 PCIe / DRAM。对于

在哈希索引中内联的 5B 至 15B 键值，大多数 GET 需要一个 PCIe / DRAM 访问，而 PUT 需要两个 PCIe / DRAM 访问。这种小型键值在许多系统中很普遍。在 PageRank 中，边的键值大小为 8B。在稀疏逻辑回归中，键值大小通常为 8B-16B。对于分布式系统中的顺控程序和锁，键值大小为 8B。



(a) 均匀分布。　　**(b)** 长尾分布。

图 5.19　KV-Direct 在 YCSB 负载下的吞吐量。

在相同的内存利用率下，由于哈希冲突的概率较高，较大的内联键值具有较低的吞吐量。62B 和更大的键值没有内联，因此它们需要额外的内存访问。长尾工作负载比统一工作负载具有更高的吞吐量，并且能够在读密集工作负载下达到 180 Mops 的时钟频率范围，或达到 ⩾ 62B 键值大小的网络吞吐量。在长尾工作负载下，无序执行引擎在最流行的键上合并到大约 15％的操作，并且板载 DRAM 在 60％负载分配比率下具有大约 60％的高速缓存命中率，这可以统一导致高达 2 倍的吞吐量作为统一的工作量。如表 5.4 所示，KV-Direct 网卡的吞吐量与具有数十个 CPU 核心的最先进的键值存储服务器相当。

5.5.4　能耗效率

插入 KV-Direct 网卡可为空闲服务器增加 10.6 W 的功率。当 KV-Direct 服务器处于峰值吞吐量时，系统功率为 121.4 瓦（在墙上测量）。与表 5.4 中最先进的键值存储系统相比，KV-Direct 的功率效率是其他系统的 3 倍，是第一个达到 100 万的通用键值存储系统商用服务器上每瓦特的键值操作。

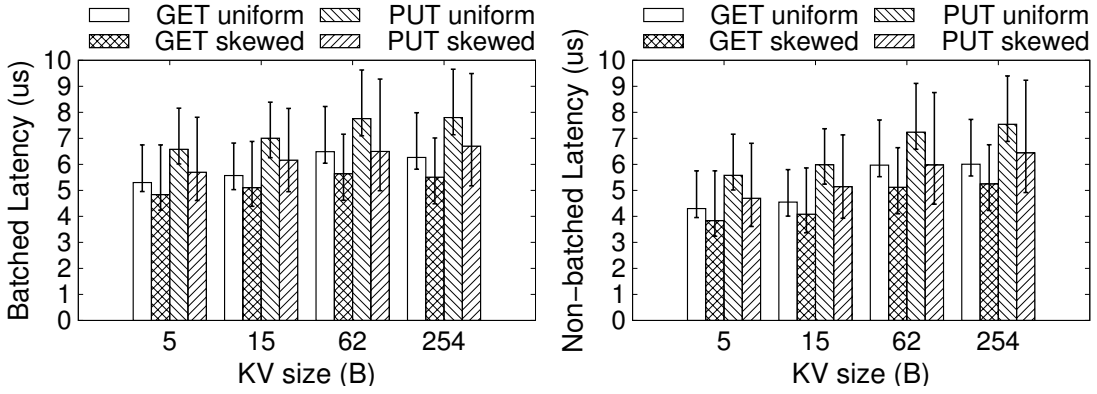
当拔出 KV-Direct 网卡时，空闲服务器的功耗为 87.0 瓦，因此可编程网卡，PCIe，主机内存和 CPU 上的守护进程的总功耗仅为 34 瓦。测量的功率差异是合理的，因为 CPU 几乎处于空闲状态，服务器可以在 KV-Direct 运行时运行其他工作负载（对单侧 RDMA 使用相同的标准，如表 5.4 的括号中所示）。在这方面，KV-Direct 的功率效率是基于 CPU 的系统的 10 倍。

表 5.4 KV-Direct 与其他键值存储系统在长尾（倾斜 Zipf）负载和 10 字节小键下的比较。对于相关工作未报告的性能数字，本文使用相似的硬件来模拟这些系统，并报告粗略的测量结果。对于 CPU 绕过的系统，括号内的数字报告峰值负载和空闲情况下的功耗差异。

键值存储 表 4.3	注释	性能瓶颈	吞吐量 (Mops)		功耗效率 (Kops/W)		平均延迟 (μs)	
			GET	PUT	GET	PUT	GET	PUT
Memcached[197]	传统	CPU 核间同步	1.5	1.5	~5	~5	~50	~50
MemC3[33]	传统	操作系统网络协议栈	4.3	4.3	~14	~14	~50	~50
RAMCloud[29]	内核绕过	分派线程	6	1	~20	~3.3	5	14
MICA[30]	内核绕过，24 个核，12 个网卡口	CPU 键值处理	137	135	342	337	81	81
FaRM[70]	单边 RDMA GET	RDMA 网卡	6	3	~30 (261)	~15	4.5	~10
DrTM-KV[72]	单边 RDMA 和 HTM	RDMA 网卡	115.2	14.3	~500 (3972)	~60	3.4	6.3
HERD'16[47]	双边 RDMA, 12 核	PCIe	98.3	~60	~490	~300	5	5
Xilinx'13[217]	FPGA	网络	13	13	106	106	3.5	4.5
Mega-KV[206]	GPU (4 GiB 板上 RAM)	GPU 键值处理	166	80	~330	~160	280	280
KV-Direct (1 网卡)	可编程网卡，两个 Gen3 x8	PCIe & DRAM	180	114	1487 (5454)	942 (3454)	4.3	5.4
KV-Direct (10 网卡)	可编程网卡，每卡一个 Gen3 x8	PCIe & DRAM	1220	610	3417 (4518)	1708 (2259)	4.3	5.4

5.5.5　延迟

图 5.20 显示了 YCSB 工作负载峰值吞吐量下 KV-Direct 的延迟。在没有网络批处理的情况下，尾部延迟范围为 3 至 9 μs，具体取决于键值大小，操作类型和键分配。由于额外的内存访问，PUT 具有比 GET 更高的延迟。由于更有可能在板载 DRAM 中进行缓存，因此倾斜的工作负载具有比均匀更低的延迟。由于额外的网络和 PCIe 传输延迟，较大的键值具有较高的延迟。网络批处理比非批处理操作增加了不到 1 μs 的延迟，但显著提高了吞吐量，已在图 5.18 中进行了评估。



(a) 有批量操作。　　**(b)** 无批量操作。

图 5.20　在 YCSB 工作负载的峰值吞吐量下 KV-Direct 的延迟。

5.5.6　对 CPU 性能的影响

KV-Direct 旨在绕过服务器 CPU，仅使用一部分主机内存用于键值存储。因此，CPU 仍然可以运行其他应用程序。当单个网卡 KV-Direct 处于峰值负载时，测量对服务器上的其他工作负载的影响最小。表 5.5 量化了 KV-Direct 峰值吞吐量的影响。除了 CPU 0 的顺序吞吐量以访问其自己的 NUMA 内存（以粗体标记的行）之外，CPU 内存访问的延迟和吞吐量大多不受影响。这是因为 8 个主机存储器通道可以提供比所有 CPU 核心消耗的更高的随机访问吞吐量，而 CPU 确实可以强调 DRAM 通道的顺序吞吐量。当键值大小的分布相对稳定时，主机守护程序进程的影响是最小的，因为仅当不同板大小的可用槽的数量不平衡时才调用垃圾收集器。

表 5.5　当 KV-Direct 达到峰值吞吐量时，对 CPU 内存访问性能的影响。使用英特尔性能计数器监视器（Intel PCM）V2.11 测量。

KV-Direct 状态 →		空闲	繁忙
随机访问延迟	CPU0-0	82.2 ns	83.5 ns
	CPU0-1	129.3 ns	129.9 ns
	CPU1-0	122.3 ns	122.2 ns
	CPU1-1	84.2 ns	84.3 ns
顺序访问吞吐量	CPU0-0	60.3 GB/s	55.8 GB/s
	CPU0-1	25.7 GB/s	25.6 GB/s
	CPU1-0	25.5 GB/s	25.9 GB/s
	CPU1-1	60.2 GB/s	60.3 GB/s
随机访问吞吐量	32B 读	10.53 GB/s	10.46 GB/s
	64B 读	14.41 GB/s	14.42 GB/s
	32B 写	9.01 GB/s	9.04 GB/s
	64B 写	12.96 GB/s	12.94 GB/s

5.6　扩展

5.6.1　基于 CPU 的分散 - 聚集 DMA

对于 64B DMA 操作，PCIe 具有 29% 的 TLP 报头和填充开销（§5.2.4），并且 DMA 引擎可能没有足够的并行性来使用小 TLP 使 PCIe 带宽延迟积（BDP）饱和。系统中的 PCIe 根（root complex）支持更大的 DMA 操作，最高 256 字节的 TLP 有效负载。在这种情况下，TLP 头和填充开销仅为 9%，并且 DMA 引擎具有足够的并行性（64）以通过 27 个正在进行的 DMA 读取来使 PCIe 链路饱和。要批量处理 PCIe 链路上的 DMA 操作，可以利用 CPU 来执行分散 - 聚集（scatter gather）（图 5.21）。首先，网卡 DMA 将地址发送到主机内存中的请求队列。主机 CPU 轮询请求队列，执行随机内存访问，将数据放入响应队列并将 MMIO 门铃写入网卡。然后，网卡通过 DMA 从响应队列中提取数据。

图 5.22 表明，与 CPU 旁路方法相比，基于 CPU 的分散 - 聚集 DMA 的吞吐量提高了 79%。除了 CPU 开销之外，基于 CPU 的分散 - 聚集的主要缺点是额外的延迟。为了将 MMIO 从 CPU 保存到网卡，每个门铃批量 256 个 DMA 操作，这需要 10 μs 才能完成。使用基于 CPU 的分散 - 聚集网卡访问主机内存的总延迟是大约 20 美元，比直接 DMA 高出近 20 倍。

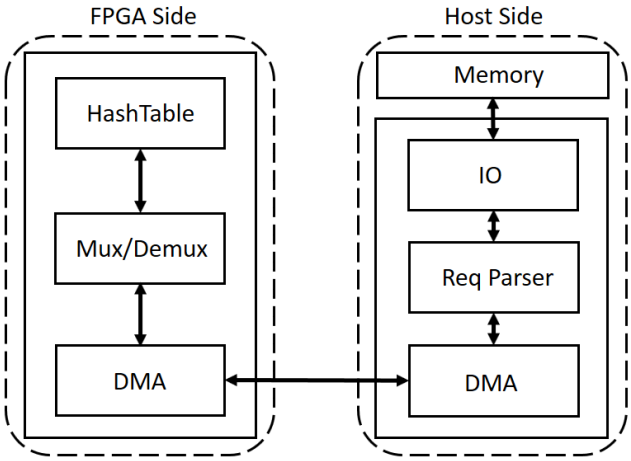
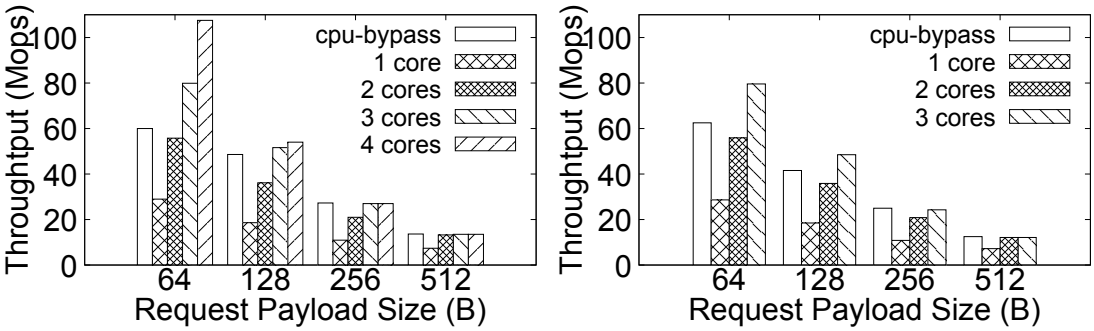


图 5.21　分散 - 聚集（scatter-gather）架构。



(a) 读操作。　　**(b)** 写操作。

图 5.22　分散 - 聚集（scatter-gather）性能。

5.6.2　单机多网卡

KV-Direct 的主要用例是启用远程直接键值访问，而无需服务器上的 CPU 开销。在某些情况下，可能需要构建一个具有每服务器最大吞吐量的专用键值存储。通过模拟，[31] 显示了在具有四个（当前不可用的）60 核 CPU 的单个服务器中实现十亿键值运行的可能性。如表 5.4 所示，在服务器上有 10 个 KV-Direct 网卡，使用商用服务器可以轻松实现 10 亿键值 op/s 性能。

如图 5.23，服务器消耗 357 瓦的功率（在墙壁上测量）以达到 1.22 Gop/s GET 或 0.61 Gop/s PUT 的性能。

为了使两个 Xeon E5 CPU 的 80 个 PCIe Gen3 通道饱和，用带有 10 个 PCIe Gen3 x8 插槽的 SuperMicro X9DRX+-F 主板替换了基准测试服务器的主板，PCIe 拓扑如图 5.24 所示。

使用 PCIe x16 到 x8 转换器连接每个插槽上的 10 个可编程网卡，每个网卡上只启用一个 PCIe Gen3 x8 链路，因此每个网卡的吞吐量低于图 5.19。每个网卡在主机内存中拥有一个独占内存区域，并提供不相交的键分区。多个网卡遇到与

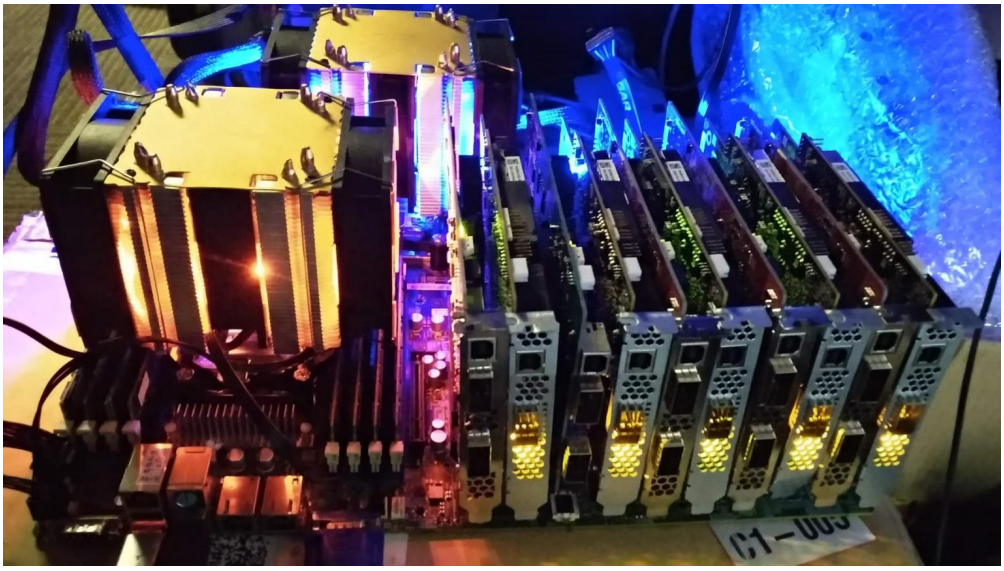


图 5.23　以 357 瓦功耗实现 12.2 亿次键值操作每秒的 10 卡 KV-Direct 系统。

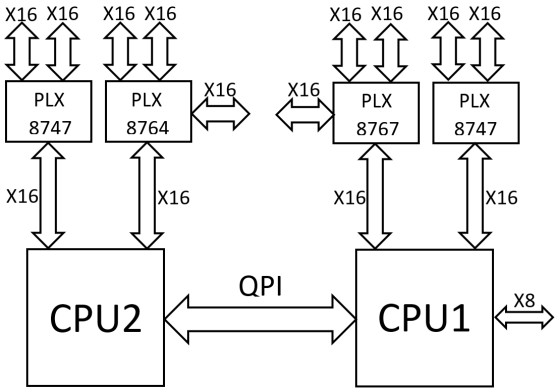


图 5.24　10 卡 KV-Direct 系统的 PCIe 拓扑图。

多核键值存储实现相同的负载不平衡问题。幸运的是，对于少量分区（例如 10 个），负载不平衡并不重要[30-31]。在 YCSB 长尾工作负载下，负载最高的网卡的平均负载为 1.5 倍，非常流行的键所增加的负载由无序执行引擎提供（§5.4.3）。相比之下，为了实现与 240 个 CPU 内核的匹配性能，最热 CPU 核心的负载将是平均值的 10 倍。图5.25显示 KV-Direct 吞吐量几乎与服务器上的网卡数量呈线性关系。

5.6.3　基于 SSD 的持久化存储

基于内存数据结构存储断电后数据会丢失。为了持久化，本节利用 SATA SSD 实现了持久化键值存储。由于服务器上的 SSD 数量是有限的，且操作系统和应用程序也运行在 SSD 上，键值存储需要与操作系统和应用程序共享 SSD 硬件。为此，SSD 提供块存储（block storage）和键值存储两种访问接口。与内存键值存储采用专门预留的内存空间类似，键值存储也位于专门预留的块存储空

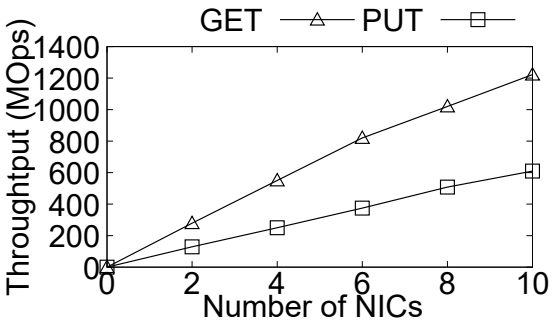


图 5.25　单机多网卡的性能可扩放性。

间内。CPU 需要两种访问块存储的方式：一是通过操作系统的存储协议栈来访问块设备，二是通过用户态的快速接口绕过操作系统直接访问。由于操作系统本身和很多软件运行在块存储上，保持第一种传统访问方式的兼容性是必要的。存储性能敏感的应用则使用本文提供的运行库来通过第二种方式访问块存储。

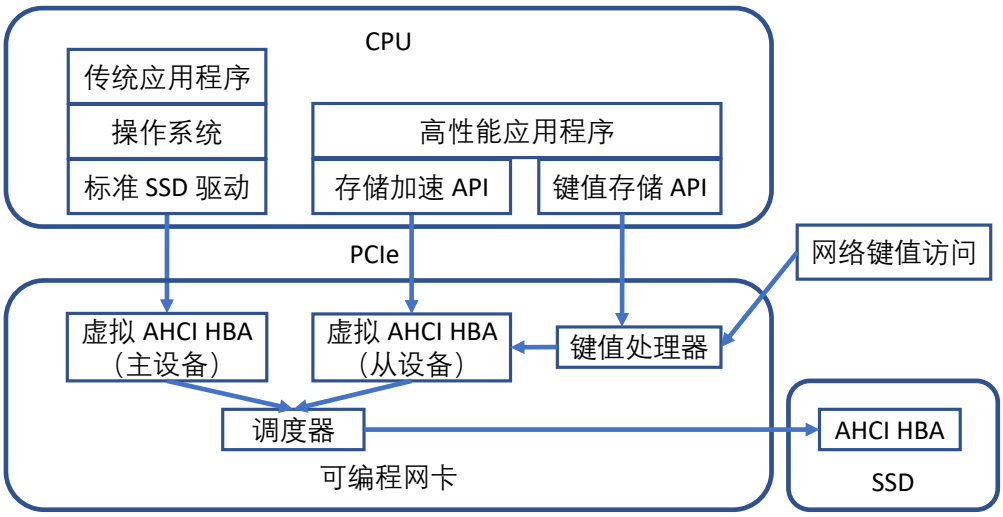


图 5.26　SSD 持久化存储架构。

如图 5.26，可编程网卡将 SSD 虚拟化成两个虚拟 AHCI HBA 设备。可编程网卡内的调度器将存储硬件的数据平面（如 SATA 的 32 个请求槽位和 NVMe 的请求队列）虚拟化成两个逻辑存储设备。存储硬件的控制寄存器（如 PCIe 配置寄存器）透传给主逻辑存储设备，由原有的操作系统管理。从逻辑存储设备没有控制面，仅有数据面，只能进行数据读写，不能进行管理操作。上述存储虚拟化架构无需可编程网卡管理控制面，简化了调度器的设计；而且保持了与原有存储设备驱动程序和操作系统的兼容性。如图 5.27，存储虚拟化后通过主逻辑存储设备和原有的操作系统和软件的顺序访问吞吐量没有明显变化。延迟从约 30 μs 升高到了约 60 μs，这是可编程网卡转发的开销。由于延迟增加，单线程随机访问（4K 块大小）的吞吐量也有所降低。在 64 个线程随机访问时，由于 SATA 只有 32 个请求槽位，即只能并行进行 32 个读写请求，吞吐量受限于请求的平均延

迟。

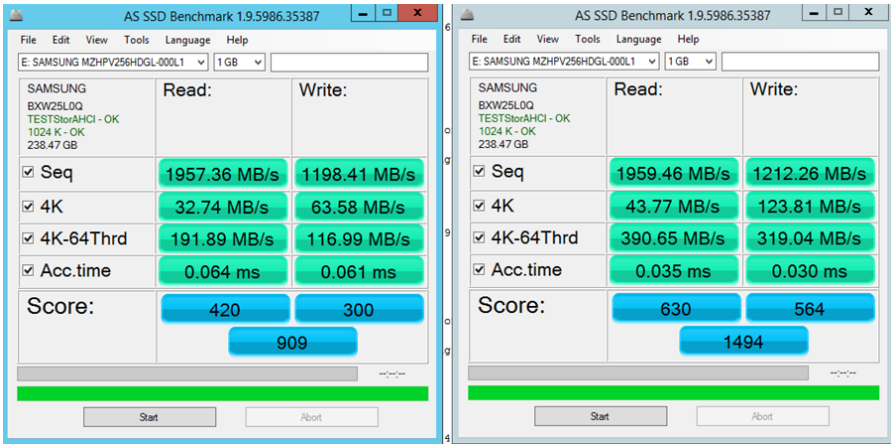


图 5.27　SSD 虚拟化的性能评估。左图为存储虚拟化后原有操作系统和 SSD 性能测试程序的结果。右图为不使用存储虚拟化时该 SSD 的性能测试结果。

为了提供高效的块设备访问接口，本节利用第 4 章的 PCIe I/O 管道，使应用程序可以通过存储加速 API 直接访问可编程网卡，绕过操作系统和驱动程序。存储加速 API 可以访问 SSD 上任意的存储块，避免与操作系统的文件系统冲突是应用程序的责任。通常，应用程序创建一个大文件以预留存储空间。实验表明，仅用单个 CPU 线程，存储加速 API 就可以充分利用 SSD 约 2 GB/s 的顺序读写吞吐量和约 50 K 次每秒的 4K 块大小随机读写吞吐量。传统操作系统存储协议栈需要 8 个 CPU 线程才能充分利用 4K 块大小随机读写的吞吐量。为了提供持久化的键值存储，键值处理器连接到虚拟存储的从设备上，把虚拟存储当作一大块内存来读写。本文没有针对 SSD 的读写特性进行优化，这将是未来的工作。

5.6.4　分布式键值存储

在分布式键值存储中，我们假设每台主机既是使用键值存储的客户端，又能分配一些内存资源作为键值服务器。每个键值对需要在多个服务器节点上复制，以提供高可用性，并提高键值访问的性能。传统的键值存储服务简单地根据键的哈希值选定服务器。然而，并不是每台主机都以相同的概率访问每个键。例如，图计算中一台主机如果负责处理一个顶点，那么访问该顶点对应的键值的概率将高于其他主机。这时，该顶点的键值对最好存储在该主机上，以便利用局部性加速访问。

分布式存储系统需要决定每个键值对在哪些主机上复制。读操作只需读取最近的副本。写操作则需要通过主节点（master）同步到所有的副本。如果复制份数过多，主节点将写操作同步到各个副本会成为瓶颈。为了平衡主节点与各个副本节点的负载，复制的份数取决于读写比例。假定读写操作的比例为 R，且 R

远大于 1[①]。可以推导得出，当主节点与副本节点的负载相等时，复制的份数约为 $\sqrt{R}$。相比单个副本，读操作的负载降低了约 $\sqrt{R}$ 倍；相比复制到每个主机，写操作的负载降低了约 $\sqrt{R}$ 倍。

决定复制的份数后，下一个问题是选定读取最频繁的主机来复制。本文在主节点的可编程网卡内为每个键分别维护读次数、写次数的近似计数器[②]。在写操作同步到各个副本时，主节点汇总所有副本的读次数。根据读写次数之比可以计算出最佳副本数量，并在与当前副本数量相差较多时增加或减少副本数量，使得副本存储在读次数最多的若干节点上。为了防止写操作稀少导致主节点不能及时响应大量的读请求，副本节点在收到大量读请求后也可以主动向主节点汇报。

另一个问题是一些键的访问频率可能很高，以至于单机吞吐量可能成为瓶颈。例如，分布式事务中的序列号发生器、热门网络资源的访问计数器和共享资源的锁需要高吞吐量的单键原子操作。最近的 NetCache[220]、NetChain[221] 等工作使用可编程交换机作为缓存，获得了比单个主机更高的单键读写性能，但仍然受到交换机性能的限制。

本文提出，使用多主复制（multi-master replication），单键读写性能可以随主机数量扩放，并保证强一致性。保证一致性的机制是令牌环（token ring），任意时刻有且仅有一个持有令牌的节点在处理写操作。令牌在各个主节点组成的环上按顺序传递，并附带当前最新的值。令牌环能够提升单键性能的关键是读写操作和很多原子操作是可结合（associative）的。例如，多个写操作可被归约为最后一次写操作；多个原子加减操作可以归约为一次原子加减操作；比较并交换的原子操作较为复杂，但多个操作也可以归约为一张大小不超过原子操作数量的查找表。因此，当一个节点不持有令牌时，就将收到的操作记录在缓冲区内，并归约这些操作。在令牌到来时，节点可以迅速在最新值的基础上执行归约后的操作，并将更新后的值随令牌发送给下一节点。此后，节点重放缓冲区内的操作并将每个操作的结果返回给客户端。理论分析表明，在请求均匀到达各个主节点时，系统的吞吐量和平均延迟都随主节点数量线性增加。最坏延迟是令牌环转一圈所需的时间。由于缓冲区内的操作被归约，每个节点的处理延迟是有界的，令牌环转一圈的时间也就有界。每个主节点所需的请求缓冲区大小等于最坏延迟与网络带宽的乘积。令牌环的开销是一个在网络中不断传递的数据包，即使没有读写请求，令牌也要在网络中不断传递。

实际的键值存储系统不只有一个访问频繁的键。为每个键设置一个令牌的

[①]在写入密集型负载中，如果仅有一个主节点，不做任何复制显然是性能最优的。为了高可用性，可能需要限制副本的最小数量。

[②]近似计数器是为了节约存储空间。例如，近似计数用 11 位真数（significand）和 5 位幂数（exponent）的浮点数表示。每次访问时以 2 的幂数次方分之一的概率将真数加 1。

网络开销过高，因此本文使用一个令牌服务所有的键。然而，等待所有键都处理完成后再集中发送有更新的所有键值对，会带来较高的延迟。假设一段时间内访问的键是随机的，也就是大多数键值操作涉及的键是不重复的。此时，令牌所附带的键值更新大小与缓冲区内的键值操作数量成正比，传输这些键值更新的时间不再是有界的，因此请求延迟会随负载因子接近 1 而趋于无穷大。为此，本文将相邻的主节点间流水线化键值处理和传输。主节点将缓冲区内的键值操作组织成按照键递增排序的优先队列，键值更新也按键递增顺序发送，令牌标示键值更新序列的尾部。当主节点收到键 K 的更新时，缓冲区内不超过 K 的键值操作就可以被处理，并沿令牌环发送到下一主节点。这样，令牌环上的多个主节点可以并发处理不同的键，即令牌传递方向上靠前的主节点处理较小的键。理论分析表明，在请求的键服从均匀分布且请求均匀到达各个主节点时，请求延迟仍然是有界的，与负载因子无关，且仅比单键情形下的延迟略高。

5.7　讨论

5.7.1　不同容量的网卡硬件

KV-Direct 的目标是利用数据中心的现有硬件来卸载重要的工作负载（键值访问），而不是设计特殊的硬件来实现最大的键值存储性能。可编程网卡通常包含有限数量的 DRAM 用于缓冲和连接状态跟踪。大型 DRAM 在芯片尺寸和功耗方面都很昂贵。

即使未来的网卡具有更快或更大的板载内存，在长尾工作负载下，本文的负载分配设计（§5.4.4）仍然显示出比简单的分区设计更高的性能。键统一根据网卡和主机内存容量。表 5.6 显示了具有 10 亿个键的长尾工作负载的最佳负载分配比率，不同的网卡 DRAM 和 PCIe 吞吐率以及不同的网卡和主机比率内存大小。如果网卡具有更快的 DRAM，则将更多负载分派给网卡。负载分配比率为 1 表示网卡内存的行为与主机内存的高速缓存完全相同。如果网卡具有更大的 DRAM，则将稍微少量的负载分派给网卡。如表 5.7 所示，即使网卡 DRAM 的大小只是主机内存的一小部分，吞吐量增益也很大。

乱序执行引擎（§5.4.3）可以应用于需要隐藏延迟的各种应用程序，本文希望未来的 RDMA 网卡能够支持更更性能的原子操作。

在 40 Gbps 网络中，网络带宽限制了非批量传输的键值吞吐量，因此本文使用客户端批处理。通过更高的网络带宽，可以减少批量大小，从而降低延迟。在 200 Gbps 网络中，KV-Direct 网卡无需批量传输即可达到 180 Mop/s。

KV-Direct 利用广泛部署的可编程网卡和 FPGA 实现[48,133]。FlexNIC[122,222] 是另一种具有可重构匹配行动表（RMT）[103] 的可编程网卡的有前途的架构。Net-

表 5.6　不同网卡 DRAM / PCIe 吞吐率（垂直）和网卡 /主机内存大小比（水平）下长尾工作负载的最佳负载分配比。

	1/1024	1/256	1/64	1/16	1/4	1
1/2	0.366	0.358	0.350	0.342	0.335	0.327
1	0.583	0.562	0.543	0.525	0.508	0.492
2	0.830	0.789	0.752	0.718	0.687	0.658
4	1	0.991	0.933	0.881	0.835	0.793
8	1	1	1	0.995	0.937	0.885

表 5.7　与简单分区相比，负载分派的相对吞吐量。行列标题与表 5.6 相同。

	1/1024	1/256	1/64	1/16	1/4	1
1/2	1.36	1.39	1.40	1.37	1.19	1.02
1	1.71	1.77	1.81	1.79	1.57	1.01
2	2.40	2.52	2.62	2.62	2.33	1.52
4	3.99	4.02	4.22	4.27	3.83	2.52
8	7.99	7.97	7.87	7.56	6.83	4.52

Cache[220] 在基于 RMT 的可编程交换机中实现键值缓存，显示了在基于 RMT 的网卡中构建 KV-Direct 的潜力。

5.7.2　对现实世界应用的性能影响

当 KV-Direct 应用于端到端应用程序时，后台计算显示了潜在的性能提升。在 PageRank[69] 中，由于每次边缘遍历都可以通过一次键值操作实现，因此 KV-Direct 在具有 10 个可编程网卡的服务器上支持 1.22G TEPS。相比之下，GRAM[223] 支持每台服务器 250M TEPS，受交错计算和随机内存访问的约束。

KV-Direct 支持用户定义的函数和向量操作（表 5.2），可以通过将客户端计算卸载到硬件来进一步优化 PageRank。类似的参数适用于参数服务器[81]。本文希望未来的工作可以利用硬件加速的键值存储来提高分布式应用程序的性能。

5.7.3　可编程网卡内的有状态处理

本文第 4 章的 ClickNP 架构比较适合无状态或状态简单的流水线式数据包处理，而对基于连接状态或应用层请求的处理，就显得捉襟见肘。例如，四层负载均衡器应用中的调度器和哈希表与应用逻辑耦合紧密，既不容易扩放到大量并发连接，代码的可维护性也不强。事实上在开发过程中花费了大量的时间来解决死锁问题。

KV-Direct 提出了有状态处理（stateful processing）的一种新架构。KV-Direct 相比传统通用处理器架构的根本区别在于控制、计算和访存的分离。在传统处理

器中，计算和访存共享同一条指令流，从而对于计算和访存交替的串行程序，计算和访存逻辑之间往往需要相互等待，导致计算和访存的吞吐量都不能被充分利用。KV-Direct 使用分离的控制指令流、计算指令流和访存指令流，且利用乱序执行引擎充分挖掘 KV 操作间的并行性，使得计算和访存可以充分流水线化。开发者需要将请求的处理过程划分为计算和访存交替的若干阶段，将请求处理过程抽象成一个状态机。控制指令流管理请求的状态并将下一阶段任务分发到计算或访存部件。计算或访存部件处理完成一个阶段的任务后，返回控制器。

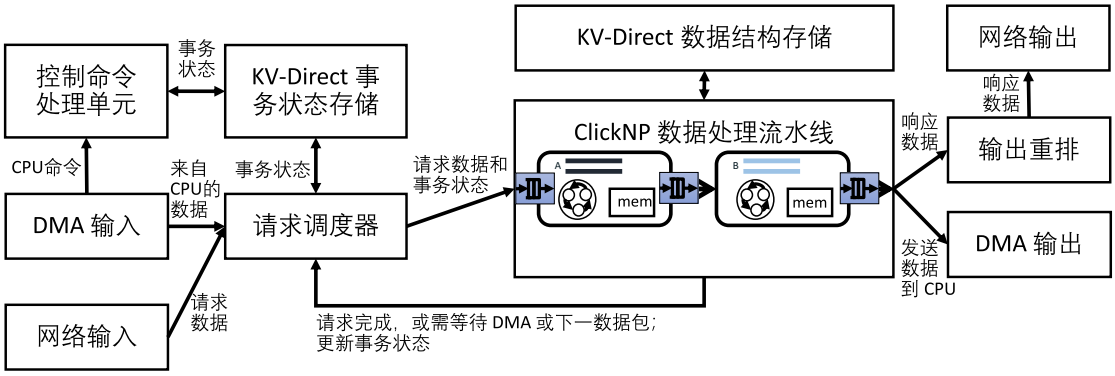


图 5.28　基于 KV-Direct 的可编程网卡应用层架构。

基于 KV-Direct 的有状态处理架构如图 5.28 所示。事务代表着前后依赖关系，例如有状态网络处理中的一个连接，分多个数据包的一个应用层 HTTP 请求，或者键值存储中对同一个键（key）的操作。同一个事务中的不同请求需要依次处理，而不同事务中的请求可以并发处理。为了隐藏延迟、最大化并发处理能力，请求调度器从基于 KV-Direct 的事务状态键值存储中查找该请求对应的事务编号，并将正在处理事务的请求排入队列。基于 ClickNP 的数据处理流水线根据请求数据和事务状态进行处理，在处理过程中可能查询其他的数据结构（如内存分配表、主机虚拟地址映射表、防火墙规则表、路由表等）。如果请求处理完成，响应数据进入输出重排模块，重新排列响应的顺序以满足事务处理的一致性要求（例如，不同事务的请求也要按照到达顺序依次响应），最终输出到网络。如果请求的处理还需要依赖下一数据包或从主机内存 DMA 来的数据，为了不阻塞数据处理流水线，该请求会返回到调度器，等待依赖操作完成后再进行下一阶段的处理。

KV-Direct 架构可以作为很多可编程网卡应用的基础，如第 4 章的有状态网络功能（如四层负载均衡器）和第 6 章的可扩放 RDMA。

5.7.4　从键值扩展到其他数据结构

KV-Direct 实现的是键-值映射的哈希表数据结构。数据中心以 Redis[78] 为代表的键值存储系统还支持二级索引、有序键值集合、消息队列等数据结构。这些

更复杂的数据结构的本质特征都是可编程网卡内的有状态处理（第 5.7.3 节）。

对于消息队列，用 FPGA 处理的优势是快速的中心化协调。在生产者-消费者（producer-consumer）模式中，消息队列需要将生产者的消息分发到各消费者，这是一个中心化 FIFO 的抽象。由于 CPU 单核处理能力有限，并行或分布式消息队列往往需要牺牲一定的一致性 ① 来提升性能可扩放性。而 FPGA 硬件逻辑的时钟频率足以处理 100 Gbps 线速的请求，无需牺牲一致性。

在基于 KV-Direct 的消息队列中，消息存储在固定大小的缓冲区组成的环形链表中。采用环形链表便于在空间不足时分配新的固定大小缓冲区，也便于回收长时间空闲的缓冲区。对于生产者-消费者模式，为所有生产者维护一个头指针，为所有消费者维护一个尾指针。对于发布者-订阅者（publisher-subscriber）模式，为所有发布者维护一个头指针，为每个订阅者分别维护一个尾指针，使得不同的订阅者可以不同的速度接收消息。

在二级索引中，请求按照一级索引在请求调度器中排队，并按照标准 KV-Direct 的方法获取匹配第一级索引的第二级索引元数据。元数据存储在事务状态存储中作为缓存。然后就第二级索引查询生成一个新的请求，按照一二级合并索引在请求调度器中排队。处理该请求时，从事务状态存储中取出对应的元数据，发起第二次 DMA 查询。此外，二级索引相比单级索引的另一个复杂之处在于第二级索引在增删元素时可能经常需要改变哈希表的大小，从而需要重新哈希，在此期间该第一级索引对应的操作需要挂起等待。用 FPGA 处理二级索引的优势是细粒度的访存延迟隐藏，如某个第二级索引重新哈希期间其他第二级索引可以操作。

5.8　相关工作

作为一种重要的基础设施，分布式键值存储系统的研究和开发受到性能的驱动。大量分布式键值存储基于 CPU。为了降低计算成本，Masstree [32]，MemC3 [33] 和 libcuckoo [34] 优化锁，缓存，哈希和内存分配算法，而 KV-Direct 带有新的哈希值专为 FPGA 设计的表和内存管理机制，以最大限度地减少 PCIe 流量。MICA [30] 将哈希表分区到每个核心，从而完全避免同步。然而，这种方法为偏移的工作负载引入了核心不平衡。

为了摆脱操作系统内核的开销，Netmap[13] 和 DPDK[224] 直接轮询来自网卡的网络数据包，而 mTCP[17] 和 SandStorm[16] 使用用户态轻量级网络堆栈处理这些数据包。键值存储系统[27-31] 受益于这种高性能优化。作为朝着这个方向迈出的又一步，最近的作品[35,46,46-47,47] 利用 RDMA 网卡的基于硬件的网络堆栈，使

① 一致性即先到先服务的顺序特性

用双面 RDMA 作为键值存储客户端和服务器之间的 RPC 机制进一步提高每核吞吐量并减少延迟。尽管如此，这些系统仍受 CPU 限制（§5.2.3）。

另一种不同的方法是利用单侧 RDMA。Pilaf [209] 和 FaRM [70] 采用单向 RDMA 读取进行 GET 操作，FaRM 实现了使网络饱和的吞吐量。Nessie [208]，DrTM [72]，DrTM + R [73] 和 FaSST [20] 利用分布式事务来实现单向 RDMA 的 GET 和 PUT。但是，PUT 操作的性能受到一致性保证的不可避免的同步开销的影响，受到 RDMA 原语的限制[47]。此外，客户端 CPU 涉及键值处理，将每个核心的吞吐量限制在客户端的大约 10 Mops。相比之下，KV-Direct 将 RDMA 原语扩展到键值操作，同时保证服务器端的一致性，使键值存储客户端完全透明，同时实现高吞吐量和低延迟，即使对于 PUT 操作也是如此。

作为一种灵活且可定制的硬件，FPGA 现已广泛部署在数据中心规模[48,133]中，并且针对可编程性进行了大幅改进[54,156]。一些早期的工作已探索在 FPGA 上构建键值存储系统。但是其中一些只使用片上数据存储（大约几 MB 内存)[225]或板载 DRAM（例如 8 GB 内存)[226-228]，因此存储容量有限。工作[218] 专注于提高系统容量而不是吞吐量，并采用 SSD 作为板载 DRAM 的二级存储。工作[225,227]只能存储固定大小的键值对，这样的键值存储系统只能用于一些特定的应用，不够通用。工作[217,229] 使用主机 DRAM 存储哈希表，而工作[230] 使用网卡 DRAM 作为主机 DRAM 的缓存，但它们没有针对网络和 PCIe DMA 带宽进行优化，导致性能不佳。KV-Direct 充分利用了网卡 DRAM 和主机 DRAM，使基于 FPGA 的键值存储系统通用，并且能够进行大规模部署。此外，精心的硬件和软件协同设计，以及对 PCIe 和网络的优化，将本文的性能推向了物理极限。

利用支持 P4[102] 的可编程交换机[107] 来加速键值存储系统也是近年来研究的热点。SwitchKV[231] 利用基于内容的路由将请求路由到基于缓存键的后端节点，NetCache [220] 进一步将访问频繁的键值缓存在交换机中。NetChain[221] 在网络交换机中实现了一个高一致性、容错的键值存储。

数据存储系统中二级索引是通过主键以外地其他键检索数据的重要功能[232-233]。SLIK [233] 在键值存储系统中使用 B+ 树算法支持多个二级键。探索如何支持二级索引以帮助 KV-Direct 迈向通用数据存储系统将会很有趣。SwitchKV [231] 利用基于内容的路由将请求路由到基于缓存键的后端节点，NetCache [220] 进一步将键值缓存在交换机中。这种负载平衡和缓存也将使系统受益。Eris [200] 利用网络序列发生器来实现高效的分布式事务，这可以为客户端同步的单侧 RDMA 方法带来新的生命。

5.9　本章小结

本章描述了 KV-Direct 的设计和评估，这是一个高性能的内存中键值存储。在计算机系统设计的悠久历史中，KV-Direct 是另一项利用可重新配置硬件来加速重要工作量的练习。KV-Direct 能够通过精心设计硬件和软件来获得卓越的性能，以消除系统中的瓶颈并实现接近底层硬件物理极限的性能。

第 6 章　SocksDirect 通信原语加速

6.1　引言

本章的主题是操作系统通信原语加速，在全文中的位置如图 6.1 所示。

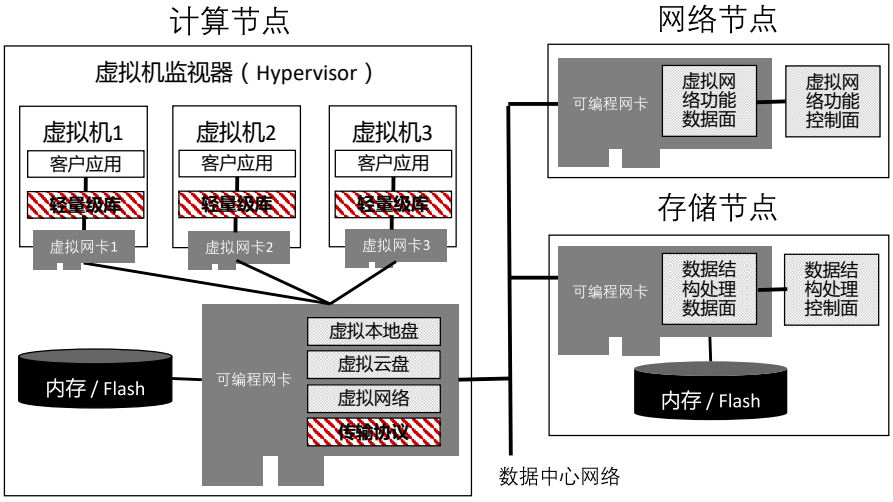


图 6.1　本章主题：操作系统通信原语加速，用粗斜线背景的方框标出。

作为本文介绍的最后一个研究工作，本文实现了一个与现有应用兼容的用户态套接字通信库，分别使用共享内存和 RDMA 作为单机进程间和不同主机之间的通信方式。作为补充，基于前面章节提出的 ClickNP 编程框架和 KV-Direct 数据结构处理服务，在可编程网卡内实现了连接数可扩放的 RDMA。

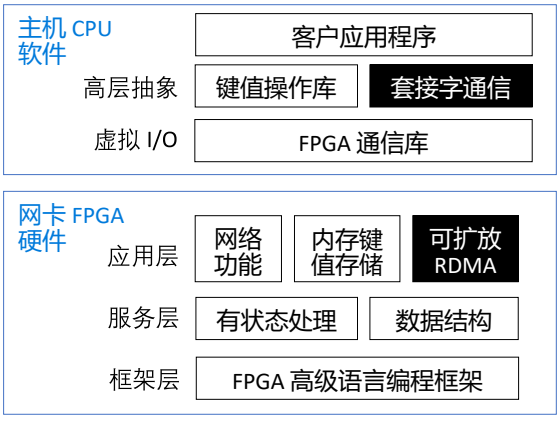


图 6.2　本章在可编程网卡软硬件架构中的位置。

Socket API 是现代应用程序中使用最广泛的通信原语，通常用于进程，容器和主机之间的通信。Linux 套接字只能实现比裸的硬件（如共享内存和 RDMA）差一到两个数量级的延迟和吞吐量。近年来，大量工作旨在改善套接字性能。现有方法或者优化内核网络协议栈[234-236]，或者将 TCP/IP 协议栈移动到用户空间[16-19,22]，或者把传输层卸载到 RDMA 网卡[43-44]。但是，所有这些解决方案都

在兼容性和性能方面存在限制。它们中的大多数在诸如进程 fork、事件轮询、多应用程序套接字共享和主机内通信等方面与 Linux 套接字不完全兼容。其中一些[17] 存在隔离问题，不允许多个应用程序共享网卡。尽管这些工作致力于提升性能，仍有很大的性能提升空间。现有的工作都不能达到接近裸 RDMA 和共享内存的性能，因为它们无法消除多线程同步、缓冲区管理和内存复制等重要的开销。例如，套接字在进程中的多个线程之间共享，因此，许多系统使用锁来避免竞争条件。

认识到这些限制，本章设计了 SocksDirect，一个用户空间套接字系统，可以同时实现兼容性、隔离性和高性能。

- **兼容性**。应用程序无需修改，即可使用 SocksDirect 作为 Linux 套接字的替代品。SocksDirect 同时支持主机内和主机间通信，并且在进程 fork 和线程创建期间行为正确。如果远程对端不支持 SocksDirect，则系统将透明地回退到标准 TCP。
- **隔离性**。首先，SocksDirect 保持了应用程序和容器间的隔离性，即任何应用程序都不能监听或干扰其他应用程序之间的连接，并且一个恶意程序不能使它的连接对端出现错误的行为。其次，SocksDirect 可以实施访问控制策略，以阻止未授权的连接。
- **高性能**。SocksDirect 提供高吞吐量和低延迟，可与原始 RDMA 和共享内存相媲美，并可通过多个 CPU 核心实现性能扩放。

为了实现高性能，SocksDirect 充分利用了现代硬件的能力。它利用 RDMA 进行主机间通信，并使用*共享内存*（shared memory）进行主机内通信。但是，将套接字操作转换为 RDMA 和共享内存操作并非易事。简单的解决方案可能会违反兼容性或在表格上留下很多性能。例如，在套接字 send() 返回后，应用程序可能会覆盖缓冲区。然而，RDMA 发送操作需要写保护缓冲区。现有的工作[43] 或者提供与未修改应用程序不兼容的零拷贝 API，或者需要协议栈管理内部缓冲区并从缓冲区复制数据。

为了同时实现所有这三个目标，首先需要了解 Linux 套接字如何提供兼容性和隔离性。Linux 套接字为应用程序提供虚拟文件系统（VFS）抽象。通过这种抽象，应用程序开发人员可以像操作文件那样进行通信，而无需深入研究网络协议细节。这种抽象还在共享地址和端口空间的应用程序之间提供了良好的隔离。但是，VFS 抽象非常复杂，许多 API 本身就不具备可扩展性[17,119-120]。

尽管 VFS 具有普遍性和复杂性，许多常用的套接字操作实际上都很简单。因此，本章的设计原则是针对常见情况进行优化，同时保持兼容性。

为了在连接管理中保持隔离的同时加速数据传输，SocksDirect 将控制和数据平面分开[12]。在每个主机中，引入一个*管程*（monitor）守护进程作为*控制平*

面来强制执行访问控制策略，管理地址、端口资源，分派新连接以及在通信对端之间建立传输通道。*数据平面*由动态加载的用户空间库 libsd 处理，它拦截对 Linux 标准 C 库的函数调用。libsd 在用户空间中实现套接字 API，并将非套接字相关的 API 转发给内核。应用程序可以通过在 Linux 中使用 *LD_PRELOAD* 环境变量来加载库来利用 libsd。

在 SocksDirect 中，数据传输和事件轮询在对端进程之间直接处理，而连接建立则委托给管程。本章利用多种技术高效利用硬件并提高系统效率。通常，线程和 fork 产生的进程之间共享套接字连接。为了避免访问套接字元数据和缓冲区带来的竞争条件（race condition），需要同步。通过基于令牌的共享方法，而不是为每个操作加锁，SocksDirect 消除了常见情况下的同步开销。从网卡发送和接收数据时，现有系统为每个数据包分配缓冲区。为了消除缓冲区管理开销，本章设计了每个连接独享的环形缓冲区，在发送方和接收方各有一个拷贝，然后利用 RDMA 和共享内存从发送方环形缓冲区同步到接收方。为了实现较大消息的零拷贝，SocksDirect 利用虚拟内存机制重新映射页面。

SocksDirect 实现了与底层共享内存队列和裸 RDMA 的性能接近的延迟和吞吐量。在延迟方面，SocksDirect 对主机内套接字实现了 0.3 微秒 RTT，是 Linux 的 1/35，仅比裸机共享内存队列高出 0.05 微秒。对于主机间套接字，SocksDirect 在 RDMA 主机之间实现了 1.7 微秒的 RTT，几乎与裸的 RDMA 写入相同，达到了 Linux 的 1/17。在吞吐量方面，单个线程可以每秒发送 23 M 个主机内消息（Linux 的 20 倍）或 18 M 个主机间（15 倍于 Linux，1.4 倍于裸的 RDMA 写入）。对于大型消息，通过零拷贝，单个连接即可饱和利用 100 Gbps 网卡的带宽。上述性能可随内核数量线性扩放。SocksDirect 为实际应用程序提供了显著的加速。例如，Nginx [237] 的 HTTP 请求延迟降低到了 1/5.5，标准 RPC 库的延迟也可以降低 50%。

总之，本章做出了以下贡献：

- 分析 Linux 套接字的开销。
- 设计和实现了 SocksDirect，一个与 Linux 兼容、能保持应用程序间隔离的高性能用户空间套接字系统。
- 支持 fork、无锁连接共享、环形缓冲区和零拷贝等技术，可能在套接字以外的许多场景中都很有用。
- 评估显示 SocksDirect 可以实现与 RDMA 和共享内存队列相当的性能。

6.2　背景

6.2.1　Linux 套接字简介

套接字（socket）是应用程序，容器和主机之间的标准通信原语。图 6.3 示意了使用套接字原语的典型服务器应用程序的伪代码。首先，服务器创建一个套接字文件描述符 `lfd` 用于监听端口、接收新连接，并设置为非阻塞以便异步处理。然后创建一个事件文件描述符 `efd` 用于接收新连接事件和各个连接发送接收数据的事件。接下来进入事件循环，对每个接收到的事件，如果是新连接，就调用 `accept` 接收之，并加入事件监视；如果是已有连接上有数据到达，就接收该连接上的所有数据（因为有接收缓冲区大小限制，一次 `recv` 可能接收不完）；如果对端已经准备好接收（即接收缓冲区有空闲空间），就将待发送的数据发送出去。

现代操作系统中的 TCP 套接字通常具有三个功能:（1）寻址，找到并连接到另一个应用程序；（2）提供可靠且有序的通信通道，由整数 *文件描述符*（File Descriptor）标识；（3）查询来自多个通道的事件，例如 poll 和 epoll。大多数 Linux 应用程序使用准备驱动（readiness-driven）的 I/O 多路复用模型，即操作系统告诉应用程序哪些文件描述符已准备好接收或发送，然后应用程序可以准备缓冲区并发出接收或发送操作。

6.2.2　Linux 套接字中的开销

现代数据中心网络具有微秒级延迟和数十 Gbps 吞吐量。但是，传统的 Linux 套接字是在具有共享数据结构的操作系统内核空间中实现的，这使得套接字成为在多个主机上运行的通信密集型应用程序众所周知的瓶颈[6]。除了主机间通信之外，同一主机上的微服务和容器经常相互通信，使得主机内套接字通信在云时代变得越来越重要。在压力测试下，Nginx [238]，Memcached [197] 和 Redis [239] 等应用程序在内核中消耗 50% 到 9%％的 CPU 时间，主要用于处理 TCP 套接字操作[17]。

从概念上讲，Linux 网络协议栈由三层组成。首先，VFS 层为应用程序提供套接字 API（例如 *connect*，*send* 和 *epoll*）。套接字连接是双向、可靠且有序的管道，由整数 *文件描述符*（文件描述符）标识。其次，传统的 TCP/IP 传输层提供 I/O 复用、拥塞控制、丢包恢复、路由和服务质量保证（QoS）功能。第三，网卡层与网卡硬件（或用于主机内套接字的虚拟环回接口）通信以发送和接收数据包。众所周知，VFS 层贡献了网络协议栈中的很大一部分成本[119-120]。这可以通过一个简单的实验来验证：主机中两个进程之间的 Linux TCP 套接字的延迟和吞吐量只比管道（pipe）、FIFO 和 Unix 域套接字（Unix domain socket）差一点。（表 6.2 中，Linux TCP 延迟为 11 μs、吞吐量为 0.9 M op/s，管道、FIFO 和 Unix 域套

```
int lfd = socket(...); // listen file descriptor (fd)
bind(lfd, listen_addr_and_port, ...);
listen(lfd, BACKLOG);
fcntl(lfd, F_SETFL, fcntl(lfd,F_GETFL,0) | O_NONBLOCK);
int efd = epoll_create(MAXEVENTS); // event fd
epoll_ctl(efd, EPOLL_CTL_ADD, lfd, ...);
while (true) { // main event loop
  int n = epoll_wait(efd, events, MAXEVENTS, 0);
  for (int i=0; i<n; i++) { // iterate events
    if (events[i].data.fd == lfd) { // new connection
      int cfd = accept(sfd, ...); // connection fd
      epoll_ctl(efd, EPOLL_CTL_ADD, cfd, ...);
      fcntl(cfd,F_SETFL,fcntl(cfd,F_GETFL,0)|O_NONBLOCK);
    }
    else if (events[i].events & EPOLLIN){//ready to recv
      do { // fetch all received data
        cnt = recv(events[i].data.fd, recvbuf, buflen);
        recvbuf = next_recv_buf();
      } while (cnt > 0);
      // do processing
    }
    else if (events[i].events & EPOLLOUT){//ready to send
      do { // flush send buf
        cnt = send(events[i].data.fd, sendbuf, sendlen);
        sendbuf += cnt; sendlen -= cnt;
      } while (cnt > 0 && sendlen > 0);
    }
  }
}
```

图 6.3　典型套接字服务器应用程序的伪代码，显示最重要的套接字操作。套接字连接是由整数 ***FD***（文件描述符）标识的 **FIFO** 字节流通道。**Linux** 使用就绪驱动的 **I/O** 多路复用模型，其中操作系统告诉应用程序哪些文件描述符已准备好接收或发送，然后应用程序可以准备缓冲区并发出套接字操作。

表 6.1　Linux 套接字的开销。

类型	开销	本章的解决方案
每操作	内核穿越（系统调用）	用户态库 (§6.3)
每操作	并发线程和进程的套接字文件描述符锁	基于令牌的套接字共享 (§6.4.1)
每数据包	传输层协议（TCP/IP）	使用 RDMA 或共享内存 (§6.4.2)
每数据包	缓冲区管理	新的环形缓冲区设计 (§6.4.2)
每数据包	I/O 多路复用	使用 RDMA 或共享内存 (§6.4.2)
每数据包	中断处理	事件通知 (§6.4.4)
每数据包	进程唤醒	事件通知 (§6.4.4)
每字节	数据复制	页面重映射 (§6.4.3)
每连接	内核文件描述符分配	文件描述符重映射表 (§6.4.5)
每连接	TCB 锁管理	分派到 libsd (§6.4.5)
每连接	分派新连接	守护进程 (§6.4.5)

接字的延迟为 8～9 μs、吞吐量为 0.9～1.2 M op/s。）管道、FIFO 和 Unix 域套接字绕过了传输层和网卡层，但它们的性能仍然不尽如人意。

Clark 等的经典工作[119] 将套接字开销划分为每数据包和每字节开销。在现代协议栈中，由于连接创建也有显著开销[17,234]，我们引入一类新的开销：每连接开销；由于每次套接字操作在 VFS 层有一定的开销，与其处理的数据包数量无关（有些操作，如 `dup2`，根本不处理数据包），我们引入另一类新的开销：每次操作开销。下文将套接字开销分为四种类型：每次操作，每个数据包，每个字节和每个连接。

1. 每次操作的开销

内核穿越（kernel crossing）。传统上，套接字 API 在内核中实现，因此需要针对每个套接字操作进行内核穿越（即系统调用）。更糟糕的是，为防止 Meltdown[240] 攻击，内核页表隔离（KPTI）补丁[241] 使内核穿越变得 4 倍昂贵，如表 6.2 所示（使用 KPTI 补丁前，内核穿越需要 50 ns，而使用 KPTI 后需要 200 ns）。本章的目标是绕过内核而不影响安全性（§6.3）。

套接字文件描述符锁。许多应用程序都是多线程的，原因有两个。首先，与 FreeBSD 不同，用于在 Linux 中读写磁盘文件的异步接口无法利用操作系统缓存和缓冲区，因此应用程序继续使用多线程和同步接口[242]。其次，许多 Web 应用程序框架更喜欢用同步编程模型处理每个用户请求，因为同步编程模型更容易编写和调试[6]。进程中的多个线程共享套接字连接。此外，在进程 fork 之后，父进程和子进程共享现有套接字。套接字也可以通过 Unix 域套接字传递给另一个进程。为了保护并发操作，Linux 内核为每个套接字操作获取每个套接字

表 6.2　往返延迟和单核吞吐量操作（测试平台设置在 §6.5.1 中）。未特别说明的情况下，消息大小为 8 个字节。

操作	延迟 (μs)	吞吐量 (M 次操作每秒)
核间缓存迁移	0.03	50
轮询 32 个空队列	0.04	24
系统调用（KPTI 前）	0.05	21
自旋锁（无竞争）	0.10	10
分配和释放缓冲区	0.13	7.7
自旋锁（有竞争）	0.20	5
无锁共享内存队列	0.25	27
主机内 SocksDirect	0.30	22
系统调用（KPTI 后）	0.20	5.0
复制 1 个内存页（4 KiB）	0.40	5.0
协作式上下文切换	0.52	2.0
映射一个内存页（4 KiB）	0.78	1.3
主机内通过网卡通信	0.95	1.0
原子共享内存队列	1.0	6.1
映射 32 个内存页（128 KiB）	1.2	0.8
打开套接字文件描述符	1.6	0.6
单边 RDMA 写操作	1.6	13
双边 RDMA 发送 / 接收操作	1.6	8
主机间 SocksDirect	1.7	8
进程唤醒	2.8~5.5	0.2~0.4
Linux 管道 / FIFO	8	1.2
Linux 中的 Unix 域套接字	9	0.9
主机间 Linux TCP 套接字	11	0.9
复制 32 个内存页（128 KiB）	13	0.08
主机间 Linux TCP 套接字	30	0.3

锁[120,234-235]。表 6.2 表明，即使没有多核争用，受原子操作保护的共享内存队列相比无锁队列的延迟可达 4 倍，吞吐量也只有无锁队列的 22%。本章的目标是通过优化常见情况并删除常用套接字操作中的同步操作来尽可能降低同步开销（§6.4.1）。

2. 每个数据包的开销

传输协议（TCP / IP）。传统上，TCP/IP 是数据中心传输协议的事实标准。TCP/IP 协议处理、拥塞控制和丢包恢复在每个发送和接收的数据包上消耗 CPU。此外，丢包检测，基于速率的拥塞控制和 TCP 状态机使用定时器，很难实现微秒级粒

度和低开销[17]。幸运的是，近年来在许多数据中心见证了 RDMA 的大规模部署[41,243-244]。RDMA 将传输协议卸载到 RDMA 网卡，提供了与 TCP/IP 相当的基于硬件的传输层。对于主机间套接字，本章的目标是利用 RDMA 硬件传输层的高吞吐量、低延迟和接近零的 CPU 开销（§1）。对于主机内套接字，本章的目标是完全绕过传输层。

缓冲管理。传统上，CPU 通过*环形缓冲区*（ring buffer）从网卡发送和接收数据包。环形缓冲区由固定数量的固定长度的元数据条目组成。每个条目都指向一个存储数据包有效负载的缓冲区。要发送或接收数据包，需要分配和释放缓冲区。表 6.2 显示了环形缓冲区的成本。此外，为了确保可以接收 MTU 大小的分组，每个接收缓冲区应该具有至少一个 MTU 的大小。但是，许多数据包小于 MTU [245]，因此内部碎片会降低内存利用率。虽然现代网卡支持 LSO 和 LRO [246] 以批量处理多个数据包，但本章的目标是完全消除缓冲区管理的开销（§6.4.2）。

I/O 多路复用。对于传统的网卡，接收到的不同连接的数据包通常在环形缓冲区中混合，因此网络协议栈需要将数据包分类到相应的套接字缓冲区中。现代网卡支持接收数据包转向[108]，它可以将特定连接映射到专用环形缓冲区，该缓冲区由高性能套接字系统使用[17,22,234]。本章利用 RDMA 网卡中的类似功能，将接收到的数据包解复用（demultiplex）到每个连接专属的环形缓冲区。

中断处理。 Linux 网络协议栈分为系统调用和中断上下文，因为它处理来自应用程序和硬件设备的事件。例如，当应用程序调用 send 时，网络协议栈进程上下文中将数据包发送出去（如果窗口允许）。当网卡接收到该数据包时，网卡向 CPU 发送一个中断，然后网络协议栈在中断上下文中处理接收到的数据包。TCP 拥塞控制中的 ACK 时钟（ACK clocking）机制[247] 要求及时处理中断和定时器。中断上下文不一定与应用程序在同一核心上，导致 CPU 核的局部性下降。但是，RDMA 网卡硬件实现了需要精确计时的数据包处理，因此主机 CPU 不再需要处理大部分数据平面的中断。

进程唤醒。当进程调用远程过程调用（RPC）并等待回复时，是否应让 CPU 切换到准备运行的其他进程？Linux 的答案是肯定的，这个进程切换的唤醒睡眠过程需要 3 到 5 μs，如表 6.2 所示。在主机内 RPC 的往返时间内，两次进程唤醒贡献了超过一半的延迟。对于通过 RDMA 的主机间 RPC，小于 MTU 大小的小消息在网络上的往返延迟甚至低于进程唤醒的延迟。为此，许多分布式系统和用户态协议栈使用轮询来避免唤醒开销。然而，简单的轮询方法会为每个线程消耗一个 CPU 核，不能扩放到大量线程。为了隐藏微秒级 RPC 延迟[6]，通过 `sched_yield` 的协作上下文切换比进程唤醒要快得多。本章的目标是高效地在多个线程之间共享核心（§6.4.4）。

容器网络。许多容器部署使用隔离的网络命名空间，这些容器通过虚拟覆盖网络（virtual overlay network）进行通信。在 Linux 中，虚拟交换机[168] 在主机网卡和容器中的虚拟网卡之间转发数据包。这种架构会在每个数据包上产生多个上下文切换和内存拷贝的开销，虚拟交换机成为瓶颈[248]。Slim [249] 将三次内核往返减少到一次。最近的几个工作[25,117,250-251] 将所有操作委托给作为守护进程运行的虚拟交换机，因此它增加了数据路径上的延迟和 CPU 成本。本章的解决方案是集中式的控制平面和分布式的数据平面（§6.4.5）。

3. 每字节的开销

有效载荷（payload）复制。在大多数套接字系统中，`send` 和 `recv` 的语义会导致应用程序和网络协议栈之间的内存复制。对于非阻塞 `send`，系统需要将数据复制出缓冲区，因为应用程序可能会在 `send` 返回后立即覆盖缓冲区。简单地删除拷贝可能会违反应用程序的正确性。零拷贝 `recv` 甚至比 `send` 更难。Linux 提供了基于准备就绪的事件模型，即应用程序知道传入的数据（例如通过 `epoll`）然后调用 `recv`，因此网卡接收但未传递给应用程序的数据必须存储在系统缓冲区中。因为 `recv` 允许应用程序提供任何缓冲区作为数据目标，所以系统需要将数据从系统复制到应用程序缓冲区。本章的目标是在标准套接字应用程序中为较大有效载荷传输实现零拷贝（§6.4.3）。

4. 每条连接的开销

内核文件描述符分配。在 Linux 中，每个套接字连接都是 VFS 中的文件，因此需要分配整数文件描述符和 *inode*。用户空间套接字的挑战是有许多 API（例如 open，close 和 epoll）同时支持套接字和非套接字文件描述符（例如文件和设备），因此必须将套接字文件描述符与其他文件描述符区分开来。用户空间中的 Linux 兼容套接字[22,43] 通常在内核中打开一个文件以获取每个套接字的虚拟文件描述符，因此它们仍然需要内核文件描述符分配。LOS [25] 将文件描述符空间划分为用户和内核部分，但违反了 Linux 分配最小可用文件描述符的属性。但是，许多应用程序，如 Redis [78] 和 Memcached [26] 都依赖于此属性。本章的目标是在保持兼容性的同时绕过内核套接字文件描述符分配（§6.4.5）。

TCP 控制块管理中的锁。在建立连接期间，Linux 获取几个全局锁来分配 TCB（TCP Control Block，TCP 控制块）。最近的工作，如 MegaPipe [235] 和 FastSocket [234] 通过对全局表进行分区来减少锁争用，但正如表 6.2 所示，非争用自旋锁是仍然很贵。本章将工作分发到每个进程中的用户空间库 libsd （§6.4.5）。

新连接调度。多个进程和线程可以侦听同一端口以接受传入连接。在 Linux 中，处理 `accept` 调用的核心在传入连接的队列上进行竞争。本章利用委托（delegation）比锁[250] 更快的事实，使用管程守护程序来分派新连接（§6.4.5）。

6.2.3　高性能套接字系统

学术界和工业界都提出了许多高性能套接字系统，如表 6.3 所示。

内核网络协议栈优化：第一类工作是优化内核 TCP / IP 协议栈。FastSocket [234]，Affinity-Accept [252]，FlexSC [253] 和零拷贝套接字[254-256] 实现良好的兼容性和隔离性。

MegaPipe [235] 和 StackMap [236] 提出了新的 API 来实现零拷贝和改进 I / O 多路复用，代价是需要修改应用程序。但是，大量的内核开销仍然存在。支持零拷贝的挑战是套接字语义。

用户态 TCP / IP 协议栈：第二类工作完全绕过内核 TCP / IP 协议栈并在用户空间（user-space）中实现 TCP / IP。在这个类别中，IX [11] 和 Arrakis[12] 是新的操作系统架构，它使用虚拟化来确保安全性和隔离性。IX 利用 LwIP [257] 在用户空间中实现 TCP / IP，同时使用内核转发每个数据包以实现性能隔离和 QoS。相比之下，Arrakis 将 QoS 卸载到网卡，因此绕过数据平面的内核。这些工作使用网卡在同一主机中的应用程序之间转发数据包。如表 6.2 所示，从 CPU 到网卡的往返（hairpin）延迟远远高于核心间缓存迁移延迟。吞吐量也受到内存映射 I / O（MMIO）的门铃（doorbell）延迟和 PCIe 带宽的限制[130,258]。

除了这些新的操作系统体系结构，许多用户空间套接字在 Linux 上使用高性能数据包 I / O 框架，例如 Netmap [13]，Intel DPDK [14] 和 PF_RING[15]，以便直接访问用户空间中的网卡队列。SandStorm [16]，mTCP [17]，Seastar [18] 和 F-Stack [19] 提出了新的 API，因此需要修改应用程序。大多数 API 更改旨在支持零拷贝，标准 API 仍然会复制数据。FaSST [20] 和 eRPC [21] 提供 RPC API 而不是套接字。LibVMA [22]，OpenOnload [23]，DBL [24]，LOS [25] 和 TAS[259] 符合标准套接字 API。

用户空间 TCP / IP 协议栈提供了比 Linux 更好的性能，但它仍然不接近 RDMA 和共享内存。一个重要原因是现有的工作都不支持在线程和进程之间共享套接字，导致 fork 和容器热迁移中的兼容性问题，以及多线程锁开销。

首先，在 LibVMA 和 RSocket 中，当进程 fork 后，对于父进程在 fork 前创建的套接字，子进程或者获取所有已有套接字的所有权，或者不能访问任何套接字（即这些套接字仍归父进程所有）。没有办法独立地控制每个套接字的所有权。但是，许多 Web 服务[237,260-263] 和键值存储[26] 都有一个主进程来从监听套接字中接受新连接，然后它可能 fork 一个子进程来处理请求，子进程需要访问新连接的套接字。与此同时，父进程仍需要通过监听套接字接受新连接。这使得此类 Web 服务无法正常工作。更棘手的情况是父进程和子进程可以通过现有套接字同时写入日志服务器。

表 6.3 高性能套接字系统的比较。

	FastSocket	MegaPipe / StackMap	IX	Arrakis	SandStorm / mTCP	LibVMA	OpenOnload	Rsocket / SDP	FreeFlow	SocksDirect
类别	内核优化		用户态 TCP/IP 协议栈					卸载到 RDMA 网卡		
兼容性										
对现有应用程序透明	✓		✓	✓		✓	✓	✓	✓	✓
`epoll` (Nginx, Memcached 等)	✓	✓	✓	✓	✓	✓	✓			✓
与普通 TCP/IP 对端兼容	✓	✓	✓	✓	✓	✓	✓			✓
主机内通信	✓	✓		✓					✓	✓
多个应用程序监听同一端口	✓	✓						✓	✓	✓
完整的 fork 支持	✓	✓					✓			✓
容器热迁移	✓	✓								✓
隔离性										
访问控制策略	内核		内核	内核				内核	守护进程	守护进程
容器 / 虚拟机间的隔离	✓		✓	✓				✓	✓	✓
QoS（性能隔离）	内核	内核	内核	网卡	网卡	网卡	网卡	网卡	守护进程	网卡
消除性能开销										
数据面上的系统调用			批处理	✓	批处理	✓	✓	✓	✓	<16KB 消息
套接字文件描述符锁										✓
传输协议								✓	✓	✓
缓冲区管理										✓
I/O 多路复用与中断处理		改进	✓	✓	改进	✓	✓	✓	✓	✓
进程唤醒										✓
数据复制		✓			✓					≥16KB 消息
内核文件描述符分配		✓			✓					✓
TCB 管理与连接分派	✓	✓	✓	✓	✓					✓

第二，多线程在应用程序中很常见。应用程序在套接字操作中承担竞争条件的风险，或者每次操作必须采用套接字文件描述符锁。后一种方法保证了正确性，但即使没有锁之间的争用（contention），锁也会损害性能。

将传输层卸载到网卡：为了降低操作系统通信原语的开销，一系列工作将套接字系统的一部分卸载到网卡硬件上。TCP 卸载引擎（TCP Offload Engine，TOE）[39] 将部分或全部的 TCP / IP 协议栈卸载到网卡，但由于通用处理器的性能按摩尔定律迅速增长，这些专用硬件的性能优势有限，仅在专用领域中获得成功，例如 iSCSI HBA 存储卡[264] 和无状态卸载（例如校验和、接收侧扩放（RSS）、大发送数据卸载（LSO）、大接收数据卸载（LRO）[246]）。近年来，由于数据中心的硬件趋势和应用需求，有状态卸载的故事开始复兴[40]。因此，RDMA[35] 在生产数据中心中广泛使用[41]。RDMA 提供了两类抽象：读写远程共享内存的单边原语，以及类似套接字发送接收语义的双边原语。与基于软件的 TCP / IP 网络协议栈相比，RDMA 使用硬件卸载来提供超低延迟和接近零的 CPU 开销。为了使套接字应用程序能够使用 RDMA，RSocket[43]，SDP [44] 和 UNH EXS [45] 将套接字操作转换为双边 RDMA 原语。它们具有相似的设计，其中 RSocket 的开发最活跃，是套接字转换 RDMA 的事实标准。FreeFlow [251] 利用 RDMA 网卡提供容器（container）覆盖网络（overlay network），它利用共享内存进行主机内部通信，利用 RDMA 进行主机间通信。为了实现 RDMA 虚拟化，FreeFlow 本质上是一种微内核架构，控制面和数据面操作都由用户态的虚拟交换机处理。FreeFlow 使用 RSocket 将套接字转换为 RDMA。

但是，由于 RDMA 和套接字的抽象不匹配，这些工作有局限性。在兼容性方面，首先，它们缺乏对几个重要 API 的支持，例如 *epoll*，因此它与许多应用程序不兼容，包括 Nginx，Memcached，Redis 等。这是因为 RDMA 仅提供传输功能，而 *epoll* 是与 OS 事件通知集成的文件抽象。其次，RDMA QP 不支持 *fork* 和容器热迁移[251]，因此 RSocket 也有同样的问题。第三，由于 RSocket 使用 RDMA 作为网络数据包格式，因此无法连接到常规 TCP / IP 对等体。这是一个部署挑战，因为分布式系统中的所有主机和应用程序必须同时切换到 RSocket。本章的目标是透明地检测远程端是否支持 Rsocket，如果没有则回退到 TCP / IP。在性能方面，它们无法删除有效负载拷贝，套接字文件描述符锁，缓冲区管理，进程唤醒和每个连接开销。例如，RSocket 在发送方和接收方分配缓冲区并复制有效负载。与 Arrakis 类似，RSocket 使用网卡进行主机内通信，从而导致性能瓶颈。

6.3　架构概览

为了简化部署和开发[149]，以及消除内核交叉开销，本章在用户空间而不是内核中实现 SocksDirect。要使用 SocksDirect，应用程序通过设置 `LD_PRELOAD` 环境变量来加载用户空间库 libsd。libsd 拦截标准 C 库（glibc）中与文件描述符操作相关的所有 Linux API，在用户空间中实现套接字功能。从安全的角度来看，因为 libsd 驻留在应用程序地址空间中，它的行为不可信。例如，一个恶意程序可能向 RDMA QP 中直接写入任意的消息，从而绕过 libsd 库中的安全检查。此外，如表 6.4 所示，尽管大多数套接字操作可以在调用者本地或连接的两端之间实现，还有很多套接字操作需要中心化的协调。例如，TCP 端口号是一个需要集中分配的全局资源[234,251]。因此，需要在 libsd 之外的可信组件来强制实施访问控制和管理全局资源。

初始化		连接管理	
API	类别	API	类别
socket	Local	connect	NoPart
bind	NoPart	accept(4)	P2P
listen	NoPart	*fcntl, ioctl*	Local
socketpair	Local	(get,set)sockopt	Local
getsockname	Local	*close*, shutdown	P2P
malloc	Local	getpeername	Local
realloc	Local	*dup(2)*	P2P
epoll_create	Local	*epoll_ctl*	Local
数据传输		**进程管理**	
API	类别	API	类别
recv(from,(m)msg)	P2P	*pthread_create*	NoPart
write(v)	P2P	*clone*	NoPart
read(v)	P2P	*execve*	NoPart
memcpy	Local	*exit*	P2P
(p)select	P2P	*sleep*	P2P
(p)poll	P2P	*daemon*	P2P
epoll_(p)wait	P2P	*sigaction*	Local

表 6.4　与套接字相关、被 libsd 截获的主要 Linux API。分类包括本地（Local）、两端之间（P2P）和需集中协调、不可划分（NoPart）。斜体的 API 表示除了套接字，还有操作系统的其他用途。粗体的 API 比其他 API 的调用更频繁，因此更值得优化。

为此，本章在每个主机上设计一个 *monitor* 守护进程（下称 管程）来协调控

制平面操作，例如连接创建。为了保证隔离性，我们把所有应用程序和管程作为一个不共享数据结构的分布式系统，使用消息通信作为唯一的通信机制。在每个主机中，所有加载 libsd 的应用程序必须与主机的管程建立共享内存（共享内存）队列，从而形成控制平面。在数据平面上，应用程序构建端到端（peer-to-peer）的队列以直接通信，从而减轻了管程守护程序的负担。管程是一个单线程的用户态程序，从所有应用程序的端到端队列中轮询消息。图 6.4 显示了 SocksDirect 的体系结构。

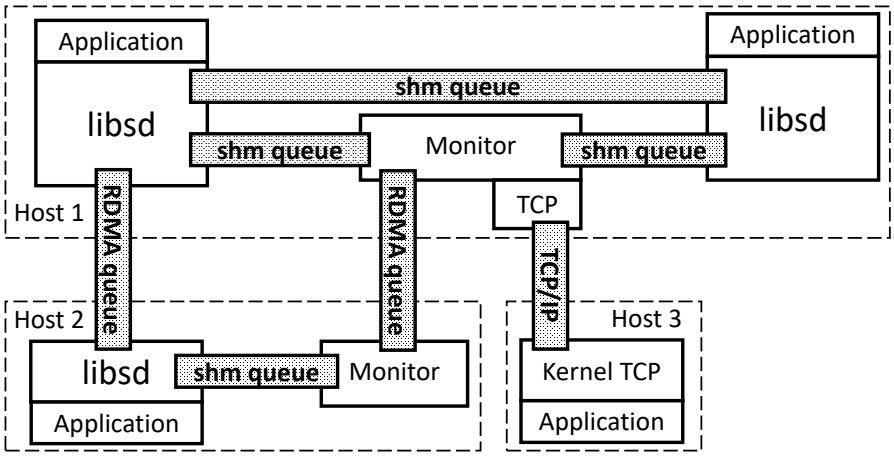


图 6.4　SocksDirect 的体系结构。主机 1 和 2 具有 RDMA 能力，而主机 3 不具备 RDMA。

为了实现低延迟和高吞吐量，SocksDirect 使用共享内存进行主机内部和 RDMA 进行主机间通信。每个套接字连接都映射到共享内存队列或 RDMA QP。共享内存或 RDMA QP 由唯一令牌标记，因此其他非特权进程无法访问它。套接字发送操作转换为对远程端点上的套接字缓冲区的共享内存或 RDMA 写操作。

对于主机内（intra-host）通信，通信发起者首先向本地管程发送请求，然后管程在两个应用程序之间（可能在不同的容器中）建立共享内存队列。之后这两个应用程序可以直接通信。

对于主机间（inter-host）通信，两个主机的管程都参与其中。当应用程序连接到远程主机时，其本地管程会建立常规 TCP 连接，并检测远程主机是否支持 SocksDirect 和 RDMA。如果都支持，两个管程之间会建立一个 RDMA 队列，以便两个主机之间未来的套接字连接可以更快创建。远程端的管程调度与目标的连接，并帮助两个应用程序建立 RDMA 队列，如图 6.4 中的主机 1 和 2 之间。如果远程主机不支持 SocksDirect，它将继续使用 TCP 连接，如图 6.4 中的主机 1 和 3 之间。详细的连接管理协议在第 6.4.5 节中介绍。

为了确保线程安全并避免锁，以及支持 fork 和容器热迁移，本章针对只有一对发送和接收线程处于活动状态的常见情况进行了优化，同时确保所有情况下的正确性（第 6.4.1 节）。为了消除缓冲区管理开销，设计了一个环形缓冲区，

每个主机间消息平摊下来只需要一次 RDMA 写操作，每个主机内消息需要一次缓存迁移（第 6.4.2 节）。本章进一步设计了一种零拷贝机制，可以在发送和接收端消除较大消息的数据拷贝（第 6.4.3 节）。最后，第 6.4.4 节提供了事件通知机制。

如图 6.5 所示，libsd 运行库由 API 封装、虚拟文件系统（VFS）、队列、传输等四层构成。API 封装层使用 *文件描述符重映射表*来区分套接字文件描述符与内核文件描述符（例如文件和设备），在用户空间中实现套接字功能，并将其他系统调用通过标准 C 库（glibc）转发到内核。虚拟文件系统层实现了连接创建与关闭、事件轮询与通知、多进程共享套接字、fork、容器迁移等功能，是最复杂的一层。虚拟文件系统的旁边是信号层，负责接收来自操作系统的事件以及与管程、对端通信。下面一层是基于环形缓冲区的无锁队列。最下层是传输层，用共享内存（SHM）或 RDMA 实现。

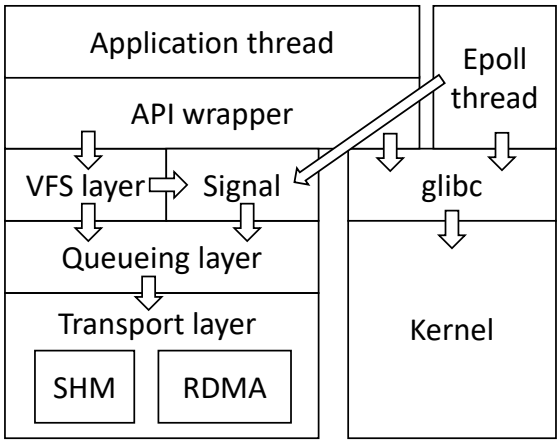


图 6.5　libsd 运行库的架构。

6.4　系统设计

6.4.1　基于令牌的套接字共享

大多数套接字系统为每个文件描述符维护一个锁，以使线程和进程共享套接字。以前的工作[120,265] 表明，许多套接字操作不可交换，并且不能始终避免同步。本章利用共享内存消息传递比锁[250] 便宜得多的事实，并使用消息传递作为唯一的同步机制。

逻辑上，套接字由两个相反传输方向的 FIFO 队列组成，每个 FIFO 具有多个并发的发送方和接收方。系统设计目标是在保持 FIFO 语义的同时最大化通用情况性能。本文观察到应用程序的如下两个特征：首先，由于成本高，高性能应用程序很少 fork 和创建线程。其次，几个进程并发从共享套接字发送或接收并不常见，因为套接字的字节流语义使得很难避免接收部分消息。需要同时发送或

接收的应用程序通常使用消息代理（message broker）[77,266-267] 而非直接共享套接字。常见的进程间套接字共享情况是应用程序隐式地将套接字从一个进程迁移到另一个进程，例如将事务从主进程卸载到工作进程。

本章的解决方案是每个 *套接字队列*（套接字的一个传输方向）有一个 *发送令牌*和一个 *接收令牌*。每个令牌都由一个 *活跃线程*持有，它具有发送或接收的权限。因此，在任何时间点只有一个活动的发送方线程和一个活动的接收方线程。套接字队列在线程和进程之间共享，允许来自一个发送方和一个接收方的并发无锁访问（详细信息将在第 6.4.2 节中讨论）。当另一个线程想要发送或接收时，它应该请求 *接管*（take over）令牌。

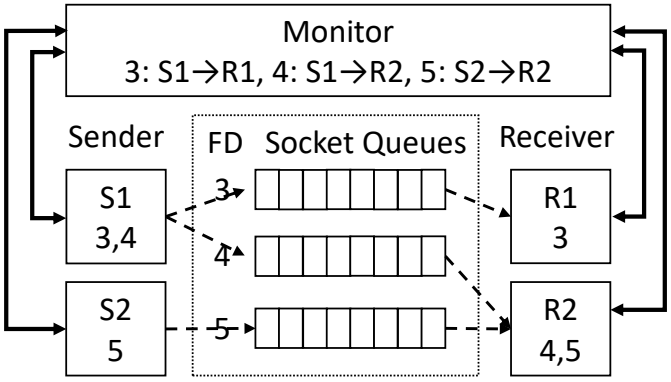


图 6.6　两个发送方和两个接收方线程共享一个基于令牌的套接字。虚线箭头表示每个套接字的活动发送方和接收方。每个线程跟踪其活动套接字并通过独占队列与管程通信。

每种类型的操作的详细信息如下:a）数据传输（`send` 和 `recv`），b）添加新的发送方和接收方（`fork` 和线程创建），c）容器热迁移，和 d）连接关闭。

1. 发送/接收操作

当一个线程没有发送令牌但想要通过套接字发送时，非活动线程需要*接管*令牌。如果在非活动线程和活动线程之间创建直接通信通道，则需要点对点的队列，其数量为线程数的平方，或者是具有锁的共享队列。为了避免这两种开销，在*接管*过程中使用管程作为代理。由于接管是不频繁的操作，管程一般不会成为瓶颈。此消息传递设计还具有以下优点：发送方进程可以位于不同的主机上，这在容器热迁移中非常有用。

接管过程如下：非活动发送方通过共享内存队列向管程发送*接管*命令。管程轮询来自各个共享内存队列的消息，将发送方添加到等待列表中，并将命令代理到目前的活动发送方。当活动发送方收到请求时，它会将发送令牌发送给管程。管程将令牌授予等待列表中的第一个非活动发送方，并更新活动发送方。非活动发送方可以在收到令牌后发送。此机制无死锁，因为至少有一个发送方或管程持有发送令牌。它也是无饥饿的，因为每个发送方最多可以出现在等待列表中一次，并按 FIFO 顺序提供。接收器侧的接管过程类似。

接管过程需要 0.6 微秒，因此，如果多个进程并发地通过同一个套接字发送，总吞吐量可能下降到 1.6 M 次操作每秒。但是，如果我们简单地使用锁，通常情况的吞吐量将下降到 5 M 次操作每秒，远低于基于令牌的套接字共享所能达到的 27 M 次操作每秒吞吐量。

2. Fork，Exec 和线程创建

套接字数据共享。主要的挑战是在 `fork` 和 `exec` 之后共享套接字元数据、缓冲区和底层的传输层。在 fork 之后，内存空间变为写时复制，并在 exec 之后被擦除，但是套接字文件描述符仍需要可用。SocksDirect 使用共享内存（共享内存）来存储套接字元数据和缓冲区，因此在 fork 之后，数据仍然是共享的。要在 exec 之后附加共享内存，libsd 将连接到管程以获取其父进程的共享内存密钥。在 fork 之后，因为父进程看不到子进程创建的套接字，所以子进程创建一个新的共享内存来存储新套接字的元数据和缓冲区。

下面考虑底层传输层机制。基于共享内存的传输层不需要特殊处理，因为 fork / exec 之前创建的共享内存仍然在 fork / exec 之后共享。但是，RDMA 在 fork / exec 后存在问题，因为 DMA 内存区域不在共享内存中。它们在 fork 之后变为写时复制，而网卡仍然从原始物理页面 DMA，因此子进程不能使用现有的 RDMA 资源。而在 exec 之后，整个 RDMA 上下文将被清除。本章的解决方案是让子进程在 fork / exec 之后重新初始化 RDMA 资源（PD，MR 等）。当子进程使用在 fork 之前创建的套接字时，它会要求管程与远程端点重新建立 RDMA QP。因此，对端进程可能会看到一个套接字的两个或更多 QP，但它们链接到套接字元数据和缓冲区的唯一拷贝。下一节（§6.4.2）我们将看到，我们仅使用 RDMA 单边写原语，因此使用任何一个 QP 都是等效的。图 6.7 显示了一个 fork 示例。

文件描述符空间共享。与套接字数据不同，文件描述符空间在 fork 之后变为写时复制：fork 前创建的文件描述符是共享的，但新文件描述符由创建者进程独享。因此，只需将文件描述符重映射表驻留在堆内存（heap memory）中，利用 fork 之后操作系统的写时复制机制。要在 exec 之后恢复文件描述符重映射表，它将在 exec 之前复制到共享内存，并在 libsd 初始化期间拷贝回新进程。

安全。安全性是一个问题，因为恶意进程可能将自己伪装成特权父进程的子进程。要在管程中标识父关系和子关系，当应用程序调用 `fork`，`clone` 或 `pthread_create` 时，libsd 首先生成一个用于配对的密钥并将其发送到管程，然后调用 *libc* 中的原始函数。在 fork 之后，子进程为管程创建一个新的共享内存队列并发送密钥（子进程继承父内存空间，从而知道该密钥）。因此，管程可以将子进程与父进程配对。

管程操作。在 fork，exec 或线程创建时，管程需要新进程添加到每个现有套接字的发送方和接收方列表中，以便管理接管操作。

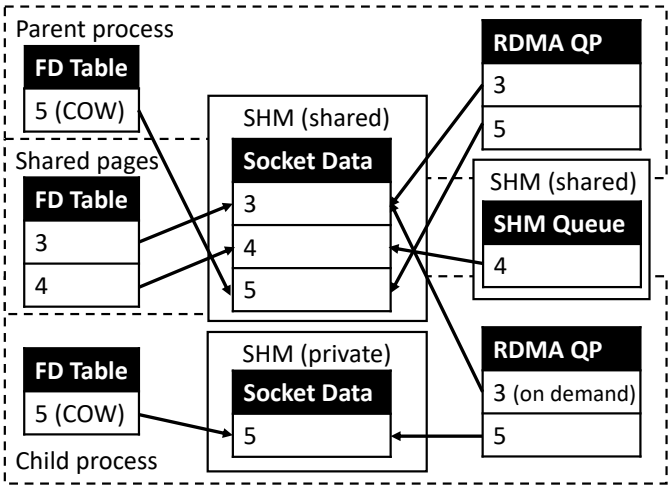


图 6.7　fork 后的内存布局。文件描述符 3 和 4 在 fork 之前创建并因此共享。在 fork 之后，父进程和子进程分别创建一个新的文件描述符 5，它在文件描述符表中写入时被复制。文件描述符 3 和 4 的套接字元数据和缓冲区在共享内存中并因此被共享。子进程创建一个新的共享内存来存储文件描述符 5 的套接字元数据和缓冲区，当它再次 fork 时，它将与子进程共享。RDMA QP 位于私有内存中，而共享内存队列是共享的。

3. 容器热迁移

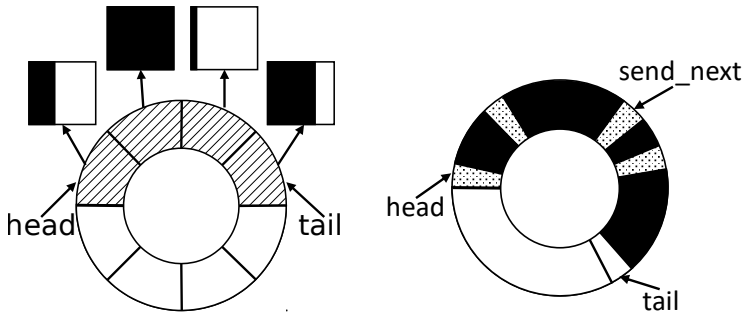
在套接字队列中迁移剩余数据。由于 libsd 在与应用程序相同的进程中运行，因此其内存状态将与应用程序一起迁移到新主机。内存状态包括套接字队列，因此不会丢失正在传输（已发送但未接收）的数据。套接字只能在容器内共享，并且容器中的所有进程都会一起迁移，因此迁移后可以取消分配原始主机上的内存。

管程状态的迁移。管程跟踪侦听套接字信息，活动线程和每个连接的等待列表以及共享内存密钥。在迁移期间，旧管程会转储已迁移容器的状态并将它们发送到新管程。

建立新的通信通道。迁移后，所有通信通道都已过时，因为共享内存在主机上是本地的，而 RDMA 不支持热迁移[249,251]。首先，新主机上的迁移容器需要与本地管程建立连接。本地管程指示以下过程。两个容器之间的主机内连接可能变为主机间，因此 libsd 在这种情况下创建 RDMA 连接。两个容器之间的主机间连接可能成为主机内部，libsd 创建共享内存连接。最后，libsd 重新建立剩余的主机间 RDMA 和主机内共享内存连接。

6.4.2　基于 RDMA 和共享内存的环形缓冲区

传统上，网络协议栈使用环形缓冲区从网卡发送和接收数据包。如图 6.8a 所示，传统的环形缓冲区由一组定长的元数据构成，每个元数据指向一个定长的内存页面以存储有效载荷。这种设计会导致内存分配开销和内部碎片。传统网卡



(a) 传统环形缓冲区。　　**(b)** SocksDirect 的环形缓冲区。

图 6.8　环形缓冲区数据结构。阴影部分是元数据，黑色部分是有效载荷。

使用这种设计的原因是环形缓冲区的数量有限，需要多个连接复用一个环形缓冲区。这种设计可以将有效载荷的元数据移动到每个连接的元数据队列，而无需复制有效载荷的内容。幸运的是，单边 RDMA 写操作（write verb）开辟了新的设计可能性。本章的创新是 *每个套接字连接拥有专有的环形缓冲区*并将数据包背靠背存储，如图 6.8b 所示。发送方确定环形缓冲区内的偏移量（即 *tail* 指针），然后使用单边 RDMA 写操作将数据包写入远程内存中 *tail* 指针所指向的位置。在传输过程中，接收端 CPU 不需要做任何事情。当接收端应用程序调用 `recv` 操作时，数据从 *head* 指针所指向的环形缓冲区位置出队。当 *head* 指针移动到与 *tail* 重合时，指针指向的元数据为空，因此接收端的 libsd 库能检测到队列空。需要注意，*head* 和 *tail* 指针分别由接收方和发送方在本地维护，因此无需同步这两个指针。通过共享内存传输数据的过程类似，因为共享内存和 RDMA 都支持写操作。

为了判断环形缓冲区是否已满，发送方将保持 *队列信用*（queue credits）计数，指示环形缓冲区中的空闲字节数。当发送方将数据包入队时，会消耗入队数据包大小的信用。信用不足时，发送方会阻塞等待。当接收方使数据包出队时，它会在本地增加计数器，并在计数器超过环形缓冲区大小的一半时，向发送方的内存中写入 *信用返回标志*。发送方在检测到该标志时重新获得队列信用。请注意，队列信用机制与拥塞控制无关；后者由网卡硬件处理[243]。

发送和接收方各有一个环形缓冲区拷贝。上述机制仍然需要发送端进行内存分配，因为发送方需要缓冲区来构造 RDMA 消息。其次，上述机制不支持容器热迁移，因为 RDMA 队列中未来得及接收的剩余数据很难迁移。第三，本章的目标是批量处理小消息以提高吞吐量。为此，本章在发送方和接收方都保留了一份环形缓冲区的拷贝。发送方写入其本地环形缓冲区，并调用单边 RDMA 写操作以使发送方与接收方的环形缓冲区同步。为了最大限度地减少空闲链路上的延迟，并最大化繁忙链路上的吞吐量，本文设计了一种自适应的批处理机制（adaptive batching）。libsd 为每个环形缓冲区创建一个 RDMA 可靠连接（RC）队

列对（QP），并维护一个 RDMA 消息的计数器。如果计数器未超过阈值，则为每个套接字 send 操作发送 RDMA 消息。否则，暂时不发送消息，而是用 *send_next* 标记第一个未发送的消息。完成 RDMA 写操作后，libsd 会在图 6.8b 中发送包含所有未发送更改的消息（从 *send_next* 到 *tail*）。对于共享内存，由于高速缓存一致性（cache coherence）硬件可以自动进行核间同步，只需一个由两个进程共享的环形缓冲区[①]。

有效载荷和元数据之间的一致性。对于共享内存，英特尔和 AMD 的 x86 处理器提供了全序写入排序（total store ordering）[268-269]，这意味着每个 CPU 核观察到其他 CPU 核的写入顺序是相同的。8 字节 MOV 指令是原子的，所以写数据包头是原子的。由于发送方在有效负载之后写入数据包头，因此接收方读取的消息是一致的，不需要内存栅栏（memory barrier）指令。

因为 RDMA 不能确保消息中各个部分的写入顺序[35]，确实需要确保消息完全到达再处理消息。虽然在使用 go-back-0 或 go-back-N 丢包恢复 [70] 的 RDMA 网卡中，消息的写入总是顺序的，但对于具有选择性重传的更高级网卡，情况并非如此[247,270]。在 libsd 中，发送方使用 RDMA *write with immediate*（带立即数的写）操作在接收方生成完成事件（completion event）。接收方轮询 RDMA *完成队列*而非环形缓冲区。RDMA 能够确保接收方的缓存一致性，并且保证完成事件晚于数据写入 libsd 环形缓冲区。

平摊轮询开销。当不经常使用套接字时，轮询环缓冲区会浪费接收方的 CPU 周期。本章使用两种技术分摊轮询开销。首先，对于 RDMA 队列，利用 RDMA 网卡将事件通知复用到单个队列中。每个线程对所有 RDMA QP 使用 *共享完成队列*，因此它只需要轮询一个队列而不是所有套接字队列。

其次，每个队列可以在*轮询*（polling）和*中断*（interrupt）模式之间切换。管程（monitor）的队列始终处于轮询模式。每个队列的接收方维护一个连续空轮询的计数器。当它超过阈值时，接收器向发送器发送消息，通知队列正在进入中断模式，并在短时间后停止轮询。当发送方以中断模式写入队列时，它还会通知管程，管程将通知接收方恢复轮询。

6.4.3　零拷贝

零拷贝的主要挑战是维持套接字 API 的语义。幸运的是，虚拟内存提供了一个间接层，许多相关工作利用了这种 *页面重映射*（page remapping）技术，从而可以将物理页面从发送方的虚拟地址重新映射到接收方，而不是复制。Linux 零复制套接字[256] 仅支持发送方，它的原理是将数据页设置为写时复制。但是，许多应用程序经常在调用 send 后覆盖发送缓冲区，因此写时复制机制只是将复制

[①]共享内存即一个物理页面分别映射到两个进程的用户地址空间。

从调用 send 操作的时间延迟到第一次覆盖的时间。为了实现零拷贝接收，20 年前，BSD [254] 和 Solaris [255] 将应用程序缓冲区的虚拟页面重新映射到操作系统缓冲区的物理页面。但是，如表 6.2 所示，在现代 CPU 上，由于内核穿越（kernel crossing）和 TLB 刷新开销，重新映射一个页面的开销甚至比复制它更高。最近，许多高性能 TCP/IP 协议栈[235-236] 和 socket-to-RDMA 库[43-44] 提供标准的套接字 API 和备用的零拷贝 API，但它们都没有实现标准 API 的零拷贝。此外，没有现有的工作支持主机内套接字的零拷贝。

要启用零拷贝，需要修改网卡驱动程序以公开与页面重新映射相关的几个内核函数。为了分摊页面重新映射成本，libsd 仅对 send 或 recv 使用零拷贝，且有效载荷至少为 16 KiB。较小的消息被直接复制。

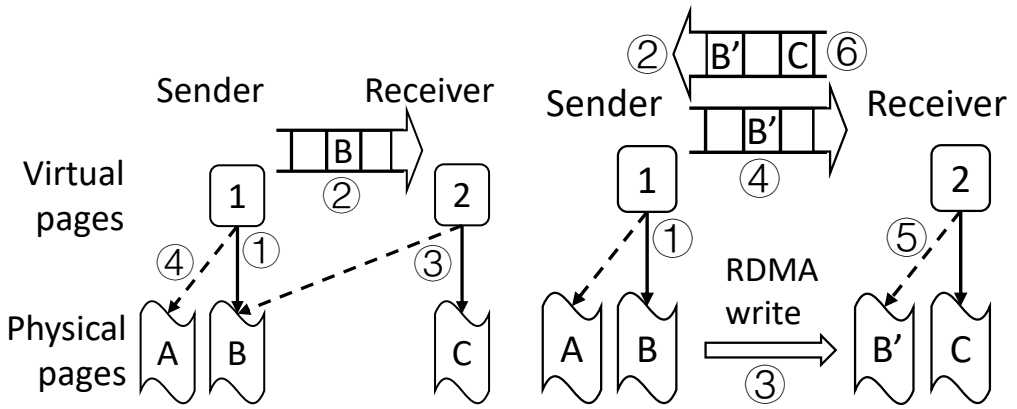
内存对齐。页面重新映射仅在发送和接收地址页面对齐且传输包含整个页面时才有效。libsd 拦截 malloc 和 realloc 函数，并为大小超过 16 KiB 的内存分配操作分配 4 KiB 对齐的地址，因此大多数缓冲区将与页面边界对齐，而较小的内存分配按照原有方法分配，避免了内部碎片。如果发送消息的大小不是 4 KiB 的倍数，则 send 和 recv 时将复制最后一块不是整页的数据。

有时，应用程序需要并不从分配的缓冲区起始地址开始接收或发送数据。例如，数据为 HTTP 请求的一部分，HTTP 请求的内存是对齐的，而由于 HTTP 头的存在，数据就不是对齐的了。对于非对齐情况，如果应用程序在接收之后直接发送，而没有读写数据本身，SocksDirect 也可以实现零拷贝的消息传输。SocksDirect 的方法是对于非对齐的接收缓冲区，默认不执行内存拷贝，而是记录页面的映射和偏移量关系。在应用程序首次访问时通过页面异常触发数据拷贝；如果应用程序不访问数据，则无需拷贝。

减少写时复制。当发送方在 send 之后覆盖缓冲区时，现有设计使用写时复制（copy-on-write）。复制是必需的，因为发送方可能会读取页面的未写入部分。由于应用程序几乎总是将缓冲区重用于后续发送操作，因此在大多数情况下会调用写时复制，这使得零拷贝对发送方基本无用。本文观察到，大多数应用程序不会逐字节写入发送缓冲区。相反，它们会通过 recv 或 memcpy 覆盖发送缓冲区的整个页面，因此无需复制页面的原始数据。对于 memcpy，libsd 调用内核重新映射新页面并禁用写时复制，然后执行实际复制。对于 recv，旧页面映射将被接收的页面替换。

页面分配开销。页面重新映射机制需要内核为每个零拷贝 send 和 recv 分配和释放内存页面。内核中的页面分配使用全局锁，这是低效的。因此，libsd 在本地管理每个进程中的可用页面池。libsd 还跟踪收到的零拷贝页面的来源。当页面未映射时，如果它来自另一个进程，libsd 会通过消息将页面返回给所有者。

通过共享内存安全发送页面地址。对于主机内部套接字，libsd 在用户态队列



(a) 服务器内共享内存。1）获取物理页面并设置写时复制; 2）通过共享内存发送页面地址; 3）映射收到的页面; 4）（可选）在发送方写入 / memcpy / recv 时重新映射。

(b) 服务器间 RDMA。1）获取物理页面并设置写时复制; 2）从页面池中获取可用页面; 3）通过 RDMA 发送数据; 4）通过 RDMA 发送页面地址; 5）映射收到的页面; 6）将未映射的页面返回到页面池。

图 6.9　发送零拷贝页面的过程。

的消息中发送物理页面地址，如图 6.9a 中的步骤 2 所示。为了安全，SocksDirect 必须防止未经发送端允许的任意页面映射。为此，libsd 调用修改后的网卡驱动程序获取发送缓冲区经过加密处理的物理页面地址，并通过共享内存队列将加密地址发送给接收方。在接收端，libsd 调用内核将加密的物理页面地址重新映射到应用程序提供的接收缓冲区虚拟地址。

RDMA 下的零拷贝。 libsd 初始化接收器上的固定页面池，并将页面的物理地址发送给发送方。页面池由发送方管理。在发送方上，libsd 将发送缓冲区的虚拟与物理页面映射固定（pin），然后从远程接收器页面池中分配页面以确定 RDMA 写入的远程地址，如图 6.9b 中的步骤 2 所示。在接收器上，当调用 `recv` 时，libsd 调用网卡驱动程序将池中的页面映射到应用程序缓冲区虚拟地址。在重新映射的页面被释放后（例如被另一个 `recv` 覆盖），libsd 将它们返回给发送方中的页面池管理器（步骤 6）。

6.4.4　事件通知

挑战 1：在内核和 libsd 之间复用事件。应用程序轮询来自 Linux 内核处理的套接字和其他内核文件描述符的事件。轮询内核事件的一种简单方法是每次都调用系统调用（例如 `epoll_wait`），这会产生很高的开销，因为事件轮询几乎是每次发送和接收都需调用的频繁操作。LOS [25] 定期用内核文件描述符调用非阻塞 `epoll_wait` 系统调用，这导致延迟和 CPU 开销之间的权衡：如果调用太频繁，CPU 开销较高；如果调用不够频繁，则内核事件被通知到应用程序的平均延迟较高。不同的是，libsd 在每个进程中创建了一个 *epoll 线程*，它调用

`epoll_wait` 系统调用来轮询内核事件。每当 epoll 线程收到内核事件时，应用程序线程将报告该事件以及用户空间套接字事件。

挑战 2：中断繁忙的进程。套接字接管机制（第 1 节）需要进程响应管程请求。但是，进程可能长时间执行应用程序代码，而没有调用 libsd，管程请求也就无法被响应。为了解决这个问题，本章设计了一个可以类比操作系统中断的*信号*（signal）机制。管程在发出请求后，首先轮询接收队列一段时间，如果超时没有回复，它会向相应进程发送 Linux *信号*并唤醒进程。

由 libsd 注册的信号处理程序首先确定进程是执行应用程序还是 libsd 代码。libsd 在库的入口和出口处设置和清除标志。如果信号处理程序发现进程在 libsd 中，它什么也不做，libsd 将在将控制权返回给应用程序之前处理该事件。否则，信号处理程序会立即处理管程的消息。因为 libsd 被设计为快速且无阻塞（可能导致阻塞的系统调用都在 epoll 线程中调用），所以管程在发送信号后很快就会收到响应。

挑战 3：让多个线程分时核心。对于阻塞套接字操作（例如，阻塞 recv，connect 和 epoll_wait），libsd 首先轮询环形缓冲区一次。如果操作没有完成，libsd 调用 *sched_yield* 以放弃 CPU，切换到同一核心上的其他进程。如章节 6.2.2 中所述，协同多任务处理中的上下文切换仅需要 0.4 μs。但是，一些应用程序可能需要等待很长时间才能收到一个套接字消息，从而导致频繁的唤醒浪费。为此，libsd 对连续的不处理任何消息的唤醒进行计数，并在达到阈值时将进程置于休眠状态。如果 libsd 空转的次数达到一定阈值，它将使自己进入休眠状态。在休眠之前，它会向管程和所有与之直接通信的进程（peers）发送消息，以便稍后通过信号将其唤醒。

6.4.5　连接管理

1. 文件描述符重映射表

套接字文件描述符和其他文件描述符（例如磁盘文件）共享命名空间，Linux 总是分配最小的可用文件描述符。为了保留这种语义而不在内核中分配虚拟文件描述符，libsd 拦截所有与文件描述符相关的 Linux API 并维护*文件描述符重映射表*以将每个应用文件描述符映射到用户空间套接字对象或内核文件描述符。当文件描述符关闭时，libsd 将其放入*文件描述符回收池*。在文件描述符分配时，libsd 首先尝试从池中获取文件描述符。如果池为空，则通过递增*文件描述符分配计数器*来分配新的文件描述符。文件描述符回收池和分配计数器在进程中的所有线程之间共享。

2. 连接建立

图 6.10 显示了连接建立过程。

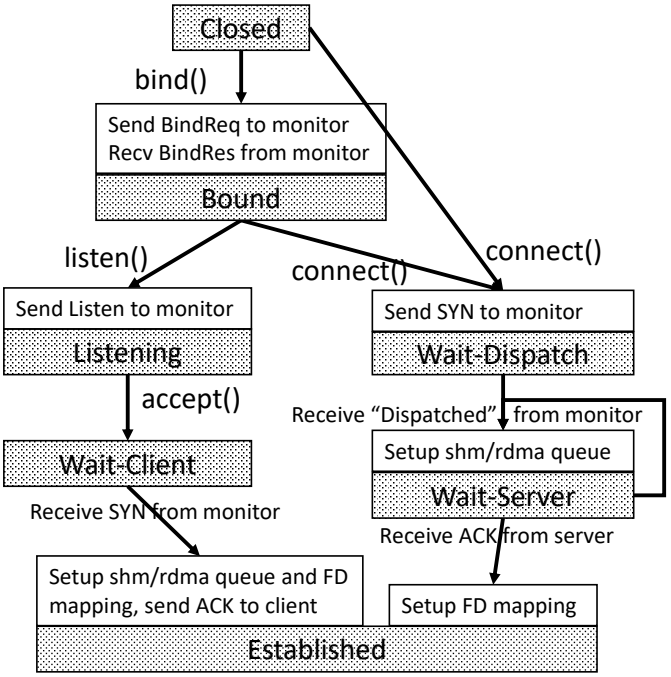


图 6.10　libsd 中连接建立过程的状态机。

绑定地址（bind）。创建套接字后，应用程序调用 `bind` 来分配地址和端口。由于地址和端口是具有权限保护的全局资源，分配由管程协调。如图 6.10 所示，libsd 将请求发送到管程。为了隐藏与管程通信的延迟，作为一个优化，如果绑定请求不可能失败（例如没有为客户端套接字指定端口时），libsd 会立即返回应用程序。

监听端口（listen）。当服务器应用程序准备好接受来自客户端的连接时，它会调用 `listen` 并通知管程。管程维护每个地址和端口上的监听进程列表，以分派新连接。

发起连接（connect）。客户端应用程序调用 `connect` 并发送 SYN 命令以通过共享内存队列进行监听。现在，管程需要将新连接分派给监听应用程序。在 Linux 中，新的连接请求在内核的积压（backlog）中排队。每次服务器应用程序调用 `accept` 时，它都会访问内核以从积压中出列，这需要同步并增加延迟。为解决此问题，SocksDirect 为每个监听套接字的线程维护一个每个监听者的积压。管程以循环方式将 SYN 分发给监听者线程。

当监听者不接受新连接时，分发到该监听者的连接可能会导致饥饿（starvation）。SocksDirect 使用 *工作窃取*（work stealing）[①] 方法。当监听者在积压为空时调用 `accept` 时，它会请求管程窃取其他人的积压。为了避免监听者和管程之间的争用（race condition），管程向侦听器发送请求以从积压中窃取。

建立点对点队列。客户端和服务器应用程序第一次通信时，服务器管程可帮

[①]工作窃取（work stealing）是实现负载均衡的一种常用方法，即每个工作者维护一个请求等待队列，当自己的队列为空时，就从其他工作者的等待队列中窃取请求。

助它们建立直接连接。对于主机内部，管程分配共享内存队列并将共享内存密钥发送到客户端和服务器应用程序。对于主机间，客户端和服务器管程建立新的 RDMA QP，并将本地和远程密钥发送到相应的应用程序。为了减少延迟，当 SYN 命令分发到监听者的积压时，管程会建立对等队列（peer-to-peer queue）。但是，如果 SYN 被另一个监听者窃取，则需要在客户端和新监听者之间建立新队列，如图 6.10 中的 Wait-Server 状态所示。

连接建立的最后步骤。服务器设置对等队列后，如图 6.10 左侧所示，服务器应用程序向客户端发送 ACK。ACK 包含 SYN 请求中的客户端文件描述符及其分配的服务器文件描述符。与 TCP 握手类似，服务器应用程序可以在发送 ACK 后将数据发送到队列。当客户端收到 ACK 时，如图 6.10 右侧所示，它设置文件描述符映射并可以开始发送数据。

3. 与常规 TCP/IP 对端的兼容性

为了与不支持 SocksDirect 和 RDMA 的对端兼容，SocksDirect 需要透明地检测 SocksDirect 功能，并在对端不支持时回退到常规 TCP/IP。但是，Linux 普通套接字不支持向 TCP SYN 和 ACK 数据包添加特殊选项。由于中间盒（middlebox）和网络重新排序，使用另一个端口（例如 LibVMA [22] 的做法）也是不可靠的。为此，libsd 首先使用内核原始套接字（raw socket）直接发送带有特殊选项的 SYN 和 ACK 数据包，如果不存在特殊选项，则回退到内核 TCP/IP 套接字。

在客户端，管程通过网络发送带有特殊选项的 TCP SYN 数据包。如果对端具有 SocksDirect 能力，则其管程将接收特殊 SYN 并且知道客户端具有 SocksDirect 能力。然后，服务器使用特殊选项响应 SYN + ACK，包括设置 RDMA 连接的凭据，以便两个管程之后可以通过 RDMA 进行通信。如果客户端或服务器管程发现对端是常规 TCP/IP 主机，它将使用 Linux 的 TCP 连接修复功能[271] 在内核中创建已建立的 TCP 连接。然后管程通过 Unix 域套接字（Unix domain socket）将内核文件描述符发送到应用程序，libsd 可以使用内核文件描述符进行未来的套接字操作。

一个棘手的问题是接收的数据包被同时传递到原始套接字和内核网络协议栈，此时内核将回复 RST 数据包，因为此连接在内核中不存在。为避免此行为，管程会安装 *iptables* 规则来过滤此类出站 RST 数据包。

4. 连接关闭

当应用程序调用 `close` 时，libsd 会从重映射表中删除文件描述符。然而，套接字可能仍然有用，因为文件描述符可以与其他进程共享，并且缓冲区中可能存在未发送的数据。libsd 跟踪每个套接字的引用计数，在 fork 时递增，在 close 时递减。为了确保未发送的数据已经发送到对端，连接关闭过程中需要在对端之间进行握手，类似于 TCP close。因为套接字是双向的，所以 `close` 相当于在发送

和接收方向上分别执行 `shutdown`。如图 6.11，当应用程序关闭连接的一个方向时，它会向对端发送 *shutdown message*（关闭消息）。对端以关闭消息响应。当 libsd 在两个方向上都收到关闭消息时，将删除套接字。

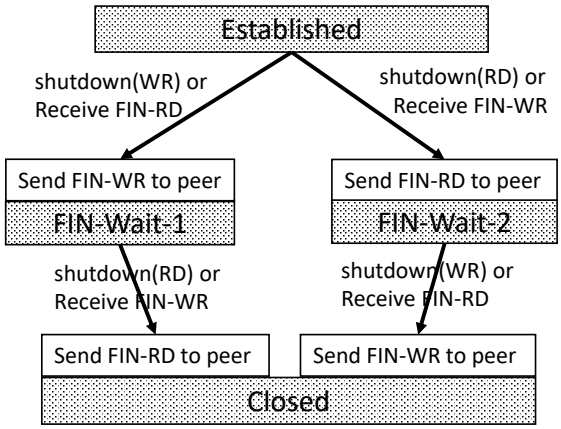


图 6.11　libsd 中连接关闭的状态机。

6.5　系统性能评估

SocksDirect 在三个组件中实现：一个用户空间库 libsd 和一个带有 17K 行 C++ 代码的监控守护进程，以及一个支持零拷贝的内核模块。本节从以下方面评估 SocksDirect：

有效地为主机内套接字使用共享内存。对于 8 字节消息，SocksDirect 实现 0.3 μs RTT 和每秒 23 M 消息的吞吐量。对于大型消息，SocksDirect 使用零拷贝来实现 Linux 的 1/13 延迟和 26x 吞吐量。

有效地使用 RDMA 进行主机间套接字。 SocksDirect 达到 1.7 μs RTT，接近原始 RDMA 性能。零拷贝时，一个连接会使 100 Gbps 链路饱和。

可扩展核心数。随着核心数量的增加，吞吐量几乎可以线性扩展。

使用未经修改的端到端应用程序显着加速。例如，SocksDirect 将 Nginx HTTP 请求延迟减少 5.5 倍到 20 倍。

6.5.1　评估方法

本节使用两个 Xeon E5-2698 v3 CPU，256 GiB 内存和一个 Mellanox ConnectX-4 网卡的服务器评估 SocksDirect。服务器与 Arista 7060CX-32S 交换机通过 100 Gbps 以太网接口互连[272]。不同于第 4、5 章，本节仅使用可编程网卡中的商用网卡部分，且商用网卡升级到了 100 Gbps，没有使用 FPGA。服务器使用 Ubuntu 16.04 和 Linux 4.15，将 RoCEv2 用于 RDMA 协议，每 64 条消息轮

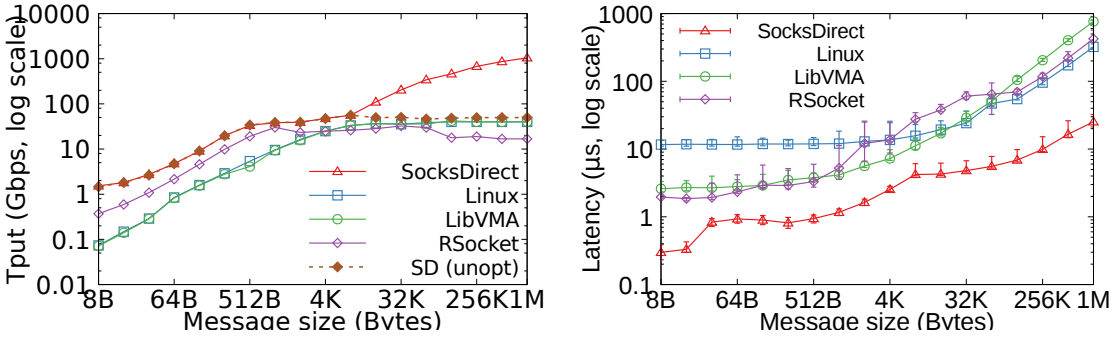
询一次完成队列。每个线程都固定在 CPU 内核上。在收集数据之前，进行了足够的预热测试。对于延迟，本节使用一个乒乓应用程序报告平均往返时间，误差条代表 1 % 和 99 % 百分位数。对于吞吐量，一方保持发送数据而另一方不断接收数据。本节将比较 Linux，原始 RDMA 写原语（write verb），Rsocket [43] 和 LibVMA [22]，这是针对 Mellanox 网卡优化的用户空间 TCP / IP 协议栈。我们也比较了没有批处理和零拷贝的 SocksDirect，用 “SD (unopt)” 表示。本节没有评估 mTCP [17]，因为底层 DPDK 库对 Mellanox ConnectX-4 网卡的支持有限。由于批处理，mTCP 具有比 RDMA 高得多的延迟，报告的吞吐量为每秒 1.7 M 包 [21]。

6.5.2　性能微基准测试

1. 延迟和吞吐量

图 6.12显示了一对发送方和接收方线程之间的主机内套接字性能。对于 8 字节消息，SocksDirect 实现 0.3 μ s 往返延迟（Linux 的 1/35）和每秒 23 M 消息吞吐量（Linux 的 20 倍）。相比之下，一个简单的共享内存队列具有 0.25 μ s 往返延迟和 27 M 吞吐量，表明 SocksDirect 增加了很少的开销。RSocket 具有 6x 延迟和 1/4 吞吐量的 SocksDirect，因为它使用网卡转发主机内数据包，这会导致 PCIe 延迟。LibVMA 只是将内核 TCP 套接字用于主机内部。SocksDirect 的单向延迟为 0.15 μ s，甚至低于内核穿越（0.2 μ s）。基于内核的套接字需要在发送方和接收方都进行内核交叉。

由于内存复制，对于 8 KiB 消息，SocksDirect 的吞吐量仅比 Linux 高 60 %，延迟低 4 倍。对于大小至少为 16 KiB 的消息，SocksDirect 使用页面重映射来实现零拷贝。对于 1 MiB 消息，SocksDirect 比 Linux 具有 1/13 延迟和 26x 吞吐量。由于事件通知延迟，RSocket 的延迟不稳定，在某些情况下甚至可能比 Linux 大。

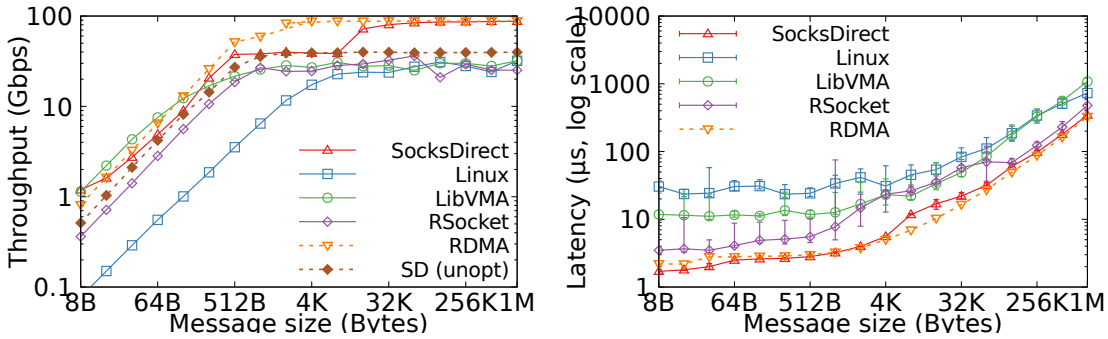


(a) 单机内吞吐量。　　**(b)** 单机内延迟。

图 6.12　不同消息大小下的单机内通信单核消息性能。

图 6.13 显示了一对线程之间的主机间套接字性能。对于 8 字节消息，SocksDirect 实现每秒 18M 消息吞吐量（Linux 的 15 倍）和 1.7 微秒的延迟（Linux 的

1/17）。吞吐量和延迟接近原始 RDMA 写操作（如虚线所示），它没有套接字语义。批处理不影响我们评测的延迟，由于仅当发送队列满时，RDMA 写操作才会被延迟处理，而我们评测延迟仅使用一条消息。由于批处理，SocksDirect 对于 8 字节消息的吞吐量甚至高于 RDMA。非批处理的 SocksDirect 消息吞吐量介于 RSocket 和 RDMA 之间。LibVMA 也使用批处理达到了较好的性能，但延迟是 SocksDirect 的 7 倍。对于小于 8 KiB 的消息大小，主机间 RDMA 的吞吐量略低于主机内共享内存，因为环形缓冲区结构是共享的。对于 512B 到 8KiB 消息，以及更大的没有启用零拷贝的消息，SocksDirect 受数据包复制的限制，但由于缓冲管理开销减少，仍然比 RSocket 和 LibVMA 更快。对于零拷贝消息（≥ 16 KiB），SocksDirect 使网络带宽饱和，其具有所有比较工作的 3.5 倍吞吐量和 RSocket 的 72 % 延迟。



(a) 跨主机吞吐量。　　**(b)** 跨主机延迟。

图 6.13　不同消息大小下的跨主机通信单核消息性能。

2. 延迟分解

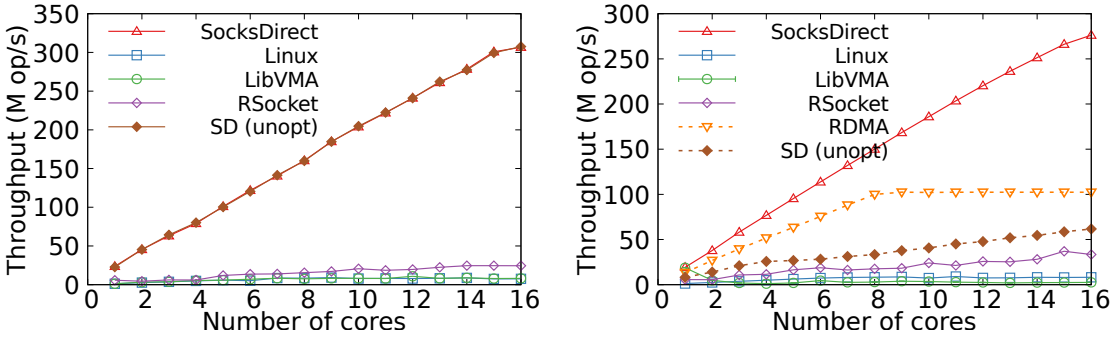
表 6.5 说明了为什么 SocksDirect 的性能超越了其他系统。每次套接字操作，Linux 都需要内核穿越，除 SocksDirect 以外的系统在线程安全模式下都需要加锁。每个数据包，SocksDirect 节约了缓冲区管理开销，并将传输层和数据包处理卸载到网卡。为了传输一个数据包，SocksDirect 利用单边 RDMA 写操作，仅需要在发送端和接收端分别进行一次 DMA 操作。RSocket 使用双边 RDMA，LibVMA 使用一个类似的数据包接口，因此接收端需要增加一次 DMA 操作。LibVMA 和 RSocket 使用网卡来转发单机内的数据包，而 SocksDirect 使用共享内存。Linux 的高延迟主要是由于中断处理和进程唤醒。对于较大的消息，SocksDirect 消除了数据拷贝，页面重映射的开销明显更低。RSocket 比 LibVMA 和 Linux 的性能更好，由于它把发送端的数据拷贝、RDMA 发送操作和接收端的数据拷贝操作流水线化了。SocksDirect 的连接建立延迟主要来源于通过 Linux 裸套接字的初始握手以及通过 `libibverbs` 创建 RDMA QP。

类型	开销	SocksDirect	LibVMA	RSocket	Linux
每操作	总共（线程不安全）	53	56	71	413
每操作	总共（线程安全）	53	177	209	413
每操作	C 库封装	15	10	10	12
每操作	内核穿越（系统调用）	N/A	N/A	N/A	205
每操作	套接字文件描述符锁	N/A	121	138	160
每数据包	总共（主机间）	850	2200	1700	15000
每数据包	总共（主机内）	150	1300	1000	5800
每数据包	缓冲区管理	50	320	370	430
每数据包	传输层协议	N/A	260	N/A	360
每数据包	数据包处理	N/A	200	N/A	500
每数据包	网卡门铃和 DMA	600	900	900	2100
每数据包	网卡处理 & wire	200			
每数据包	处理网卡中断	N/A	N/A	N/A	4000
每数据包	进程唤醒	N/A	N/A	N/A	5000
每千字节	总共（主机间）	173	540	239	365
每千字节	总共（主机内）	13	381	212	160
每千字节	线上传输	160			
每连接	总共（主机间）	47000	18000	77000	47000
每连接	总共（主机内）	700	3800	33000	14700
每连接	初始 TCP 握手	16000	16000	47000	N/A
每连接	管程处理	180	N/A	N/A	N/A
每连接	RDMA QP 创建	30000	N/A	30000	N/A

表 6.5　SocksDirect 和其他系统的延迟分解。每操作延迟使用 `fcntl()` 测量，每数据包和每千字节延迟是从 `send()` 到 `recv()` 的时间，每连接延迟是连接创建的延迟。表中数字单位为纳秒，仅代表粗略估计。

3. 多核可扩放性

SocksDirect 实现了主机内和主机间套接字的几乎线性可扩展性。对于主机内套接字，SocksDirect 在 16 对发送器和接收器核心之间提供每秒 306 M 消息的吞吐量，这是 Linux 的 40 倍和 RSocket 的 30 倍。LibVMA 回退到 Linux 用于主机内套接字。使用 RDMA 作为主机间套接字，SocksDirect 使用批处理以 16 个内核实现每秒 276 M 个消息的吞吐量，这是本章使用的 RDMA 网卡的消息吞吐量的 2.5 倍，也是 RSocket 吞吐量的 8 倍。不启用批处理时，SocksDirect 只能达到 62 M 的吞吐量，是裸 RDMA 的 60%。由于缓冲区管理的可扩展性有限，RSocket 的主机内部吞吐量为 24 M，主机间为 33 M。由于共享网卡队列上的锁争用，与单线程相比，LibVMA 的吞吐量减少到两个线程的 1/4，而三个和更多线程的 1/10。Linux 吞吐量从 1 到 7 个核心线性扩展，并在环回或具有更多核心的网卡队列上出现瓶颈。尽管本文没有测试，mTCP 预计将在多核情况下有更好的可扩放性。



(a) 单机内吞吐量。　　**(b)** 跨主机吞吐量。

图 6.14　不同 CPU 核数下的 8 字节消息吞吐量。

最后评估共享核心的多个线程的性能。每个线程都需要等待轮到它来处理消息。如图 6.15所示，尽管消息处理延迟几乎与活动进程的数量呈线性增长，但它仍然是 Linux 的 1/20 到 1/30。

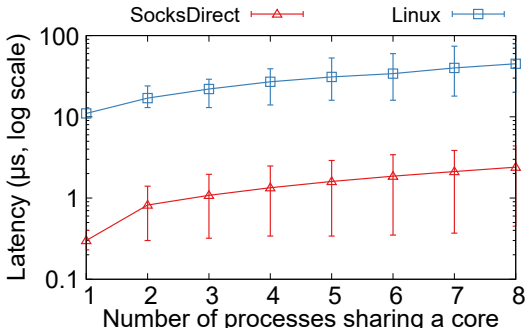


图 6.15　多进程共享 CPU 核的消息处理延迟。

6.5.3　实际应用性能

本节演示了 SocksDirect 可以在不修改代码的情况下显着提高实际应用程序的性能。Rsocket [43] 与以下任何应用程序都不兼容。

1. Nginx HTTP 服务器

为了测试客户端来自网络并在主机内提供服务的典型 Web 服务方案，本节使用 Nginx [237] v1.10 作为 HTTP 请求生成器和 HTTP 响应生成器之间的反向代理。Nginx 和响应生成器位于同一主机中，而请求生成器位于不同的主机中。生成器使用保持活动的 TCP 连接与 Nginx 进行通信。由于 fork，LibVMA [22] 不能与未修改的 Nginx 一起使用。在图 6.16中，请求生成器测量从发送 HTTP 请求到接收整个响应的时间。对于较小的 HTTP 响应大小，与 Linux 相比，SocksDirect 可将延迟减少 5.5 倍。对于大型响应，由于零拷贝，SocksDirect 可将延迟降低多达 20 倍。

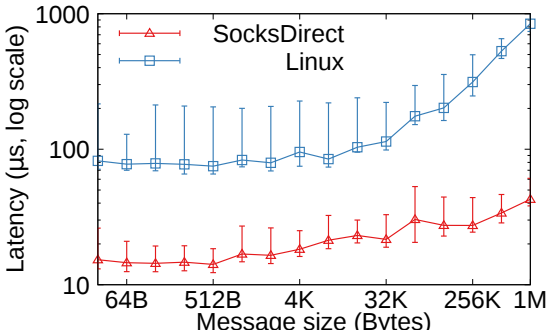


图 6.16　Nginx HTTP 请求端到端延迟。

2. Redis 键值存储

本节使用 redis-benchmark 客户端和 8 字节 GET 请求来测量 Redis [78] 内存键值存储服务器的延迟。使用 Linux 时，平均延迟为 38.9 μs，1 % 和 99 % 百分位延迟分别为 31.6 和 56.1 μs。使用 SocksDirect 后，平均延迟为 14.1 μs（比 Linux 低 64 %），1 % 和 99 % 百分位数 8.4 和 19.1 μs。

3. 远程过程调用（RPC）库

本节使用 RPClib [273] 来测量 RPC 延迟。在主机内的两个进程中运行 RPClib 中的示例 1 KiB RPC，需要 45 μs。在两个主机上，RPC 需要 79 μs。使用 SocksDirect，主机内延迟变为 21 μs（减少 53 %），主机间为 46 μs （减少 42 %）。

然而，SocksDirect 不是万能药。即使使用了 libsd, RPClib 的性能仍然远远低于最先进的 RPC 库，如 eRPC[21]，由于 RPClib 的开销成为性能瓶颈。

4. 网络功能流水线

pcap 格式的 64 字节数据包来自外部数据包生成器，通过网络功能（NF）管道，并发送回数据包生成器。本节将每个 NF 实现为一个进程，它从 *stdin* 输入数

据包，更新本地计数器，并输出到 *stdout*。对于 Linux，使用 *pipe* 和 *TCP socket* 来连接主机内的 NF 进程。图 6.17表明 SocksDirect 的吞吐量分别是 Linux 管道和 TCP 套接字的 15 倍和 20 倍。它甚至接近最先进的 NF 框架，NetBricks [274]。

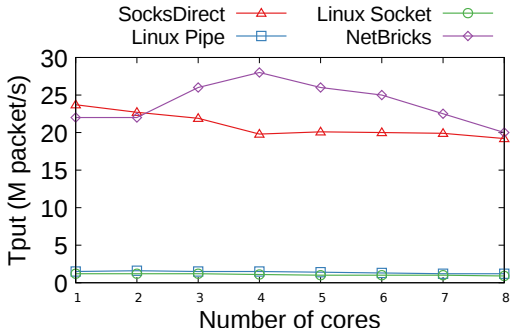


图 6.17　网络功能流水线的吞吐量。

6.6　讨论：连接数可扩放性

使用商用 RDMA 网卡时，SocksDirect 对大量连接的可伸缩性受底层传输层（即共享内存和 RDMA）的限制。为了表明 libsd 和管程不是瓶颈，本节在两个重用 RDMA QP 和共享内存的进程之间创建了很多连接。使用 libsd 的应用程序线程每秒可以创建 1.4 M 个新连接，这是 Linux 的 20 倍和 mTCP 的 2 倍[17]。管程每秒可以创建 5.3 M 个连接。

由于主机内的进程数量有限，因此共享内存连接的数量可能不会很大。但是，一台主机可能连接到许多其他主机，RDMA 的可扩展性成为一个问题。RDMA 的可扩展性归结为两个问题。首先，RDMA 网卡使用网卡内存作为缓存来保持每个连接状态。当有数千个并发连接时，性能受到频繁缓存未命中的影响[21,247,275]。因为 RDMA 传统上部署在中小型集群中，传统 RDMA 网卡上的内存容量较小。随着近年来大规模的 RDMA 部署[41]，大多数网卡厂商已经意识到这个问题。因此，近期的商用网卡内存容量越来越大，例如 Mellanox ConnectX-5[?]，可以存储上千条连接的状态[21]。而本文使用的可编程网卡甚至拥有数千兆字节的 DRAM [10,132,276]。因此，本文预测未来的数据中心不会为网卡缓存未命中问题担心过多。下一节将基于可编程网卡，提出一个连接数可扩放的传输层实现框架。第二个问题是本文的测试平台中建立 RDMA 连接需要大约 $30\mu s$，这对于短连接很重要。此过程仅涉及本地 CPU 和网卡之间的通信，因此这个连接建立延迟是可以优化的。

在大量并发连接下的服务质量保证（Quality of Service，QoS）也是数据中心的重要需求。传统网络协议栈在操作系统内核实现服务质量保证。而对于本章使用的硬件传输协议，将数据平面性能隔离和拥塞控制卸载到 RDMA 网卡

上是一个越来越流行的研究方向[12,243,247,270,277]，因为数据中心的网卡正变得越来越可编程[10,37,122,132,276]，而且公有云已经在虚拟机之外的网络功能中提供了 QoS[156,274,278]。

连接数可扩放性需要存储每个连接的传输层和数据包缓冲区。针对传输层状态问题，接下来的两节提出两种方案：在可编程网卡中或主机 CPU 的用户态库中存储连接状态并实现传输层处理。针对数据包缓冲区问题，最后一节提出多套接字共享队列，将两个进程间多个连接的缓冲区合并。

6.6.1　基于可编程网卡的传输层

本节基于可编程网卡和第 6 章的网络数据包处理平台和第 5 章的内存键值存储，实现了一个连接数可扩放的 RDMA 网卡。实现连接数可扩放的主要挑战是在缓存不命中率很高时，高吞吐量地存取主机内存中的连接状态，并能隐藏存取延迟。这正是第 5 章所解决的问题，因此本节基于高性能键值存储实现了可扩放的连接状态存储。

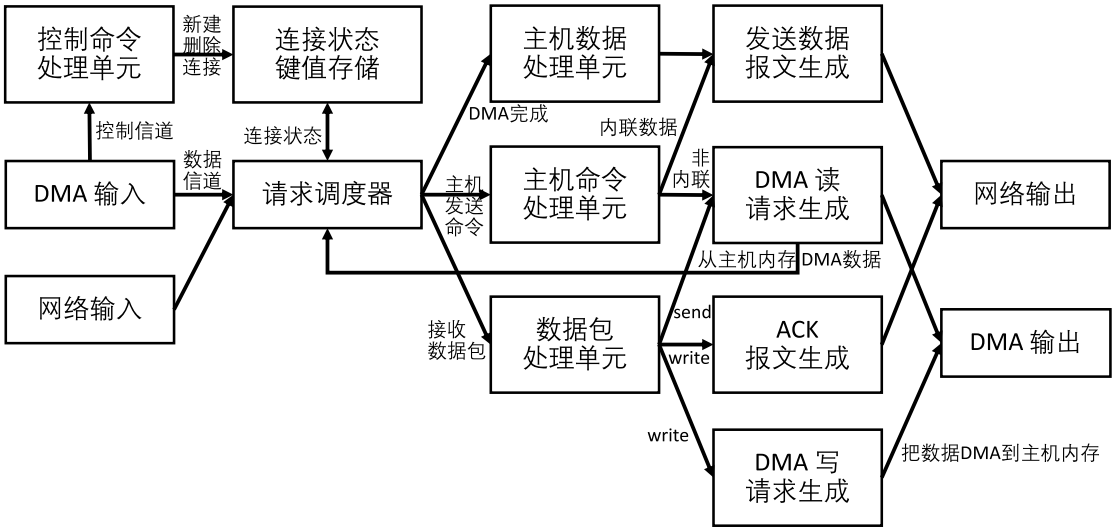


图 6.18　基于可编程网卡的连接数可扩放 RDMA。

基于可编程网卡的 RDMA 网卡架构如图 6.18 所示。RDMA 网卡需要处理来自主机的控制和数据传输命令（工作请求，work request），还需要处理来自网络的数据包。对于控制和数据传输命令，主机 CPU 把工作请求放入主机内存中的工作队列（work queue），再通过 PCIe DMA 发送到网卡[47]。通过网络接口接收的数据包也被放入网卡的输入缓冲区，对应工作队列中的一个工作请求。请求调度器根据数据包的五元组（five-tuple）信息或主机命令中的连接编号，从连接状态键值存储中取出连接的状态信息，并按照连接的优先级，将工作请求和当前连接状态分类放入网卡内部的不同工作队列。由于 RDMA 消息的处理是有状态的，处理同一个连接的两个相邻数据包可能存在依赖关系。为此，请求调度

器记录正在处理的连接，并只调度未被处理连接的工作请求，这与第 5 章键值存储中同一个键的消息处理方式相同。对于接收到的数据包，接收处理单元按照 RDMA 消息的类型处理。对于 RDMA 单边写（write）消息，只需生成主机 DMA 操作，将数据写入主机内存的相应位置，并回复 ACK 消息。对于 RDMA 单边读（read）、原子（atomic）和双边发送（send）消息，需要从主机内存中读取相应的数据，才能进行下一步操作。为了隐藏主机内存读取的 DMA 延迟，生成主机内存的 DMA 读请求后，还需要生成一个新的工作请求，等待 DMA 完成后再进行下一步处理。这种新工作请求被送回请求调度器，同时标记等待条件。当 DMA 完成后，请求调度器将处理这个新工作请求，发送数据到网络或将数据 DMA 到主机内存。数据发送命令 send 的处理方式与 RDMA 单边读（read）消息的处理方式类似。数据接收命令 recv 不需要网卡主动处理，而是从网络收到双边发送（send）或带立即数的单边写（write with immediate）消息时，才需要匹配对应的 recv 工作请求。

上述处理流程的性能挑战是单个时钟周期内很难完成一个工作请求的有状态处理（如拥塞控制），而同一个连接的工作请求不能并行处理，从而降低了单连接吞吐量。解决方案是将工作请求的处理流水线（pipeline）化，每个流水级（stage）处理连接状态的不同部分，因此流水级之间没有数据依赖。在每个流水级内设置第 5 章的数据转发（data forwarding）机制，使尚未写回请求调度器的状态更新对后续的工作请求可见。这样，同一个连接有依赖关系的多个工作请求可以在不同的流水级间并发处理。对于这类可以通过流水线和数据转发解决的依赖，请求调度器就不必记录依赖关系，而是认为所有此类请求都是不相关的。

6.6.2　基于 CPU 的传输层

实现连接数可扩放性的另一种方案是在主机 CPU 上实现传输层协议，从而网卡无需为每个连接存储状态，只需实现无状态卸载。网卡无状态卸载的常见方案是用户态协议栈与网卡之间使用收发数据包接口，而非 RDMA 远程内存访问接口。基于数据包的实现除了可以处理大量并发连接，还可以在不支持 RDMA 的虚拟化平台上使用。例如，微软 Azure 云的很多虚拟机实例不支持 RDMA，但支持 DPDK 和 LibVMA 等高性能数据包接口，基于数据包的传输层将可以在这些虚拟化平台上使用。

LibVMA 与网卡之间使用高性能数据包收发接口。LibVMA 的兼容性、性能和多核可扩放性问题主要是由于其 VFS 层。因此，本节利用 LibVMA 实现传输层功能和网卡接口，替换 libsd 中的环形缓冲区和 RDMA 硬件传输层。LibVMA[22] 用户态套接字库的结构与图 6.5 的 libsd 库类似，都是由 API 封装、VFS 层、队列层和传输层构成。LibVMA 的队列层和传输层由 LwIP 轻量级 TCP/IP 协议栈

和 Mellnox 网卡的高速数据包收发接口组成。为了使用 LibVMA，在 libsd 中将基于环形缓冲区的队列替换成 LwIP 的发送接收接口。测试表明，LibVMA 中的 LwIP 和网卡接口部分发送和接收小数据包的吞吐量为 18 M 次每秒；libsd 中的 API 封装和 VFS 层的吞吐量为 27 M 次每秒。这意味着基于 LibVMA 的 libsd 吞吐量大约可达 10.8M 个小数据包每秒。

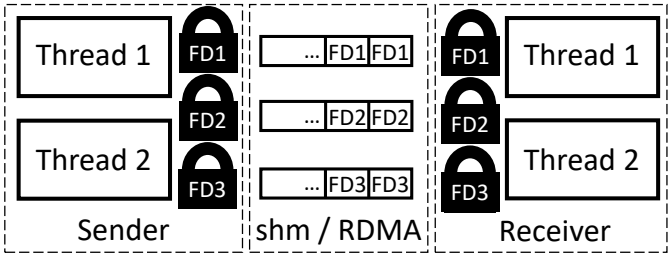
为了实现基于 TCP/IP 的零拷贝，LibVMA 中的 LwIP 传输层需要修改。为了页面重映射，发送和接收的有效载荷需要对齐到 4 KiB 边界。发送时，libsd 组建一个两块缓冲区构成的数据包：首先是从数据包头模板经过 LwIP 传输层组建的数据包头部，然后是零拷贝的有效载荷。libsd 利用网卡的分散-聚集（scatter-gather）支持来让网卡把两块缓冲区组成一个数据包。接收时，libsd 使用一个包含两块缓冲区的接收工作请求，首先是恰好能容纳标准 TCP/IP 数据包头的 54 字节缓冲区，然后是页面对齐的有效载荷缓冲区。与第 6.4.2 章的设计相同，libsd 在接收到数据包后及时补充 `recv` 工作请求，保持网卡始终有接收缓冲区可用。

上述基于数据包接口的方案需要 LibVMA 库为每条连接插入一条流重定向（flow steering）规则，把接收到的数据包映射到接收工作队列。这仍然需要网卡为每条连接维护状态，因此如第 6.5 节的评估结果显示的，并发连接数较多时性能仍然会下降。为了让网卡完全不保存连接状态，可以用可编程网卡实现不可靠、无拥塞控制的单边 RDMA 写操作（目前 Mellanox RDMA 网卡不支持基于不可靠数据报的单边 RDMA）。单边 RDMA 写操作中包含远程主机上的内存地址，因此接收端网卡只需将有效载荷通过 PCIe DMA 写入数据包中指定的地址。这样就可以应用本章第 6.4.2 节的环形缓冲区设计，发送端通过不可靠信道将环形缓冲区的变化同步到接收端。启用了 RDMA 的数据中心网络丢包率很低，因此可以通过超时检测丢包。具体地，接收端发现环形缓冲区内有数据后即发送确认（ACK）包；发送端如果超时未收到确认包则重传。为了实现基于窗口的拥塞控制，发送端需要为每个环形缓冲区（即每条连接）维护一个发送窗口，并在收到确认包和显式拥塞通知（ECN）时调整发送窗口。丢包恢复和拥塞控制等传输层功能增加的 CPU 开销是有限的。通过这种方法可以解决网卡连接数的可扩放性问题。

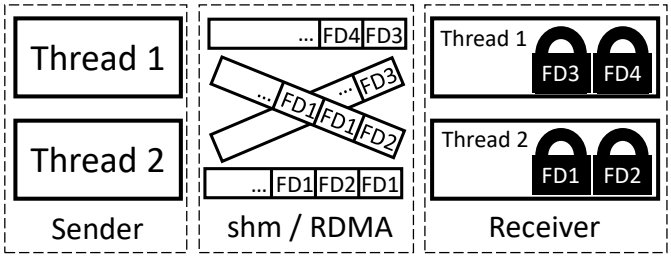
6.6.3　多套接字共享队列

很多应用在两个进程间建立多个套接字连接。例如，数据库的多个客户端线程和多个服务器线程之间可能两两建立连接。HTTP 负载均衡器与后端 Web 服务之间往往为每个 HTTP 请求建立一个连接。一些其他的传输层协议（如 SCTP）和应用层协议（如 QUIC 和很多 RPC 库）也提供多连接的抽象。在传统设计中，每个连接需要独立的缓冲区，因而所需的缓冲区数量较多。

为了降低内存占用，提升内存访问的局部性，如图 6.19，本文使用一条队列来共享一对线程之间的所有连接。队列中的每个数据元素用其文件描述符标识。通过使用一条队列，可以降低每套接字的内存占用、随机内存访问和缓存不命中。



(a) 传统队列结构。



(b) 多套接字共享队列结构。

图 6.19　队列结构的比较。假设发送方和接收方各有两个线程。首先，在每对发送方和接收方线程之间创建对等队列。不是用锁来保护队列，而是将每个文件描述符指定给接收方线程以确保排序。其次，来自所有连接（文件描述符）的数据通过共享队列进行多路复用，而不是每个文件描述符一个队列。

队列中的消息格式。在第 6.4.2 节的传统队列结构基础上，为每条消息在头部增加一个文件描述符域，表示接收端的文件描述符。这样，不同文件描述符的消息就可以共享一个队列。每条消息头部还增加一个 `下一消息指针`域，用于下述的事件轮询；增加一个 `删除`位，用于下述的从队列中间取消息。

事件轮询。为每个 epoll 文件描述符集合维护一个位图。当调用 `epoll_wait` 时，轮流扫描所有的数据队列，并在位图中检查每个数据消息的文件描述符。如果文件描述符在位图中，就返回给应用程序一个事件。维护一个全局指针，以便从上次扫描的队列位置恢复数据队列的扫描。为了避免多次扫描同一个消息，每条队列设置一个指针，保存上次扫描的位置。由于应用程序可能在一个文件描述符上反复执行接收操作，直到该文件描述符的队列为空，本文一边扫描，一边为每个文件描述符创建一个消息链表，以加速重复的接收操作。每个文件描述符维护两个指针，即该文件描述符的第一个和最后一个扫描过但尚未被接收走的消息。当扫描文件描述符的一条新消息时，消息头部的下一消息指针域被更新，指向新扫描的消息，这就组成了同一文件描述符的消息链表。

从队列中间取消息。为了从任意的文件描述符接收数据，队列需要支持从中间取走一条消息。幸运的是，这并不经常发生。事件驱动的应用程序通常按照先到先处理的顺序处理到来的事件。对于电平触发模式的 `epoll_wait` 操作，libsd 在队列中扫描所有消息，并返回那些文件描述符已被注册在 epoll 文件描述符集合中的消息。因此，当应用程序调用 `recv` 时，取走的通常是队列头部的消息。

为了从队列的中间寻找特定文件描述符的消息，如果文件描述符的消息链表非空，则链表头就是要找的消息；如果为空，则需要在队列中遍历消息，从环形缓冲区的头指针遍历到未被分配的空间（用 *有效*位标识）。因此，当从队列中间取走一条消息时，其 *有效*位不能被清空。因此，每条消息增加一个 *删除*位。当消息从中间被取走时，该 *删除*位被设置。

碎片整理。如果应用程序一直不接收某个文件描述符的数据，队列的空闲空间将变得碎片化。当环形缓冲区中没有可用空间时，有可能其中仍有很多已被删除的消息，但因其位于未被接收的其他文件描述符消息之间，这些消息的空间无法被利用。当环形缓冲区没有可用空间时，发送端通过共享内存中的控制寄存器，通知接收端进行碎片整理。接收端扫描环形缓冲区中的可用空间，将尚未被接收的消息集中在一起，并将空闲空间返回给发送端。

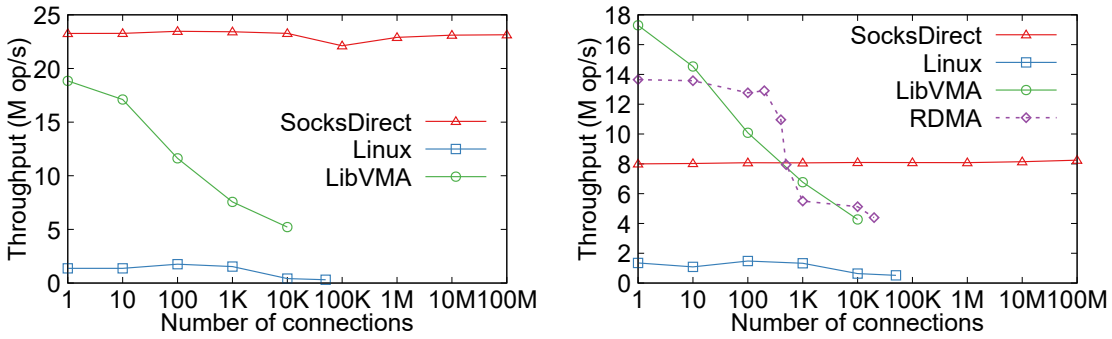
下面评估多套接字共享队列的连接数可扩放性。测试前在两个进程间预先建立指定数量的连接，然后轮流（round-robin）使用这些连接以乒乓（ping-pong）的模式收发数据。图 6.20 显示了不同并发连接数量下的单核吞吐量。SocksDirect 可以用 16 GB 的主机内存支持 100 M 条并发连接，并且在如此高的并发度下吞吐量不降低。作为对比，RDMA、LibVMA 和 Linux 的性能随连接数量的增加而迅速降低。对于 RDMA，性能在超过 512 条并发连接后迅速降低，这是由于 RDMA 传输层状态占满了网卡缓冲区。尽管 LibVMA 和 Linux 并不使用 RDMA 作为传输层，它们为每条连接维护缓冲区，因此在数千条并发连接时会导致 CPU 缓存和 TLB 不命中。此外，LibVMA 在网卡中为每个连接安装了一条连接重定向规则（flow steering rule），这也会导致网卡缓存不命中。

6.7　局限性

除了 §6.6 讨论的高并发下的性能局限，本章将讨论 SocksDirect 在兼容性和 CPU 开销方面的局限性。

6.7.1　兼容性局限

传输层。 SocksDirect 把传输层机制卸载到 RDMA 网卡。读者可能对 RDMA 网卡的传输层机制有一些疑问。例如，大多数商用网卡依赖于基于优先级的流控



(a) 单机内吞吐量。　　**(b)** 跨主机吞吐量。

图 6.20　不同并发连接数量下的单核吞吐量。

（PFC）来消除以太网上由于拥塞导致的丢包。PFC 会带来很多问题，诸如线头阻塞，拥塞扩散，甚至死锁[41]，使网络难以管理和理解。我们注意到很多旨在提高 RDMA 传输层性能的工作。近年来提出的 RDMA 拥塞控制算法[243-244,247?]不仅改进了吞吐量和延迟，还减少了 PFC 暂停（pause）帧的数量。很多高级丢包恢复机制[270,277]也使 RDMA 在有丢包的网络上不再需要 PFC。因此，我们预期未来的 RDMA 网卡可以在有丢包的数据中心网络上提供低延迟和高吞吐量的传输层。

优先级与服务质量保证。多个线程共享同一 CPU 核心时，SocksDirect 使用非抢占调度。但是，为了保证实时性和性能隔离，数据中心内不同优先级的任务通常安排在不同 CPU 核上处理。运行在同一 CPU 核上的进程一般处理同类的工作任务，现有软件（如 Nginx 负载均衡器、Memcached 键值存储等）的各工作进程通常按照先到先服务的顺序处理请求，并没有设置进程优先级。

其他用户态协议栈也存在的兼容性局限。首先，与其他用户态协议栈相同，libsd 使用 LD_PRELOAD 拦截应用程序的 glibc API，不能截获直接的系统调用，从而静态链接的应用程序不能使用。第二，SocksDirect 创建的套接字在 `/proc` 文件系统中不可见，从而一些网络监控工具不能工作。第三，SocksDirect 缺少一些内核协议栈的功能，例如 `netfilter` 和流量控制。然而，现代数据中心网卡已经支持 QoS 和 ACL 卸载 [108]，因此这些功能可以被卸载到硬件。

6.7.2　CPU 开销

SocksDirect 消除了很多现有协议栈中的开销，但引入了一些新的开销。

管程轮询开销。管程的轮询占用了一个 CPU 核。如果在内核中实现管程，通过系统调用来访问管程功能，将消除轮询开销，但又会增加内核穿越（系统调用）的开销和内核的多核同步开销。由于大多数控制平面的操作不需要经过管程，在内核中实现管程增加的每操作开销将是可接受的，但可以节约一个 CPU

核的固定开销。

空闲进程轮询开销。 libsd 的协作式非抢占调度有两个缺陷。首先，如果较多进程共享一个 CPU 核，而事件的到来是相对随机的，上述轮询方法会唤醒大量没有待处理事件的进程，造成延迟增加。为此，需要让内核的协作式调度变得更“智能”，根据到来的请求来调度有任务可做的进程，同时又不重新引入原有抢占式调度的一系列开销。核心方法是根据消息来调整内核的调度队列。考虑两种情况：第一，单机内，一个分派进程发送消息到多个运行在同一 CPU 核上的工作进程。这是一种常见的通信模式，例如任务分派器将任务分发到不同客户的网络功能进程，或者消息源把事件分发给多个订阅者进程。由分派进程管理工作进程的调度顺序。操作系统内核把同一个 CPU 核上运行的工作进程组织成一个进程组，用位图（bitmap）表示，并将其映射到分派进程的用户态地址空间。分派进程在向共享内存队列写入数据后，设置位图中工作进程对应的位。修改内核调度器，不再依次调度所有就绪状态的进程，而是扫描位图并调度下一个被置位的进程。为了防止同一 CPU 核上的其他进程饿死，将工作进程组作为一个持续处于就绪状态且绑定了 CPU 核心的传统进程。由于非工作进程通常处于非就绪（阻塞）状态，不会浪费 CPU 时间调度它们。

第二种情况是跨机器的通信。此时网卡作为中心分派器，支持的通信模式是任意的，不局限于一个分派进程和若干工作进程。网卡的事件队列（event queue）提供了操作系统内核的调度顺序。管程为每个 CPU 核建立一个事件队列，汇总了该 CPU 核上各进程的所有 RDMA 连接的完成队列事件，由网卡写入，操作系统内核读出。内核按照事件队列的顺序调度进程，从而被调度的进程恰好是有事件可处理的。此外，可编程网卡可以观察到每个 CPU 核上事件队列的长度，从而在一个 RDMA 消息可以分发给多个 CPU 核之任一的情况下，可以将消息分发到事件队列最短的 CPU 核，实现更好的负载均衡。

6.8　未来工作

6.8.1　应用、协议栈与网卡间的接口抽象

本章中，应用程序与用户态协议栈之间使用套接字接口通信，协议栈与网卡间使用 RDMA 接口通信。这是为了兼容现有的应用程序和 RDMA 网卡。然而，应用程序在套接字层上往往还有更高层次的通信抽象，如第 5 章的键值存储原语，以及远程过程调用（RPC）、消息队列等原语。随着可编程网卡的出现，主机 CPU 与网卡间任务划分的界限也未必遵循 RDMA 接口。因此，应用程序、协议栈与网卡间的接口抽象可以通盘考虑。

应用、协议栈与网卡之间的接口抽象不仅需要考虑性能问题，还要考虑是否

容易编程。如果仅从性能的角度考虑，对于现有内存容量较小的网卡，较好的任务划分是在网卡上实现高带宽和低延迟连接的传输层，在主机 CPU 上实现大量其他连接的传输层。然而这需要开发者指定哪些连接是需要高带宽和低延迟的，增加了编程负担；或者由协议栈和网卡自动划分和迁移，这也将增加系统的复杂度。

协议栈与应用之间如果可以不遵循套接字接口，将有更大的设计空间，本章 6.2 节介绍了很多相关工作。例如，在零拷贝方面，如果应用程序能给协议栈更多的提示信息，将可以避免很多不必要的内存拷贝。当应用程序调用 `send` 后，可能继续读写发送缓冲区。为了保证应用程序能读到缓冲区的内容，零拷贝的页面必须设置为只读，这使得同一主机内的接收端就地修改接收内容时需要写时复制。当应用程序写入发送缓冲区时，协议栈并不知道该缓冲区未被写入的部分是否会被应用程序读取，因此并不能映射一个空页面，而需要写时复制。本文通过拦截 `memcpy` 来优化整页写入的情况，然而不能优化页面被部分写入的情况。很多应用程序事实上不需要在发送后读取缓冲区内容。解决上述问题的最佳方案是由应用程序告知协议栈，被发送的缓冲区内容是否还需要读取。这可以通过给 `send` 调用增加一个选项，或者额外的 `mem_is_junk` API 来实现。

网卡基于消息的 RDMA 原语与基于字节流的套接字原语存在不匹配。发送端与接收端之间的环形缓冲区在 CPU 的共享内存中不需要软件显式同步。但在基于单边 RDMA 的共享内存中，就需要软件显式发送 RDMA 操作来同步两个缓冲区，即将发送端的数据同步到接收端，并将接收端释放的缓冲区空间同步到发送端。相比硬件实现的一致（coherent）共享内存，软件显式同步增加了 CPU 开销。用硬件实现环形缓冲区同步可以达到更高的吞吐量，特别是在消息很小时。

目前商用 RDMA 网卡与主机 CPU 之间的传输层功能划分不够灵活。如第 6.6.2 节讨论的，Mellanox RDMA 网卡中的单边 RDMA 操作只支持可靠连接（RC）而不支持不可靠数据报（UD），这意味着如果希望在主机 CPU 上实现传输层，就必须使用双边 `send` 和 `recv` 操作或者网卡提供的其他数据包收发接口，而无法使用远程内存访问原语。此外，传输层中的按序传输、拥塞控制、丢包重传等功能也是紧密耦合的，要么全部使用网卡厂商固定的硬件实现，要么全部在 CPU 上用软件实现。可编程网卡提供了解耦传输层功能的机会。

如果网卡具备处理大量并发连接的能力，在网卡中实现连接建立将可以节约 CPU 创建连接过程中的开销和延迟。

6.8.2　模块化网络协议栈

网络协议栈由接口抽象、拥塞控制、丢包恢复、服务质量（QoS）、访问控制列表（ACL）、数据包格式等组件构成，如表 6.6 所示。

表 6.6　网络协议栈的组件选择

组件	选择
接口抽象	RDMA, BSD socket, StackMap, RPC, 消息队列, …
拥塞控制	TCP, DCTCP, DCQCN, TIMELY, MP-RDMA, IRN, …
丢包恢复	Go-back-0, Go-back-N, 选择重传, Cut-Payload, …
QoS	严格优先级，RR，WFQ，多级反馈队列, …
ACL	netfilter, OpenFlow, P4, …
数据包格式	TCP/IP, RDMA, RoCE, RoCEv2, …

目前最具代表性的 RDMA 协议栈和 TCP 协议栈各自实现了一套不同的组件。在数据中心广泛部署的 RoCEv2 协议栈脱胎于 RDMA 协议栈，但替换了数据包格式，以兼容现有数据中心网络基于 IP 地址和端口号的寻址方式。SocksDirect 将两种协议栈的组件进行融合，使用 TCP 协议栈的套接字（socket）接口抽象，但其余组件均使用 RDMA 的对应组件。如第 6.7 节讨论的，SocksDirect 中所使用的 RDMA 拥塞控制、丢包恢复算法与标准 TCP 可能存在公平性问题，也缺少 QoS、ACL 等功能。为此，应当将上述组件模块化，并允许用户灵活地组合。其中，每个组件都可能在 CPU 用户态、CPU 内核态或可编程网卡中实现。一个网络协议栈组合包括用户态、内核态和可编程网卡间的任务划分，以及每种组件的选择。可灵活组合的模块化协议栈可以使用一种模块化的网络功能编程框架实现，如第 4 章的 ClickNP。

如表 6.7 所示，网络协议栈的接口抽象可以进一步细分为多个组件。各组件可以组合出不同的接口抽象，适合不同类型的应用程序，也具有不同的性能特性。SocksDirect 的很多设计旨在实现 Linux socket 的接口抽象，并为实现其中一些抽象付出了性能代价（如为了保证消息全序，连接需要隐式地被一个线程独享；由于用户管理缓冲区，非按页对齐的缓冲区无法使用零拷贝）。我们期待未来的工作提出可灵活组合的模块化网络协议栈接口抽象。

6.9　本章小结

SocksDirect 是 Linux 兼容的高性能用户空间套接字系统。为了控制平面的可信，本文设计了一个每主机的监控守护进程; 一个点对点无同步数据平面，完全支持 fork 和多线程套接字共享; 和一个有效利用共享内存和 RDMA 的环形缓冲区。SocksDirect 实现了接近硬件限制的性能，并提高了实际应用程序的端到端性能。

表 6.7　网络协议栈接口抽象的组件选择

组件	选择
寻址方式	IP 地址 + 端口号，Infiniband 地址，内存地址，基于元数据的节点 ID，…
连接的抽象	字节流，消息流，共享内存，无连接，…
可靠性保证	可靠有序，容忍乱序，容忍丢包，容忍错误，…
连接的共享范围	单线程，线程间，fork 父子进程，容器内所有进程，…
连接共享方式	隐式共享，显式共享，独享并显式传递所有权，…
共享连接消息顺序	全序，因果序，无序，同步屏障，…
缓冲区管理	用户管理（RDMA），协议栈管理（socket），用户分配协议栈释放，协议栈分配用户释放，…
通知方式	阻塞，轮询（select），就绪时通知（epoll），完成后通知（aio），完成队列（RDMA CQ），…

第 7 章　总结与展望

7.1　全文总结

过去几十年，定制化硬件的发展经历过高潮与低谷。十年前，在数据中心的每台服务器上添加一种定制化计算设备无异于天方夜谭。近年来，云计算规模化的趋势、数据中心应用的需求以及通用处理器的性能局限将定制化硬件的发展带上了快车道，数据中心网络的性能也一日千里。

得益于定制化硬件的发展和分布式系统的通信需求，可编程网卡在数据中心被广泛部署：微软用 FPGA 加速搜索引擎、虚拟网络、压缩、机器学习推理等，亚马逊和阿里云加速虚拟网络、虚拟存储和虚拟机监控器，腾讯云用 FPGA、华为云用网络处理器加速虚拟网络……回望历史长河，网络虚拟化也许是可编程网卡的第一个杀手级应用，但这只是可编程网卡潜力的冰山一角。

要使应用程序充分利用数据中心网络的高性能，必须尽量降低 “数据中心税”，这不仅包括网络虚拟化，还包括网络功能和操作系统通信原语。本文提出用基于 FPGA 的可编程网卡加速网络功能。为了简化 FPGA 编程，本文提出首个适用于高速网络数据包处理、基于高级语言的 FPGA 编程框架，相比传统基于 CPU 的网络功能，吞吐量提高了 10 倍，延迟降低到 1/10。为了降低操作系统通信原语的开销，本文提出一个软硬件结合的用户态套接字系统，与现有应用程序完全兼容，并能实现接近硬件极限的吞吐量和延迟，解决了长期以来通用协议栈性能较低、专用协议栈兼容性较差的矛盾。

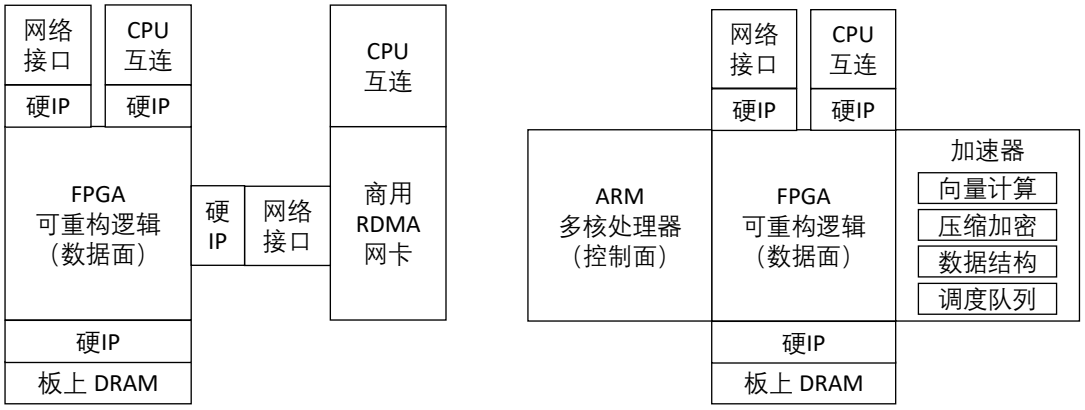
“可编程网卡” 得名于网络加速，但它不会止步于网络，会继续向系统的各个领域深入。内存数据结构存储是分布式系统的重要基础组件。本文提出远程直接键值访问原语，作为远程直接内存访问（RDMA）原语的扩展。通过在服务器端绕过 CPU，用可编程网卡直接访问主机内存，以及一系列性能优化，本文实现了 10 倍于 CPU 键值存储系统的吞吐量和微秒级的延迟，是首个单机性能达到 10 亿次每秒的通用键值存储系统。

毫无疑问，可编程网卡可以提高系统的性能，降低数据中心的成本。本文提出的三个系统为虚拟网络功能、通用内存键值存储和套接字网络协议栈树立了新的性能里程碑。但本文的目的不是打破性能记录，而是启发读者思考：如何建设包括硬件、开发工具链、操作系统在内的可编程网卡生态系统？可编程网卡等新硬件将如何改变数据中心的架构和分布式系统的编程范式？正如有人所说，“预测未来最好的方式就是创造未来”，可编程网卡的故事才刚刚开始。

7.2　未来工作展望

基于可编程网卡的高性能数据中心系统需要软硬件结合的生态系统，主要由硬件、开发工具链和操作系统构成。第 7.2.1 节将展望未来的可编程网卡硬件架构。开发工具链包括编程框架、编译器、运行库、调试工具等，在软硬件协同设计中至关重要。第 7.2.2 节将展望开发工具链方面的几个未来工作。操作系统包括虚拟化、调度、监控、高可用、灵活缩放等。第 7.2.3 节将展望操作系统方面的未来工作。最后，作为数据中心一等公民的可编程网卡在提高系统性能的同时，也使我们重新思考分布式系统的总体架构，可能带来系统创新，这是第 7.2.4 节将讨论的。

7.2.1　基于片上系统的可编程网卡



(a) 本文使用的 Catapult 可编程网卡。　　**(b)** 未来的片上系统。

图 7.1　可编程网卡结构的比较。

本文使用了图 7.1a 所示的 Catapult 可编程网卡。这种架构有三个局限性。首先，现有的商用 RDMA 网卡当并发连接数较多时，性能会急剧下降[247]。我们希望利用第 5 章可扩放键值存储的技术，在 FPGA 可重构逻辑中实现 RDMA 硬件传输协议，实现高并发连接数下的高性能。这已经在第 6.6 节讨论过。其次，FPGA 只适合加速数据面，控制面仍然留在主机 CPU 上。尽管它的计算量不大，但为了性能隔离，计算节点仍然需要预留少量 CPU 核用于控制面处理。第 1 章已经指出，即使预留一个物理 CPU 核也是相当昂贵的。为此，我们希望在可编程网卡中加入 ARM 多核处理器，用于实现控制面，从而完全消除主机 CPU 上的虚拟化开销。ARM 多核处理器的成本为数十美元，远低于一个物理 CPU 核的成本。最后，一些类型的工作负载在 FPGA 内实现的效率不是很高，应当固化在 ASIC 加速器中。第一类是深度学习和机器学习中的向量操作、加密解密操作等计算密集型操作。例如，Intel QuickAssist 加速卡[142] 基于 ASIC 的 RSA 非对称加密比第 4 章基于 FPGA 的实现，吞吐量约高 10 倍；基于 ASIC 的 LZ77 压缩算

法比本文基于 FPGA 的实现，吞吐量也高一个数量级。所用 ASIC 和 FPGA 芯片的功耗、面积和制程都接近。第二类是常见数据结构和调度队列。基于内容寻址内存（Content-Addressable Memory，CAM）的查找表是哈希表、乱序执行引擎、缓存、模糊匹配表等多种常见数据结构的必要组件。CAM 在 ASIC 中可以用三态门实现，而在 FPGA 中实现的效率较低[279]。此外，优先队列（可用移位寄存器序列或堆实现）、轮转（round-robin）调度队列、考虑依赖关系的乱序执行调度器、定时器等结构在很多应用中广泛使用，从而可以借鉴网络处理器（Network Processor）的架构，将这些通用结构硬化，让 FPGA 可重构逻辑专注于定制化计算和灵活互连。

因此，本文期待未来的可编程网卡使用如图 7.1b 所示的片上系统架构。相比使用片外总线互连的分离组件，片上系统可以使组件间通信的带宽更高、延迟更低，更适合将计算细粒度地拆分到更适合的处理组件。位于片上系统中心的 FPGA 不仅提供了可编程性和计算能力，也可以灵活互连和组合片上的各种计算加速器，组建定制化的内存层次结构，还可以灵活互连主机内外的各种硬件设备，组成数据中心智能互连（intelligent fabric）。

目前，业界已有基于片上系统的可编程网卡架构。例如，Xilinx 的 Versal 架构[280-282] 将可重构硬件（FPGA），基于超长指令字（VLIW）的深度学习和传统机器学习加速器、数字信号处理器（DSP）和硬核（hard IP），以及多核通用处理器集成在一块芯片上，组成片上系统（System on Chip）。与传统 FPGA 相比，Versal 架构最大的区别是组成了片上系统，体现在三方面：第一，把内存控制器、PCIe 等外部接口的控制逻辑从可重构逻辑硬化成数字逻辑，减少了 FPGA 面积的开销，还能使 FPGA 实现即插即用。第二，认识到向量操作等大数据、机器学习常见计算在 FPGA 上实现的较低效率，并使用硬化的数字逻辑进行加速。第三，增加了通用处理器，可以处理复杂逻辑和控制平面，而无需绕回 CPU，这使得 Versal 片上系统可以直接驱动 Flash 存储等，组成低成本的存储服务器，而无需传统的 x86 CPU 等组件。片上系统的部件之间通过片上网络互连[282-283]。Versal 架构针对数据中心服务器的多种应用加速，开发者可以把应用分解成通用处理器上的控制面、可重构硬件数据面、向量计算数据面，用合适的体系结构处理应用中的相应部分。

7.2.2　开发工具链

目前，可编程网卡是主要由云计算厂商推动的新兴事物，其生态系统还不够完善。首先，可编程网卡的编译器、调试工具、代码库等开发工具链不够灵活和易用，相关厂商的支持也不够完善。近年来的高层次综合工具主要关注 FPGA 在计算密集型处理（如深度学习）方面的可编程性，而对通信密集型处理的关注

较少。尽管本文第 4 章的 ClickNP 在此方向做了一些努力，但与规模化的商业应用还有较大距离。

其次，目前的研究中，可编程网卡与应用程序间的任务划分是较为随意（ad-hoc）的，需要量化研究方法来确定哪些工作负载适合卸载到可编程网卡。对于一个现有应用程序，要利用可编程网卡加速其数据面功能，需要重写大量代码：不仅需要在可编程网卡内从头实现数据面的处理逻辑，还需要修改主机 CPU 上的控制面代码以充分利用网卡的并行性和隐藏处理延迟。未来的开发工具链需要降低现有应用的二次开发成本。

1. 基于可编程网卡的 PCIe 调试工具

数据中心服务器承载着越来越多的 PCIe 设备，如 GPU、NVMe SSD、网卡、加速卡和 FPGA 等。为了 PCIe 设备之间高吞吐量和低延迟的通信，GPU-Direct、NVMe over Fabrics 等技术开始流行。然而，很多 PCIe 设备只能跟 CPU 上的设备驱动程序通信，其 PCIe 寄存器和 DMA 接口很复杂，且可能没有文档。为了在 PCIe 上抓包和调试 PCIe 协议的实现，开发者往往需要昂贵的物理层 PCIe 协议分析仪（价值 25 万美元左右）。协议分析仪需要实验室环境，难以在生产环境中动态调试。而且，协议分析仪无法修改 PCIe 数据包，也没有足够的可编程性来从大量的流量数据中发现异常或统计规律。

一个未来的方向是基于可编程网卡实现透明 PCIe 传输层协议（TLP）调试器。PCIe 调试器抓取 PCIe 设备和 CPU 之间的通信数据包。这项工作的挑战在于，由于 PCIe 的物理拓扑和路由是固定的，不可能在 PCIe 上实施与局域网中的 ARP 类似的攻击。然而，通过欺骗设备驱动程序，PCIe 流量可以被重定向到 PCIe 调试器。根据请求的发起方，把 PCIe 和 CPU 之间的通信分成两类。

第一类是 CPU 发起的内存映射 I/O（MMIO）操作。这类操作中，CPU 访问 PCIe 基址寄存器（BAR）指向的内存区域。驱动程序从操作系统内核例程中获得 BAR 地址，因此可以修改该操作系统内核例程，返回 PCIe 调试器的地址，而非设备本身的地址。然后在 PCIe 调试器中建立地址映射，使 CPU 的内存映射 I/O 操作传输到 PCIe 调试器，PCIe 调试器作为代理再把请求发送给目标设备。

第二类是设备发起的 DMA 操作，用以访问主机内存。表面上看来，没有办法预知设备会访问哪个内存地址。然而，良定义的设备应当只访问驱动程序分配给该设备的地址。在 Linux 中，有两种设备驱动程序获取可 DMA 内存区域及其物理地址的方法。计划对这两种操作系统例程分别加以修改，在分配 DMA 内存区域时用 PCIe 调试器的地址取代主机内存地址，并建立 PCIe 调试器中的地址映射。这样，当设备试图 DMA 到主机内存时，事实上是 DMA 到了 PCIe 调试器，调试器随后根据映射表把数据再 DMA 到主机内存。

通过这种方法，主机驱动程序和 PCIe 设备的通信都将被 PCIe 调试器截获。

基于 FPGA 的 PCIe 传输层协议调试器有足够的灵活性来修改、统计、过滤和注入数据包，进而实现对 PCIe 设备的模糊测试（fuzz testing）和压力测试。

2. 微秒级延迟隐藏

使用定制化硬件加速应用程序上的 CPU 处理时，原有的 CPU 软件处理逻辑被替换成了向加速器发送命令、等待加速器处理和从加速器接收结果三个步骤。在等待加速器处理期间，CPU 线程被阻塞。类似地，在分布式系统中，经常需要进行远程过程调用（RPC）并等待其他微服务或节点返回结果。传统上，开发者一般采用增加更多线程的方式隐藏加速器处理和远程过程调用延迟，也就是让操作系统在此期间切换到其他线程进行处理。然而，随着数据中心加速器性能的提高和加速任务粒度的降低，一些加速任务的执行时间只有数微秒至数十微秒。类似地，远程过程调用的网络延迟也从之前的数毫秒降低到数微秒至数十微秒。操作系统切换线程调度也需要 3 至 5 微秒，几乎与加速任务的执行时间和远程过程调用的网络延迟相当。这就意味着在等待期间切换到其他线程并不经济，让 CPU 在当前线程上等待加速任务完成可能是更好的做法。但是，这也就意味着等待期间 CPU 时间的浪费，在一定程度上影响了定制化硬件加速器节约 CPU 的效果。

一个未来研究方向是从编译的角度出发，实现应用程序的微秒级延迟隐藏。我们有两个主要的观察：首先，应用程序可能有多个互相不依赖的硬件加速任务需要处理，因此可能挖掘出这些不依赖的加速任务，进行并发处理。其次，很多应用程序是事件驱动的，也就是在一个永久循环里依次处理到来的事件。不同的事件处理之间可能没有依赖关系，这时就可以暂时挂起正在被处理的事件，去处理下一个不相关的事件。

这两种延迟隐藏方案的困难在于“依赖关系”的判断。在函数式编程语言中，纯函数之间的依赖关系比较容易判断。但在大多数开发者通常使用的编程语言中，内存是共享的，很多代码之间都存在依赖。例如，创建对象时需要分配内存，影响内存布局，因此从严格意义上讲，任意两个对象的创建顺序都是有依赖的。再如，两个远程过程调用是否存在依赖，往往取决于其语义。因此，问题的核心挑战是由开发者指明哪些依赖事实上是不必要的。

一种可能的方案是“async”修饰器，允许开发者指定一个函数可以被异步执行。async 函数内部可以使用 wait 调用来注册事件、释放 CPU 并在事件成立时唤醒（例如等待加速器或远程过程调用的返回）。可被异步执行的函数执行过程中不会被打断（除非调用了 wait，或有可被异步执行的子例程），因此不必担心可重入问题。每个 async 函数的执行用协程（coroutine）实现。进一步地，提出“async pure”修饰器，允许开发者指定一个函数不仅可以被异步执行，还没有任何副作用，这样就可以推测执行，即在执行的条件尚未确定时就执行之，而无需

担心其产生不可撤回的副作用。

例如，把无状态计算卸载到加速器、只读的远程过程调用、打开文件是 async pure 函数。而执行写操作的远程过程调用、处理一个事件的例程是一般的 async 函数。如果 async 函数之间存在逻辑依赖关系，例如同一个用户发起的不同事件需要按顺序依次处理，那么可以为每个用户设置一个锁，在事件处理开始时加锁并在结束后解锁。锁使用 wait 调用实现，因此开销很小。

除了从编译的角度挖掘应用程序内部的并行性，另一个未来研究方向是从体系结构的角度出发，实现硬件管理的高性能上下文切换和调度[6]。第 2.3.2 节介绍的网络处理器硬件调度器可以作为有益的借鉴。

3. 网络应用数据面自动生成

为了提升网络应用的性能、降低 CPU 开销，数据中心引入了可编程交换机和网卡以卸载虚拟化网络功能、传输协议、键值存储、分布式一致性协议等。与通用处理器相比，可编程交换机和可编程网卡的资源较少，支持的编程模型也较为受限。为此，开发者通常把一个网络功能分割成处理通常情况数据包的数据面和处理其余情况的控制面。数据面功能在一个数据包处理语言（如 P4）中实现，并卸载到硬件。

为网络应用卸载而编写数据包处理程序需要很多劳动。首先，即使拥有协议说明书或源代码，开发者仍然需要阅读上千页的文档或代码，进而发现哪一部分是常用功能。其次，很多实现与协议说明书之间存在细微的区别，因而开发者经常需要检查数据包的抓包记录，手工反向工程出特定于一个实现的行为。

未来的研究方向是自动学习指定网络应用的行为，从而自动生成数据面参考代码。这样，开发者只需设计一些简单的数据面的测试用例，并运行指定的网络应用。数据面自动生成系统将捕获输入和输出的数据包，并搜索一个数据包程序来对指定的输入测试用例产生测试得到的输出。显然，通过测试用例并不意味着程序能在其他输入的情况下正确地泛化，因此自动生成的代码只能作为开发者的参考，开发者可以在其基础上补充特殊情况处理的细节。尽管如此，自动生成的参考程序可以帮助开发者理解协议在通常情况下的工作方式，节约大量开发时间。

一般意义上，通过例子生成程序被认为是困难的，由于巨大的搜索空间和理论上不可判定的停机问题。幸运的是，可以被卸载到硬件的数据包程序通常是比较简单的。商用可编程交换机和网卡并不支持循环和递归，因此不存在停机问题的判定难题。此外，对于每个持久化状态，每个数据包在数据面上只允许一次读写操作。而且，从数据包输入到输出的逻辑深度被硬件的流水线深度所限制。这些限制极大地降低了程序的搜索空间。更重要的是，为了减小搜索空间，可以主动生成测试用例，以消除一些可能的搜索方向。为了尽可能泛化测试用例，使用

生成测试（generate and test）的方法来观察指定应用的行为。为了在可能的无穷多种可以生成指定输出的程序中选定一种，使用奥卡姆剃刀准则，选择具有最小描述长度的程序。当存在多个描述长度相同的程序时，系统可以生成判定性测试用例来决定正确的那个，或者报告用户。

4. 异构分布式系统的任务划分

数据中心是一个异构硬件组成的分布式系统。每种硬件有一定的计算、存储和网络互连资源。不同硬件能够进行的计算类型和计算效率都不同，例如 CPU 适合控制密集型的计算，GPU 适合一般的单指令流多数据流（SIMD）类型计算，TPU 适合卷积和矩阵乘法类计算，FPGA 适合通信密集型计算。异构的硬件之间通信的能力也不同，例如 GPU 之间可以通过 NVLink 直接通信，而作为可编程网卡的 FPGA 是服务器主机与数据中心网络之间的必经之路。

给定一个计算流图及其中每个元件的高层次语言描述，一个重要的问题是如何把元件映射到异构的计算硬件。显然，仅仅考虑每个元件在各种计算硬件上的执行效率是不够的，还需要考虑元件之间通信的开销。例如，神经网络中卷积层之间的归一化运算可能在 GPU 上执行效率高于 TPU，但 GPU 和 TPU 之间数据搬移的开销可能超过在 TPU 上执行归一化运算的性能损失，因此在 TPU 上融合卷积和归一化运算可能是性能更优的。

一般地，可以将异构计算集群形式化为一张拓扑图，图中的顶点是计算设备、内存和存储设备和网络交换设备，边是节点间的数据通路。每个计算设备有若干支持的计算类型和各类型计算的带宽和延迟，而数据通路的属性包括带宽、延迟。计算流图中的每个顶点表示计算量和计算类型，每条边表示所需传输的数据量。任务划分问题的目标就是找到计算流图到异构计算集群拓扑图的一个映射，使得延迟、吞吐量满足应用的约束。

对于计算规模大的元件，还需要拆分到多个硬件上并行执行或流水线执行。一个元件的计算可能有多种拆分方式，不同拆分方式所需的通信开销不同。需要根据系统性能需求或异构硬件数量的限制计算出每个硬件上所需执行的计算量，然后根据通信和计算开销模型得到优化的元件拆分方案。

7.2.3　操作系统

分布式系统的“操作系统”包括主机上的传统操作系统和分布式系统的调度、管理、监控系统及共享的基础服务中间件。本文研究了操作系统网络协议栈和分布式系统键值存储的优化，但操作系统中还有存储等多个子系统，分布式系统中也有消息队列等多种中间件。这些子系统和中间件显然也可以用可编程网卡加速。

此外，在传统分布式系统中，由于通信成本较高，热迁移和高可用往往需要

开发者使用特定的编程框架。在拥有高性能数据中心网络的数据中心，基于可编程网卡，将可能实现通用应用程序的高效热迁移和高可用。这将让数据中心更像是一台巨大的计算机，应用可以充分利用异构的计算、存储资源，且几乎不可感知硬件故障。

1. 存储虚拟化加速

云计算中的虚拟存储由本地存储和远程存储两部分构成。远程存储则是由存储节点虚拟出的分布式存储系统，提供高可靠性、高可用性、容量可扩放性和吞吐量可扩放性，是云平台中的主要存储方式。本地存储包括非易失性内存（Non-Volatile Memory，NVM）和 NVMe 高速闪存盘（flash storage），主要用于需要极致性能，但数据不需要高可靠性存储的分布式数据库等应用。

虚拟存储提供给客户的最基本服务是块存储（block storage），可以作为块设备（block device）挂载到虚拟机作为磁盘使用。云服务还提供了对象存储（object storage）、文件存储（file storage）等存储服务。这类服务大多提供类似键-值（key-value）映射的抽象，即用户指定键，读取（GET）或写入（PUT）相应的值。键-值存储作为一种基础数据结构，可以分为持久化存储、临时存储；根据是否需要复制和容灾，是否支持事务（transaction），提供强一致性或最终一致性[284]，是否支持范围索引、二级索引、内容索引等，可以组合出形形色色的存储系统，满足不同应用的需求。

虚拟存储系统的两个基本逻辑概念是客户端和服务器。如图 7.2 所示，客户端是云存储服务的使用者，如云上承载客户虚拟机的计算节点；服务器提供块存储、对象存储、文件存储等抽象，把逻辑存储的读写请求映射到物理存储介质的读写请求。多个客户端可能共享同一个虚拟存储，例如分布式数据处理系统中的多个计算节点可能需要访问共享的原始数据和配置参数，数据处理的中间结果也可以通过存储来传递。同一个虚拟存储可能对应多个存储服务器，用于实现存储的容量可扩放性、吞吐量可扩放性、容错和高可用。

上述存储服务器的结构是较为简化的，事实上往往分为多个层次。例如，微软 Azure 的云存储服务分为前端节点、中间节点和后端节点[147]。前端节点负责解析和验证请求，并根据数据分片映射表（例如键的哈希值），分发到数据所在分片的中间节点。中间节点负责实现请求的处理和存储数据结构的处理，把用户的请求映射成一系列存储读写操作，分发给对应的后端节点。后端节点负责实现数据的复制（replication）以及在物理介质上的存储。

在软件处理之外，数据中心存储还有显著的网络开销。在数据中心中，由于存储服务器需要安装较大容量的存储介质，存储节点的硬件配置一般与计算节点是不同的。此外，计算节点上由于运行客户的虚拟机，虚拟机监控器软件经常需要升级以增加新功能和修补安全漏洞，因此计算节点的稳定性通常比存储节

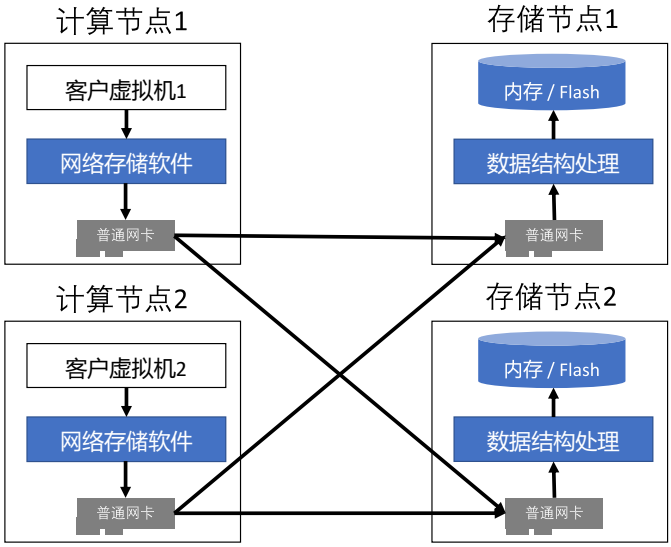


图 7.2　数据中心云存储的简要架构。

点低。为了保证存储的高可用性，存储节点与计算节点通常是分离在不同物理主机上的。因此，计算节点上的存储客户端通常需要把数据从存储服务器通过网络搬运过来。也就是说，客户虚拟机的每个 I/O 请求都需要被虚拟机监控器捕获，然后从计算节点上虚拟机监控器内的存储客户端软件出发，依次经过存储服务器的前端、中间和后端节点的处理，才能到达存储介质。为了保证数据的安全性，物理存储介质上的数据一般需要加密存储。为了节约存储空间，降低单位存储容量的存储介质成本，很多云厂商还会对存储的内容进行压缩。压缩和加密一般在存储服务器上进行，属于计算密集型操作。例如，根据实验，在 LZ77 较好的压缩率下，一个服务器 CPU 核心每秒通常只能压缩 100 MB 的数据；对 1 KB 的块，AES 加密和 SHA-256 签名每秒也只能处理 100 MB 的数据。

由于软件处理和网络传输的开销，云计算平台上块存储的延迟一般为 0.5 至 1 毫秒，对象存储的延迟一般为 1 至 10 毫秒[94]，远高于物理存储介质的延迟（如 SSD 的延迟一般为 0.1 毫秒）。此外，云存储的吞吐量也低于相应的物理存储介质，例如 SSD 云盘最高的吞吐量为 50 K 次 I/O 每秒，而单块数据中心级 SSD 的吞吐量就已达数百 K 次 I/O 每秒[94]。为了充分利用最新的数据中心存储硬件的性能，云存储服务需要全栈优化。例如，很多数据中心在存储协议栈的网络传输方面，已经使用 RDMA 协议降低了网络协议栈的 CPU 开销和延迟[41]。一些数据中心还通过改进云存储的协议栈，把客户端、服务器前端、中间、后端节点的功能进行适当的整合，减少层次[151]。HyperLoop[285] 利用 RDMA 网卡和非易失性内存（NVM）降低了存储写入事务的延迟。

2. 远程过程调用和消息队列加速

分布式系统中的消息传递通常采用远程过程调用（RPC）或消息队列（message queue）模型，或者两者的结合。在 RPC 模型中，服务器端注册一个

过程（procedure）来响应客户端的 RPC 请求。在消息队列模型中，生产者将消息广播或者分发到若干消费者。为了实现生产者与消费者的解耦以及消息的缓冲和可靠投递，消息队列模型往往引入一个经纪人（broker）服务，如 Kafka[77]。在编程接口方面，分布式应用程序通常使用 RPC 库和消息队列中间件（middleware），这些库和中间件则依赖操作系统的套接字（socket）接口来发送和接收消息。本文研究了操作系统套接字接口的加速，但没有考虑更上层的 RPC 和消息队列中间件。谷歌的研究[6] 表明，这些消息中间件经常增加数十微秒的延迟，占到整个端到端网络延迟的很大一部分。为了降低分布式系统的端到端消息传递延迟，有必要利用可编程网卡等硬件和用户态库等软件，实现高性能的 RPC 和消息队列。一种方案是在本文第 6 章的用户态套接字系统基础上，实现 RPC 和消息队列等高层抽象；另一种方案是打破传统网络协议栈边界的全栈优化，如最近的 eRPC[21] 是这方面很有意义的探索。对于消息队列等比较简单的应用，甚至可以探索在可编程网卡中实现，绕过主机 CPU。

3. 基于微内核的用户态操作系统

本文第 6 章提出了一个用户态网络协议栈 SocksDirect。第 6 章的技术可以用于加速操作系统的更多抽象。

在网络协议栈之外，操作系统的存储协议栈也有较高的开销。Linux 存储协议栈逻辑上由五层构成。首先，是与网络协议栈类似的虚拟文件系统层，提供基于文件描述符的 API。其次，文件系统层实现文件系统的抽象，提供文件的路径查找、权限管理、空间分配等功能。第三，缓存缓冲层与 Linux 的内存管理机制紧密结合，负责管理读缓存和写缓冲，以及页面换入换出机制。第四，块设备层把存储设备抽象为若干个 “块”（block），实现块访问的合并、排序等。第五，在设备驱动层，存储介质驱动程序与硬件通信以读取和写入硬盘块。在存储协议栈中，虚拟文件系统层也是开销的重要来源。对数据库等很多使用直接 I/O 的应用，文件系统、缓存缓冲层是不必要的。与网络协议栈类似，存储协议栈中也存在多次数据拷贝。对于很多应用，还存在存储和网络协议栈间的拷贝。基于页面重映射的零拷贝技术可以用于网络和存储协议栈，实现每份数据在物理内存中只有一份拷贝，在协议栈间复制的仅是虚拟内存的映射关系。

本文第 6 章的技术有更广阔的应用前景：基于微内核的用户态操作系统。操作系统主要包括三方面的功能：资源虚拟化、进程间通信和高层次抽象。第 6 章实现了网络资源的虚拟化和进程间的消息传递，提供了套接字的高层次抽象。用户态存储协议栈可以实现存储资源的虚拟化和文件系统的高层次抽象。其余的操作系统功能还包括计算资源虚拟化（即进程调度）和进程间同步（如锁和信号量）。这些功能可以在用户态守护进程中实现，也可以在可编程网卡中实现。将传统操作系统的功能被移动到用户态和可编程网卡后，就可以采用微内核，并保

持与现有应用程序的兼容性。

基于微内核的操作系统不仅具有更高的性能，还便于实现通用分布式应用的高可用性，这将是下一小节讨论的主题。

4. 通用分布式应用的高可用性

硬件故障和操作系统崩溃都可能导致分布式应用的部分节点出错。分布式应用的高可用性是很重要的。很多现有的请求处理和批量处理系统可以简化分布式应用的容错编程。这些程序通常需要程序员把组昂泰显式地从计算中分离，并把状态存储到一个容错存储系统中。然而，很多现有应用（如 Node.js，Memcached 和 Tensorflow 中的 Python 逻辑）并不原生支持容错。此外，容错编程框架通常比不容错的版本性能低。我们希望解决通用分布式应用程序的透明和高效容错的挑战。具体来说，挑战分为进程迁移、确定性重放和分布式快照三个方面。

首先，在不同层次上的容错存在着权衡。在体系结构层次上的容错需要定制化硬件。在虚拟机层次上的容错机制认为所有网络通信都是双向的（因为有数据传输和 ACK 确认报文），并且不能发现诸如进程间通信的高层次语义。系统调用层次上的容错需要对操作系统内核的修改以实现进程迁移，例如，从源主机把进程状态提取出来，并注入到目的主机里。Linux 系统中的进程迁移较为复杂，因为来自不同进程的状态混杂在一个宏内核中。而 Unikernel 方法不能支持很多现有的进程间通信机制。为此，未来的研究工作可以借鉴本文第 6 章的 SocksDirect 架构，设计一个分布式的用户态运行库操作系统，与现有 Linux 应用程序编程接口兼容。进程的内存快照同时获得了运行库和应用程序的状态，同时保留了高层语义，便于优化。

第二，状态机复制（State Machine Replication，SMR）和快照重放（snapshot replay）是实现容错的两种主要方法。SMR 至少需要两台主机才能执行完全相同的应用程序，从而引入 CPU 开销。基于快照的系统通常在两个相邻快照之间的间隔期间缓冲应用程序的输出，因为当主机发生故障时，系统无法保证自上次快照以来的确定性执行。这种所谓的输出提交（output commit）问题为透明容错系统引入了显着的请求服务延迟。或者，记录应用程序的所有非确定性事件仍然会产生很大的开销。未来的研究方向是采用根据其最近的执行历史来预测应用程序的非确定性事件。如果预测正确，则应用程序继续。否则，等待很短的时间来实现预测，因为许多不确定性源于微小的时间波动。这样，系统只需在等待超时的时候记录错误预测的事件，减少记录开销。

第三，透明容错机制需要在不暂停整个系统的情况下拍摄分布式应用程序的快照。一致的快照算法要求所有主机以相同的速度拍摄快照，并在任何主机发生故障时同时回滚。这种全局同步的行为与容错的目标相矛盾，容错需要系统在主机发生故障时继续提供对延迟敏感的请求。如何异步地对分布式系统生成一

致的快照是未来的研究方向。

5. 基于热迁移的数据中心资源打包

现代数据中心的资源利用率较低，有较大的优化空间。例如，数据中心内大部分物理服务器的平均使用率只有大约 10% 到 15%，但是闲置时的功耗却与使用率最高时相差仅 30%[4]。一方面，服务器和硬件的节能设计可以改进，以在使用率较低甚至闲置时尽量降低功耗。另一方面，云计算的一项优势就是支持热迁移，理论上可以在客户几乎不感知的情况下，根据虚拟机的计算、内存、存储、网络等需求，把需求互补的虚拟机打包安排在一台物理服务器主机上，从而最大化利用物理服务器的各种硬件资源。目前，热迁移对客户服务会造成一段时间的性能下降甚至中断，因此在云计算中的利用还不够广泛。

热迁移的主要难点在于生成一致的虚拟机状态快照。在异构硬件组成的数据中心，虚拟机的状态不仅包括其 CPU、内存和本地存储的状态，还包括 GPU、网卡等硬件的状态。这些硬件往往没有提供高效的快照和恢复功能，从而只能在新物理节点上重新加载硬件驱动程序、初始化硬件内部状态，带来较高的延迟。第 4 章的 ClickNP 框架可以实现网络元件内部状态的快照和迁移，因此采用 ClickNP 框架编写的可编程网卡可以高效热迁移。GPU-Direct RDMA 等技术可以实现 GPU 内部存储的高效传输。对于本地存储，可以利用存储解聚的思想，不必等待数据迁移完成，而可以在迁移过程中把新节点上虚拟机的访问请求重定向到原有存储。

6. 端云融合的分布式操作系统

智能终端（如智能手机和 PC）和云（数据中心）是目前最重要的两类计算和存储设备。端和云的计算和存储能力都在快速增长，且随着 5G 技术的发展，端云之间的通信成本将大幅降低，带宽和延迟也将显著改善。因此，端云融合将成为重要的趋势。一方面，端上的应用将可以更细粒度地调用云上的服务、访问云上的数据；另一方面，云上用于高性能通信和计算的技术也会逐渐应用到端上。例如，通过在端上部署可编程网卡，可以为 5G 网络通信降低功耗开销，消除性能瓶颈；可以提高访问 Flash 存储的性能，本文第 6 章的高性能用户态套接字技术也可以在端上得到应用。

当端上的应用需要较大的算力、存储空间，或需要长时间地运行任务时，往往需要卸载到云上处理。无服务器计算（serverless computing）简化了云上的任务调度，但仍然需要开发者在端云间进行任务划分，并设计远程过程调用（RPC）接口。本文期待未来的工作提出端云融合的分布式操作系统。在端云融合操作系统中，应用在端和云上有相同的开发环境和运行环境。应用可以直接访问云上的计算、存储和网络资源，而无需关注其本身是运行在端上还是云上。一个应用由若干进程组成，每个进程运行在端或云上的一个计算设备上，并允许动态迁移

计算设备；不同进程可以运行在不同的计算设备上，进程间可通过分布式操作系统提供的基于消息传递或共享数据结构存储的中间件来通信。为了支持网络测量、爬虫等需要分布在不同地理位置的应用、由于隐私保护法律需要限制数据的处理和存储区域的应用以及为了开发者提升数据局部性，应用可以指定进程可以运行、数据可以存储的计算设备集合。分布式操作系统的主要挑战是将进程内存状态和共享数据结构存储中的数据在端云的计算节点之间合理复制和缓存，以提高数据局部性，降低通信延迟。

作为一个示例，用户在 PC 终端上启动一个并行编译任务，传统上该任务的并行度将受限于 PC 上的 CPU 核数，且消耗 PC 较多的电池能量。在端云融合的分布式系统中，如果传输源文件及编译结果的通信代价低于计算的代价（或者待编译文件在云上已经有副本），该任务将被分发到云端的多台服务器上，编译任务的不同子进程将运行在不同服务器的 CPU 核上，并行度理论上仅受限于编译任务可并行执行的最大子进程数以及端云之间的通信带宽。对于编译结束后的链接过程，分布式操作系统不必将编译结果传回 PC 终端再上传到云上做链接，而是可以直接在云端的计算节点间通信，完成链接操作，最后 PC 终端只需要下载链接后的最终二进制。甚至，如果该二进制与用户的交互并不频繁，延迟也不敏感，分布式操作系统可以不传输该二进制到端上，而是在该二进制执行时只传输用户与计算节点间的交互，相当于在操作远程的服务器。

值得讨论的是分布式操作系统的抽象层次。如果在 Linux 操作系统的层次上抽象，兼容性将最好，可以实现现有并行程序的自动分布式处理。但 Linux 系统调用的抽象层次较低，如果开发者不显式提供更多信息，较难预测应用未来访问的资源，对一些类型的应用将性能不佳。实现分布式 Linux 操作系统的一种可能的方式是远程系统调用，即对每个迁移到远程的进程，在本地保留一个影子进程；捕获远程系统上的系统调用，发送到本地影子进程并实际调用，并将系统调用的结果发送到远程系统。应用进程需要等待远程系统调用返回，即系统调用的延迟大大增加，因此该实现方案的性能可能不佳。

7.2.4　系统创新

把整个世界看作一台大型计算机是微软 CEO 萨提亚 · 纳德拉的愿景，也是很多系统研究者的梦想。云计算的成功使数据中心吸纳了人类世界大部分的计算和存储，而数据中心可以看作是由计算、内存、存储和网络及互连四部分组成的一台大规模计算机。英伟达 CEO 黄仁勋[286]、谷歌工程副总裁 Luiz Barroso[4] 等已经将数据中心看作一台大规模计算机。系统创新就是从全局的角度考虑各种软硬件组件如何高效而可靠地协同工作，以及给用户提供怎样的抽象。

1. 基于可编程网卡的内存解聚与二层内存

内存解聚（Memory Disaggregation）指的是计算机的 CPU 可以自由而透明地高效共享远程计算机的内存，这样可以大大增加内存的利用率，降低云计算平台的成本。尽管目前数据中心网络的性能远低于 CPU 访问主机内存的性能，幸运的是，利用内存访问的局部性，如果一部分热数据仍在本地，剩余的数据通过远程访问，则远程内存的带宽和延迟要求能比本地内存大大降低。加州大学伯克利分校的研究指出，要把内存解聚后的系统性能与全部使用本地内存的差距控制在 5% 以内，带宽需要达到 40 Gbps，端到端往返延迟需要不超过 3 至 5 微秒，这是目前的数据中心网络可以达到的。

非易失性内存（Non-Volatile Memory，NVM）是内存和存储领域的研究热点。相比传统的 NAND Flash，非易失性内存的存取速度要快得多。虽然它们短期内还不能完全取代 DRAM，但至少希望能够在不久的将来会代替一部分普通内存。非易失性内存相比 DRAM 有价格低，容量大，功耗小，断电可以保持数据的优点，但同时又有存取速度慢，写入周期有限等限制。非易失性内存作为 DRAM 和 NAND Flash 之间的存储层级，既可以用来扩充 DRAM 内存的容量，又可以作为快速的持久化存储。如何有效地利用非易失性内存目前是一个重要研究方向。

内存解聚和非易失性内存构成了二级内存（second-tier memory），即比 DRAM 慢但容量更大的内存[287]。为了扩充内存容量并尽量减少对应用性能的影响，二级内存系统需要把热数据放在本地 DRAM 中，把冷数据放在解聚的远程内存或非易失性内存中。目前的大多数内存解聚系统（如 Infiniswap[288]）和二级内存系统（如 Thermostat[289]）采用页面换入换出的方式。首先，页面换入换出需要经过操作系统内核，每换入一个页面需要增加约 2.5 微秒的内核开销，而允许的端到端访问延迟只有 3 至 5 微秒。其次，解聚到远程存储的内存一般是冷数据，这些数据的访问粒度可能小于一般为 4 KB 的页面大小，因此传输一整个页面不仅浪费网络带宽，也增加了延迟。最后，页面换入换出的决策在软件上进行，难以准确统计每个页面的访问频率。

未来的研究方向是基于可编程网卡的内存解聚和二级内存。通过使用直接内存映射取代页面换入换出，避免了操作系统内核的开销，也把内存访问的粒度从 4 KB 的页面降低到 64 字节的缓存行。本地与远程内存仍然是以页面为单位，依靠页表维护映射关系。可编程网卡可以统计每个页面的远程内存访问，从而及时把热数据迁移到本地内存，避免长期影响性能。

基于现有 CPU 和 PCIe 体系结构实现基于直接内存映射的内存解聚存在一系列技术挑战。幸运的是，CPU 厂商已经意识到了同样的问题。我们期待随着 CCIX 等主机内互连协议的实现，直接内存映射的可编程网卡与 CPU 之间将达

到更好的吞吐量和延迟，并且直接内存映射区域可以像主机内存一样运行所有指令。

2. 基于数据中心网络的可扩放全序通信

传统数据中心网络中的延迟是任意的，从而消息不能保证按照一致的顺序被投递。例如，分布式数据库的多个分片向多个副本发送日志。每个副本可能以不同的顺序收到各个分片的日志。如果不加特殊处理，这种不一致的顺序可能破坏数据一致性。解决这个问题的方案经常引入同步开销，并使分布式系统的设计复杂化。

全序通信提供了一种抽象，保证不同的接收端按照一致的顺序处理来自发送端的消息。全序（但不可靠）地传递一组消息可以简化和加速很多分布式应用，例如减少多版本并发控制（MVCC）协议中的冲突，加速分布式共识协议，实现无中心瓶颈的可扩放日志复制（replication），提早检测 TCP 尾丢包，降低散播-汇聚（scatter-gather）模式远程过程调用（RPC）的尾延迟。例如，近年来，通过提高数据中心内传输的有序性，分布式共识协议（consensus protocol）和分布式事务的性能得到了极大的提升。快速 Paxos[290-293] 协议采用尽力而为的方法提高传输的有序性。Speculative Paxos[294] 和 NOPaxos[295] 利用可编程交换机作为中心化的序列号发生器或序列化点。NetPaxos[296-297] 和[298] 把传统 Paxos 协议放在网络交换机中实现。Eris[200] 提出使用网络交换机作为序列号发生器，在网络中实现并发控制，实现了快速事务处理。HotOS ’19 上的工作[299] 提出构建同步，即网络延迟固定的数据中心网络，可以简化分布式系统的设计。

自从分布式系统研究的兴起，全序广播和多播问题就吸引了大量的研究。然而，现有方案受限于可扩放性或效率。一类研究工作利用逻辑上中心化的协调，例如中心化的序列号发生器，或者在发送端或接收端之间传递的令牌。近年来分布式系统和数据中心网络共同设计的研究工作属于此类。然而，这些中心化的方案难以扩放。另一类研究工作用完全分布式的协调，例如在接收端开始处理消息之前交换时间戳。这导致额外的网络通信开销和延迟，降低系统效率。此外，多播的语义还有一个限制，即所有接收者必须收到相同的消息。

相比全序多播，全序消息散播（Total-Order Message Scattering，TOMS）原语的应用范围更广。消息散射是一种一个主机同时发送一组（可能不同的）消息给多个主机的通信原语。消息散射在分布式系统中很常见。例如在分布式存储中，一个客户端把元数据写到一个存储站点，把数据写到另一个存储站点；与此同时，另一个客户端并发地读取它们。元数据和数据间的一致性要求这些操作被原子地散射到两个存储站点。全序消息散播在数据中心网络中一对多地散射一组消息，并保持可线性化的顺序，每条消息至多被投递一次。

为了支持更好的可扩放性，也为了加速除分布式事务外的更多分布式应用，

基于数据中心网络的可扩放全序通信是一个有趣的研究方向。在数据中心环境中，网络拓扑是规则的，交换机一般有较好的可编程性。全序消息散播把工作分配给每个交换机和终端服务器，从而实现了高可扩放性。核心设计原则是把顺序信息的处理与消息转发分离开来。为了得到顺序信息，利用可编程交换机，在网络中汇聚顺序信息，这形成了系统的 “控制面”。在 “数据面” 上，全序消息散播像往常一样转发消息，并在接收端缓冲并重排收到的消息。发送端给散播的每组消息打上递增的时间戳，而接收端需要按照时间戳的顺序向应用投递消息。控制面的顺序信息为接收端提供了 “在此之后收到的消息都晚于某个时间戳” 的屏障（barrier），使其可以按照时间戳顺序投递消息。

本研究的初步工作已经由合作者左格非发表在 ACM SOSP 2017 学生研究竞赛（SRC）上[300]。

全序通信研究的一大难点是可靠性。如果需要在有丢包和节点故障的网络中保证可靠全序通信，这至少与分布式共识（consensus）问题一样困难，将需要较为复杂的容错与故障恢复机制，且局部的故障很容易影响全局的通信效率。如果不保证通信的可靠性，而是只保证收到的数据包有序，则应用范围将大大缩小，必须与其他传统方法结合才能保证分布式系统的正确性，但可以大大减少乱序情况而提高效率。

分布式事务的其他方面也可以受益于与数据中心网络的协同设计。Hyperloop[285] 在存储节点上利用可编程网卡把写操作写入非易失性内存中的缓冲区，并立即向计算节点回复确认消息，再由存储节点上的软件异步处理非易失性内存中的写操作。这消除了写操作等待存储节点软件处理的延迟。Google Spanner[301] 利用全球同步的 GPS 时钟实现了跨地理区域复制的高性能数据库。

3. 结合在线事务、批量和流式处理的数据库

现代大数据处理主要有在线事务处理（OLTP）、批量处理（batch processing）和流式处理（stream processing）三种范式。在线事务处理用于需要较快响应时间、较强一致性的事务，一般每个事务只涉及数据集的一小部分，且更新操作频繁。批量处理主要用于离线数据分析，其特点是数据量和计算量都很大。流式处理适用于需要高实时性的分析任务，可以针对数据的改变增量地更新状态并输出结果。

传统上，大数据处理系统一般使用 lambda 架构，即在线事务处理作为批量处理和流式处理的数据源，其产生的数据更新分别同步到批量处理部分和流式处理部分。批量处理部分定期重新计算结果，而流式处理部分根据上次的批量处理结果和流式输入的更新数据来持续更新输出。最后，批量处理部分和流式处理部分的输出被合并起来，输出给用户。首先，lambda 架构需要数据分析人员显式把数据分成在线、批量和流式三部分，分别编写处理程序，并将结果归并，开

发较复杂，且容易导致不一致。其次，lambda 架构中的流式处理可能依赖上次批量处理的结果，批量处理延迟可能导致结果的不准确性，而这种延迟在性能上不一定必要。

近年来，在同一个数据库中结合在线事务处理（OLTP）和离线数据分析处理（OLAP）事务的 HTAP（Hybrid Tranactional and Analytical Processing）数据库开始流行。HTAP 数据库解决了从在线事务处理到批量处理分析的延迟问题，但仍然不支持流式处理。用户需要显式重新运行查询来获取更新后的批量处理结果，而且处理是基于查询开始时的数据库状态，不能反映数据库的实时状态。学术界提出的 DBToaster 等响应式数据库结合了在线事务处理和流式处理，但所有中间结果都被缓存和增量处理，其中的开销是很大的。例如，一些类型的批量处理难以增量更新，性能上比较合理的做法是允许一定的数据更新延迟。

未来的研究方向是同时高效支持在线事务处理、离线数据分析和流式处理的响应式数据库系统。响应式体现在三方面。首先，每个存储过程事务都对其他并行事务的更新操作是响应式的。基本表的更新被同步到运行着的离线数据分析和流式处理事务。这些正在运行的事务保存适当的中间状态，并增量更新之。因此，每个事务都天然地在事务完成时间被序列化，也就是存储过程事务的查询结果反映了数据库的实时状态。流式处理事务则被认为是持续运行的，能够把数据库增量更新对查询结果的改变实时报告给用户。

其次，事务处理的计算流图的 “推” 和 “拉” 是响应式的。在数据库内部的计算流图中，传统数据库的每个算子都是 “拉” 模式的，也就是每次用户需要查询结果时，就重新执行计算流图；而流式处理和响应式数据库中，每个算子都是 “推” 模式的，也就是每次基础表的数据有更新时，都会更新并保存所有中间算子的结果，直到更新最终查询结果，不论用户是否需要实时的更新。根据用户对更新时效的需求，数据库动态调整计算流图中每个算子的 “推” 和 “拉” 模式，以及 “推” 的频率。

最后，物理数据存储结构与索引是响应于数据访问模式的。把基础表的数据更新日志作为数据源，而基于行、列的数据存储结构都是缓存，为点查询和分析性查询分别优化。索引也被认为是缓存。视图和分析型查询的中间结果也可能被缓存下来。数据库需要根据数据的访问模式来调整缓存与否的选择，因为缓存可以加速读操作，但对写操作增加了负担。

参 考 文 献

[1] BARROSO L A, HÖLZLE U. The datacenter as a computer: An introduction to the design of warehouse-scale machines[J]. Synthesis lectures on computer architecture, 2009, 4(1): 1-108.

[2] BORKAR S, CHIEN A A. The future of microprocessors[J]. Communications of the ACM, 2011, 54(5):67-77.

[3] JOUPPI N, YOUNG C, PATIL N, et al. Motivation for and evaluation of the first tensor processing unit[J]. IEEE Micro, 2018, 38(3):10-19.

[4] BARROSO L A, HÖLZLE U, RANGANATHAN P. The datacenter as a computer: Designing warehouse-scale machines[J]. Synthesis Lectures on Computer Architecture, 2018, 13(3):i-189.

[5] BARROSO L A, CLIDARAS J, HÖLZLE U. The datacenter as a computer: An introduction to the design of warehouse-scale machines[J]. Synthesis lectures on computer architecture, 2013, 8(3):1-154.

[6] BARROSO L, MARTY M, PATTERSON D, et al. Attack of the killer microseconds[J]. Communications of the ACM, 2017, 60(4):48-54.

[7] PATEL P, BANSAL D, YUAN L, et al. Ananta: Cloud scale load balancing[C]//ACM SIGCOMM Computer Communication Review: volume 43. ACM, 2013: 207-218.

[8] DOBRESCU M, EGI N, ARGYRAKI K, et al. Routebricks: Exploiting parallelism to scale software routers[C/OL]//Proc. ACM SOSP. 2009: 15-28. http://doi.acm.org/10.1145/1629575.1629578.

[9] GANDHI R, LIU H H, HU Y C, et al. Duet: Cloud scale load balancing with hardware and software[J]. ACM SIGCOMM Computer Communication Review, 2015, 44(4):27-38.

[10] GREENBERG A. SDN for the Cloud[Z]. 2015.

[11] BELAY A, PREKAS G, PRIMORAC M, et al. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane[J]. ACM Transactions on Computer Systems (TOCS), 2017, 34(4):11.

[12] PETER S, LI J, ZHANG I, et al. Arrakis: The operating system is the control plane[J]. ACM Transactions on Computer Systems (TOCS), 2016, 33(4):11.

[13] RIZZO L. Netmap: a novel framework for fast packet i/o[C]//21st USENIX Security Symposium (USENIX Security 12). 2012: 101-112.

[14] INTEL. Data plane development kit[EB/OL]. 2014. https://dpdk.org/.

[15] Pf_ring[EB/OL]. 2019. http://www.ntop.org.

[16] MARINOS I, WATSON R N, HANDLEY M. Network stack specialization for performance [C]//ACM SIGCOMM Computer Communication Review: volume 44. ACM, 2014: 175-186.

[17] JEONG E, WOO S, JAMSHED M A, et al. mTCP: a highly scalable user-level TCP stack for multicore systems.[C]//NSDI '14. 2014: 489-502.

[18] Seastar: High-performance server-side application framework[EB/OL]. 2019. http://seastar.io/.

[19] High-performance network framework based on dpdk[EB/OL]. 2019. http://f-stack.org/.

[20] KALIA A, KAMINSKY M, ANDERSEN D G. FaSST: fast, scalable and simple distributed transactions with two-sided RDMA datagram rpcs[C]//OSDI '16. 2016: 185-201.

[21] KALIA A, KAMINSKY M, ANDERSEN D. Datacenter rpcs can be general and fast[C]// 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). Boston, MA: USENIX Association, 2019.

[22] MELLANOX. Messaging accelerator (vma)[EB/OL]. 2019. https://github.com/mellanox/libvma.

[23] POPE S, RIDDOCH D. Introduction to openonload—building application transparency and protocol conformance into application acceleration middleware[R]. 2011.

[24] Myricom dbl[EB/OL]. 2019. https://www.cspi.com/ethernet-products/software/dbl/.

[25] HUANG Y, GENG J, LIN D, et al. Los: A high performance and compatible user-level network operating system[C]//Proceedings of the First Asia-Pacific Workshop on Networking. ACM, 2017: 50-56.

[26] FITZPATRICK B. Distributed caching with memcached[J]. Linux journal, 2004, 2004(124): 5.

[27] KAPOOR R, PORTER G, TEWARI M, et al. Chronos: predictable low latency for data center applications[C]//Proceedings of the Third ACM Symposium on Cloud Computing. ACM, 2012: 9.

[28] OUSTERHOUT J, AGRAWAL P, ERICKSON D, et al. The case for RAMClouds: scalable high-performance storage entirely in dram[J]. ACM SIGOPS Operating Systems Review, 2010, 43(4):92-105.

[29] OUSTERHOUT J, GOPALAN A, GUPTA A, et al. The ramcloud storage system[J]. ACM Transactions on Computer Systems (TOCS), 2015, 33(3).

[30] LIM H, HAN D, ANDERSEN D G, et al. MICA: a holistic approach to fast in-memory key-value storage[C]//NSDI '14. 2014: 429-444.

[31] LI S, LIM H, LEE V W, et al. Full-stack architecting to achieve a billion requests per second throughput on a single key-value store server platform[J]. ACM Transactions on Computer

Systems (TOCS), 2016, 34(2):5.

[32] MAO Y, KOHLER E, MORRIS R T. Cache craftiness for fast multicore key-value storage [C]//Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012: 183-196.

[33] FAN B, ANDERSEN D G, KAMINSKY M. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing[C]//NSDI '13. 2013: 371-384.

[34] LI X, ANDERSEN D G, KAMINSKY M, et al. Algorithmic improvements for fast concurrent cuckoo hashing[C]//Eurosys '14. ACM, 2014: 27.

[35] Infiniband architecture specification: Release 1.0[M]. InfiniBand Trade Association, 2000.

[36] HAN S, JANG K, PARK K, et al. Packetshader: a gpu-accelerated software router[C]// volume 41. ACM, 2011: 195-206.

[37] Cavium Networks OCTEON II processors.[Z].

[38] Netronome Flow Processor NFP-6xxx.[Z].

[39] CORPORATION M. Information about the tcp chimney offload[EB/OL]. 2008. https://support.microsoft.com/en-us/help/951037/information-about-the-tcp-chimney-offload-receive-side-scaling-and-net.

[40] GUO C. Rdma in data centers: Looking back and looking forward[Z]. 2017.

[41] GUO C, WU H, DENG Z, et al. Rdma over commodity ethernet at scale[C]//Proceedings of the 2016 ACM SIGCOMM Conference. ACM, 2016: 202-215.

[42] ASSOCIATION I T. Infiniband architecture specification release 1.2.1 annex a17: Rocev2 [Z]. 2014.

[43] rsocket(7) - linux man page.[J/OL]. 2019. https://linux.die.net/man/7/.

[44] PINKERTON J. Sockets direct protocol v1. 0 rdma consortium[J]. 2019.

[45] RUSSELL R. The extended sockets interface for accessing rdma hardware[C]//Proceedings of the 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2008). Nov, 2008: 279-284.

[46] KALIA A, KAMINSKY M, ANDERSEN D G. Using RDMA efficiently for key-value services[C]//ACM SIGCOMM Computer Communication Review: volume 44. ACM, 2014: 295-306.

[47] KALIA A, KAMINSKY M, ANDERSEN D G. Design guidelines for high performance RDMA systems[C]//USENIX ATC '16. 2016.

[48] PUTNAM A, CAULFIELD A M, CHUNG E S, et al. A reconfigurable fabric for accelerating large-scale datacenter services[J]. ACM SIGARCH Computer Architecture News, 2014, 42 (3):13-24.

[49] Vivado Design Suite.[Z].

[50] CORPORATION I. Intel high level synthesis compiler[EB/OL]. 2019. https://www.intel.com/content/www/us/en/programmable/products/design-software/high-level-design/intel-hls-compiler/support.html.

[51] NIKHIL R S, ARVIND. What is bluespec?[J/OL]. ACM SIGDA Newsletter, 2009, 39(1): 1-1. http://doi.acm.org/10.1145/1862876.1862877.

[52] AUERBACH J, BACON D F, CHENG P, et al. Lime: a java-compatible and synthesizable language for heterogeneous architectures[C]//ACM SIGPLAN Notices: volume 45. ACM, 2010: 89-108.

[53] BACHRACH J, VO H, RICHARDS B, et al. Chisel: constructing hardware in a scala embedded language[C]//DAC Design Automation Conference 2012. IEEE, 2012: 1212-1221.

[54] BACON D F, RABBAH R, SHUKLA S. Fpga programming for the masses[J]. Communications of the ACM, 2013, 56(4):56-63.

[55] SINGH D. Implementing fpga design with the opencl standard[J]. Altera whitepaper, 2011.

[56] WESTER R. A transformation-based approach to hardware design using higher-order functions[J]. 2015.

[57] Altera SDK for OpenCL.[Z].

[58] XILINX. Sdaccel: Enabling hardware-accelerated software[EB/OL]. 2019. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[59] STRACHEY C. Time sharing in large fast computers[C]//Communications of the ACM: volume 2. ASSOC COMPUTING MACHINERY 1515 BROADWAY, NEW YORK, NY 10036, 1959: 12-13.

[60] AMDAHL G M, BLAAUW G A, BROOKS F. Architecture of the ibm system/360[J]. IBM Journal of Research and Development, 1964, 8(2):87-101.

[61] BACH M J, et al. The design of the unix operating system: volume 5[M]. Prentice-Hall Englewood Cliffs, NJ, 1986.

[62] POPEK G J, GOLDBERG R P. Formal requirements for virtualizable third generation architectures[J]. Communications of the ACM, 1974, 17(7):412-421.

[63] AGESEN O, GARTHWAITE A, SHELDON J, et al. The evolution of an x86 virtual machine monitor[J]. ACM SIGOPS Operating Systems Review, 2010, 44(4):3-18.

[64] GHEMAWAT S, GOBIOFF H, LEUNG S T. The google file system[J]. 2003.

[65] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2):4.

[66] DEAN J, GHEMAWAT S. Mapreduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1):107-113.

[67] WHITE T. Hadoop: The definitive guide[M]. ” O’Reilly Media, Inc.”, 2012.

[68] ZAHARIA M, CHOWDHURY M, FRANKLIN M J, et al. Spark: Cluster computing with working sets.[J]. HotCloud, 2010, 10(10-10):95.

[69] PAGE L, BRIN S, MOTWANI R, et al. The pagerank citation ranking: Bringing order to the web.[R]. Stanford InfoLab, 1999.

[70] DRAGOJEVIĆ A, NARAYANAN D, CASTRO M, et al. FaRM: fast remote memory[C]// NSDI '14. 2014.

[71] AL-FARES M, LOUKISSAS A, VAHDAT A. A scalable, commodity data center network architecture[C]//ACM SIGCOMM Computer Communication Review: volume 38. ACM, 2008: 63-74.

[72] WEI X, SHI J, CHEN Y, et al. Fast in-memory transaction processing using rdma and htm [C]//Proceedings of the 25th Symposium on Operating Systems Principles. ACM, 2015: 87-104.

[73] CHEN Y, WEI X, SHI J, et al. Fast and general distributed transactions using rdma and htm [C]//Proceedings of the Eleventh European Conference on Computer Systems. ACM, 2016: 26.

[74] WEI X, DONG Z, CHEN R, et al. Deconstructing rdma-enabled distributed transactions: Hybrid is better![C]//13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). 2018: 233-251.

[75] WANG S, LOU C, CHEN R, et al. Fast and concurrent {RDF} queries using rdma-assisted {GPU} graph exploration[C]//2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18). 2018: 651-664.

[76] KSHEMKALYANI A D, SINGHAL M. Distributed computing: principles, algorithms, and systems[M]. Cambridge University Press, 2011.

[77] KREPS J, NARKHEDE N, RAO J, et al. Kafka: A distributed messaging system for log processing[C]//Proceedings of the NetDB. 2011: 1-7.

[78] ZAWODNY J. Redis: Lightweight key/value store that goes the extra mile[J]. Linux Magazine, 2009, 79.

[79] ATTIYA H, BAR-NOY A, DOLEV D. Sharing memory robustly in message-passing systems [J]. Journal of the ACM (JACM), 1995, 42(1):124-142.

[80] 刘铁岩王太峰 高飞. 分布式机器学习：算法、理论与实践[M]. 机械工业出版社, 2018.

[81] LI M, ANDERSEN D G, PARK J W. Scaling distributed machine learning with the parameter server.[C]//2014.

[82] DEAN J, CORRADO G, MONGA R, et al. Large scale distributed deep networks[C]// Advances in neural information processing systems. 2012: 1223-1231.

[83] KRIZHEVSKY A, SUTSKEVER I, HINTON G E. Imagenet classification with deep con-

volutional neural networks[C]//Advances in neural information processing systems. 2012: 1097-1105.

[84] Multiverso: a distributed key-value stores to make writing distributed system easily[EB/OL]. https://github.com/Microsoft/multiverso/wiki/Overview.

[85] ABADI M, BARHAM P, CHEN J, et al. Tensorflow: A system for large-scale machine learning[C]//12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). 2016: 265-283.

[86] DENNARD R H, GAENSSLEN F H, RIDEOUT V L, et al. Design of ion-implanted mosfet's with very small physical dimensions[J]. IEEE Journal of Solid-State Circuits, 1974, 9(5): 256-268.

[87] BORKAR S. Design challenges of technology scaling[J]. IEEE micro, 1999, 19(4):23-29.

[88] POLLACK F. Pollack' s rule of thumb for microprocessor performance and area[J]. URL: http://en. wikipedia. org/wiki/Pollack' s_Rule.

[89] 刘慈欣. 三体 2 · 黑暗森林，中部咒语第 8 部分[M]. 2008.

[90] HARDWARE T. Intel xeon e5-2600 v4 broadwell-ep review[EB/OL]. 2016. https://www.tomshardware.com/reviews/intel-xeon-e5-2600-v4-broadwell-ep,4514-2.html.

[91] BERNSTEIN D. Containers and cloud: From lxc to docker to kubernetes[J]. IEEE Cloud Computing, 2014, 1(3):81-84.

[92] ZHOU H, CHEN M, LIN Q, et al. Overload control for scaling wechat microservices[C]// Proceedings of the ACM Symposium on Cloud Computing. ACM, 2018: 149-161.

[93] BURNS B, GRANT B, OPPENHEIMER D, et al. Borg, omega, and kubernetes[J]. 2016.

[94] JONAS E, SCHLEIER-SMITH J, SREEKANTI V, et al. Cloud programming simplified: A berkeley view on serverless computing[J]. arXiv preprint arXiv:1902.03383, 2019.

[95] RAMAKRISHNAN R, SRIDHARAN B, DOUCEUR J R, et al. Azure data lake store: a hyperscale distributed file service for big data analytics[C]//Proceedings of the 2017 ACM International Conference on Management of Data. ACM, 2017: 51-63.

[96] MORITZ P, NISHIHARA R, WANG S, et al. Ray: A distributed framework for emerging {AI} applications[C]//13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). 2018: 561-577.

[97] MCKEOWN N, ANDERSON T, BALAKRISHNAN H, et al. Openflow: enabling innovation in campus networks[J]. ACM SIGCOMM Computer Communication Review, 2008, 38 (2):69-74.

[98] KOPONEN T, CASADO M, GUDE N, et al. Onix: A distributed control platform for large-scale production networks.[C]//OSDI: volume 10. 2010: 1-6.

[99] VOELLMY A, AGARWAL A, HUDAK P. Nettle: Functional reactive programming for

openflow networks[R]. YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE, 2010.

[100] FOSTER N, HARRISON R, FREEDMAN M J, et al. Frenetic: A network programming language[J]. ACM Sigplan Notices, 2011, 46(9):279-291.

[101] JIN X, GOSSELS J, REXFORD J, et al. Covisor: A compositional hypervisor for software-defined networks[C]//12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15). 2015: 87-101.

[102] BOSSHART P, DALY D, GIBB G, et al. P4: Programming protocol-independent packet processors[J]. ACM SIGCOMM Computer Communication Review, 2014, 44(3):87-95.

[103] BOSSHART P, GIBB G, KIM H S, et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn[J]. ACM SIGCOMM Computer Communication Review, 2013, 43(4):99-110.

[104] KAUFMANN A, PETER S, SHARMA N K, et al. High performance packet processing with flexnic[C]//ACM SIGARCH Computer Architecture News: volume 44. ACM, 2016: 67-81.

[105] WANG H, SOULÉ R, DANG H T, et al. P4fpga: A rapid prototyping framework for p4[C]//Proceedings of the Symposium on SDN Research. ACM, 2017: 122-135.

[106] SHAHBAZ M, CHOI S, PFAFF B, et al. Pisces: A programmable, protocol-independent software switch[C]//Proceedings of the 2016 ACM SIGCOMM Conference. ACM, 2016: 525-538.

[107] NETWORKS B. Tofino: World's fastest p4-programmable ethernet switch asics[EB/OL]. 2019. https://www.barefootnetworks.com/products/brief-tofino/.

[108] Mellanox adapters programmer's reference manual (prm)[EB/OL]. 2019. http://www.mellanox.com/related-docs/user_manuals/Ethernet_Adapters_Programming_Manual.pdf.

[109] XILINX. Sdnet packet processor user guide[EB/OL]. 2017. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/UG1012-sdnet-packet-processor.pdf.

[110] FIRESTONE D. {VFP}: A virtual switch platform for host {SDN} in the public cloud[C]//14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17). 2017: 315-328.

[111] SON J, XIONG Y, TAN K, et al. Protego: Cloud-scale multitenant ipsec gateway[C]//2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17). 2017: 473-485.

[112] LAN C. An architecture for network function virtualization[M]. UC Berkeley, 2018.

[113] NETWORKS F. F5 load balancer[EB/OL]. https://www.f5.com/services/resources/glossary/load-balancer.

[114] 3RD GENERATION PARTNERSHIP PROJECT (3GPP). 3gpp ts 23.501, system architecture for the 5g system (5gs), release 16[J]. 2018.

[115] 3RD GENERATION PARTNERSHIP PROJECT (3GPP). 3gpp ts 38.300, technical specification group radio access network; nr; nr and ng-ran overall description, release 16[J]. 2018.

[116] KOHLER E, MORRIS R, CHEN B, et al. The click modular router[J]. ACM Transactions on Computer Systems (TOCS), 2000, 18(3):263-297.

[117] MARTINS J, AHMED M, RAICIU C, et al. Clickos and the art of network function virtualization[C]//11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14). 2014: 459-473.

[118] PANDA A, HAN S, JANG K, et al. Netbricks: Taking the v out of NFV[C/OL]//12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Savannah, GA: USENIX Association, 2016: 203-216. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda.

[119] CLARK D D, JACOBSON V, ROMKEY J, et al. An analysis of tcp processing overhead[J]. IEEE Communications magazine, 1989, 27(6):23-29.

[120] BOYD-WICKIZER S, CLEMENTS A T, MAO Y, et al. An analysis of linux scalability to many cores.[C]//OSDI: volume 10. 2010: 86-93.

[121] TECHNOLOGIES M. Connectx-5 en single/dual-port adapter supporting 100gb/s ethernet [EB/OL]. 2018. http://www.mellanox.com/page/products_dyn?product_family=260&mtag=connectx_5_en_card.

[122] KAUFMANN A, PETER S, ANDERSON T E, et al. Flexnic: Rethinking network dma.[C]//HotOS '15. 2015.

[123] SULTANA N, GALEA S, GREAVES D, et al. Emu: Rapid prototyping of networking services[C]//2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17). 2017: 459-471.

[124] RADHAKRISHNAN S, GENG Y, JEYAKUMAR V, et al. {SENIC}: Scalable {NIC} for end-host rate limiting[C]//11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14). 2014: 475-488.

[125] RAM K K, MUDIGONDA J, COX A L, et al. snich: Efficient last hop networking in the data center[C]//Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems. ACM, 2010: 26.

[126] LE Y, CHANG H, MUKHERJEE S, et al. Uno: uniflying host and smart nic offload for flexible packet processing[C]//Proceedings of the 2017 Symposium on Cloud Computing. ACM, 2017: 506-519.

[127] STEPHENS B, AKELLA A, SWIFT M M. Your programmable nic should be a programmable switch[C]//Proceedings of the 17th ACM Workshop on Hot Topics in Networks. ACM, 2018: 36-42.

[128] KAFFES K, CHONG T, HUMPHRIES J T, et al. Shinjuku: Preemptive scheduling for μsecond-scale tail latency[C]//16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19). 2019: 345-360.

[129] OUSTERHOUT A, FRIED J, BEHRENS J, et al. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads[C]//16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19). 2019: 361-378.

[130] LI B, RUAN Z, XIAO W, et al. Kv-direct: High-performance in-memory key-value store with programmable nic[C]//Proceedings of the 26th Symposium on Operating Systems Principles. ACM, 2017: 137-152.

[131] LU G, GUO C, LI Y, et al. Serverswitch: a programmable and high performance platform for data center networks.[C]//Nsdi: volume 11. 2011: 2-2.

[132] TECHNOLOGIES M. Mellanox bluefield(tm) smartnic vpi[EB/OL]. 2019. http://www.mellanox.com/page/bluefield_smartnic_vpi?mtag=smartnic_vpi1.

[133] CAULFIELD A M, CHUNG E S, PUTNAM A, et al. A cloud-scale acceleration architecture [C]//Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on. IEEE, 2016: 1-13.

[134] OUYANG J, LUO H, WANG Z, et al. Fpga implementation of gzip compression and decompression for idc services[C]//2010 International Conference on Field-Programmable Technology. IEEE, 2010: 265-268.

[135] HEMSOTH N. Baidu takes fpga approach to accelerating sql at scale[EB/OL]. 2016. https://www.nextplatform.com/2016/08/24/baidu-takes-fpga-approach-accelerating-big-sql/.

[136] OUYANG J, LIN S, QI W, et al. Sda: Software-defined accelerator for large-scale dnn systems[C]//2014 IEEE Hot Chips 26 Symposium (HCS). IEEE, 2014: 1-23.

[137] OUYANG J. Xpu: A programmable fpga accelerator for diverse workloads[C]//2017 IEEE Hot Chips 29 Symposium. 2017.

[138] NAOUS J, GIBB G, BOLOUKI S, et al. Netfpga: reusable router architecture for experimental research[C]//Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow. ACM, 2008: 1-7.

[139] OVTCHAROV K, RUWASE O, KIM J Y, et al. Accelerating deep convolutional neural networks using specialized hardware[M/OL]. Microsoft Research, 2015. https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/.

[140] ZHANG J, XIONG Y, XU N, et al. The feniks fpga operating system for cloud computing [C]//Proceedings of the 8th Asia-Pacific Workshop on Systems. ACM, 2017: 22.

[141] KHAWAJA A, LANDGRAF J, PRAKASH R, et al. Sharing, protection, and compatibility for

reconfigurable fabric with amorphos[C]//13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). 2018: 107-127.

[142] CORPORATION I. Intel quickassist technology[EB/OL]. 2019. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html.

[143] FOWERS J, KIM J Y, BURGER D, et al. A scalable high-bandwidth architecture for lossless compression on fpgas[C]//2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines. IEEE, 2015: 52-59.

[144] OVTCHAROV K, RUWASE O, KIM J Y, et al. Toward accelerating deep learning at scale using specialized hardware in the datacenter[C]//2015 IEEE Hot Chips 27 Symposium (HCS). IEEE, 2015: 1-38.

[145] CHUNG E, FOWERS J, OVTCHAROV K, et al. Serving dnns in real time at datacenter scale with project brainwave[J]. IEEE Micro, 2018, 38(2):8-20.

[146] FOWERS J, OVTCHAROV K, PAPAMICHAEL M, et al. A configurable cloud-scale dnn processor for real-time ai[C]//Proceedings of the 45th Annual International Symposium on Computer Architecture. IEEE Press, 2018: 1-14.

[147] CALDER B, WANG J, OGUS A, et al. Windows azure storage: a highly available cloud storage service with strong consistency[C]//Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. ACM, 2011: 143-157.

[148] CAULFIELD A, CHUNG E, PUTNAM A, et al. A cloud-scale acceleration architecture [C/OL]//Proceedings of the 49th annual ieee/acm international symposium on microarchitecture ed. IEEE Computer Society, 2016. https://www.microsoft.com/en-us/research/publication/configurable-cloud-acceleration/.

[149] DALTON M, SCHULTZ D, ADRIAENS J, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization[C]//15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18). Renton, WA: USENIX Association, 2018: 373-387.

[150] SCHLAEGER C. Aws ec2 virtualization: Introducing nitro[EB/OL]. 2018. http://aws-de-media.s3.amazonaws.com/images/AWS_Summit_2018/June7/Alexandria/Introducing-Nitro.pdf.

[151] GREGG B. Aws ec2 virtualization 2017: Introducing nitro[EB/OL]. 2017. http://www.brendangregg.com/blog/2017-11-29/aws-ec2-virtualization-2017.html.

[152] BARR J. Amazon ec2 update －additional instance types, nitro system, and cpu options [EB/OL]. 2018. https://aws.amazon.com/blogs/aws/amazon-ec2-update-additional-instance-types-nitro-system-and-cpu-options/.

[153] WHITWAM R. Amazon buys secretive chip maker annapurna labs for 350 million dollars

[EB/OL]. 2015. http://www.extremetech.com/computing/198140-amazon-buys-secretive-chip-maker-annapurna-labs-for-350-million.

[154] ATHER A. 2 million packets per second on a public cloud instance[EB/OL]. 2016. http://techblog.cloudperf.net/2016/05/2-million-packets-per-second-on-public.html.

[155] ATHER A. 3 million storage iops on aws cloud instance[EB/OL]. 2017. http://techblog.cloudperf.net/2017/04/3-million-storage-iops-on-aws-cloud.html.

[156] LI B, TAN K, LUO L L, et al. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware[C]//Proceedings of the 2016 ACM SIGCOMM Conference. ACM, 2016: 1-14.

[157] ECLYPSIUM. The missing security primer for bare metal cloud services[EB/OL]. 2019. https://eclypsium.com/2019/01/26/the-missing-security-primer-for-bare-metal-cloud-services/.

[158] Towards converged smartnic architecture for bare metal and public clouds[Z].

[159] 阿里云高级技术专家陈静. 阿里云开发智能网卡的动机、功能框架和软转发程序[EB/OL]. 2018. https://yq.aliyun.com/articles/604505.

[160] 阿里云. 阿里云弹性裸金属服务器-神龙架构（X-Dragon）揭秘[EB/OL]. 2018. https://m.aliyun.com/yunqi/articles/594276.

[161] 华为技术有限公司. 华为 SD100 智能网卡技术白皮书[EB/OL]. 2017. https://support.huawei.com/enterprise/zh/servers/sd100-pid-22040214.

[162] 华为技术有限公司. 华为 IN200 智能网卡用户指南[EB/OL]. 2018. https://support.huawei.com/enterprise/zh/servers/in500-solution-pid-23507369.

[163] 华为技术有限公司. 华为发布智能加速引擎部件，全面加速企业应用[EB/OL]. 2018. https://www.huawei.com/cn/press-events/news/2018/10/intelligent-acceleration-engine-series.

[164] DONG Y, YANG X, LI J, et al. High performance network virtualization with sr-iov[J]. Journal of Parallel and Distributed Computing, 2012, 72(11):1471-1480.

[165] RUSSELL R. virtio: towards a de-facto standard for virtual i/o devices[J]. ACM SIGOPS Operating Systems Review, 2008, 42(5):95-103.

[166] NISHTALA R, FUGAL H, GRIMM S, et al. Scaling memcache at facebook[C]//NSDI '13. 2013.

[167] ALIZADEH M, YANG S, SHARIF M, et al. pfabric: Minimal near-optimal datacenter transport[C]//ACM SIGCOMM Computer Communication Review: volume 43. ACM, 2013: 435-446.

[168] PFAFF B, PETTIT J, KOPONEN T, et al. The design and implementation of open vswitch. [C]//NSDI: volume 15. 2015: 117-130.

[169] ANDERSON J W, BRAUD R, KAPOOR R, et al. xomb: extensible open middleboxes with

commodity servers[C]//Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems. ACM, 2012: 49-60.

[170] SEKAR V, EGI N, RATNASAMY S, et al. Design and implementation of a consolidated middlebox architecture[C]//Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12). 2012: 323-336.

[171] HWANG J, RAMAKRISHNAN K K, WOOD T. Netvm: High performance and flexible networking using virtualization on commodity platforms[J]. IEEE Transactions on Network and Service Management, 2015, 12(1):34-47.

[172] RAM K K, COX A L, CHADHA M, et al. Hyper-switch: A scalable software virtual switching architecture[C]//Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13). 2013: 13-24.

[173] EISENBUD D E, YI C, CONTAVALLI C, et al. Maglev: A fast and reliable software network load balancer[C]//13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16). 2016: 523-535.

[174] SUN C, BI J, ZHENG Z, et al. Nfp: Enabling network function parallelism in nfv[C]// Proceedings of the Conference of the ACM Special Interest Group on Data Communication. ACM, 2017: 43-56.

[175] SHAH N, PLISHKER W, RAVINDRAN K, et al. Np-click: A productive software development approach for network processors[J]. IEEE Micro, 2004, 24(5):45-54.

[176] SHERRY J, GAO P X, BASU S, et al. Rollback-recovery for middleboxes[C]//ACM SIGCOMM Computer Communication Review: volume 45. ACM, 2015: 227-240.

[177] LOCKWOOD J W, MCKEOWN N, WATSON G, et al. Netfpga–an open platform for gigabit-rate network switching and routing[C]//2007 IEEE International Conference on Microelectronic Systems Education (MSE'07). IEEE, 2007: 160-161.

[178] RUBOW E, MCGEER R, MOGUL J, et al. Chimpp: A click-based programming and simulation environment for reconfigurable networking hardware[C]//Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems. ACM, 2010: 36.

[179] LAVASANI M, DENNISON L, CHIOU D. Compiling high throughput network processors [C]//Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays. ACM, 2012: 87-96.

[180] KULKARNI C, BREBNER G, SCHELLE G. Mapping a domain specific language to a platform fpga[C]//Proceedings of the 41st annual Design Automation Conference. ACM, 2004: 924-927.

[181] SCHELLE G, GRUNWALD D. Cusp: a modular framework for high speed network applica-

tions on fpgas[C]//Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays. ACM, 2005: 246-257.

[182] RINTA-AHO T, KARLSTEDT M, DESAI M P. The click2netfpga toolchain[C]//Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12). 2012: 77-88.

[183] RUAN Z, HE T, LI B, et al. St-accel: A high-level programming platform for streaming applications on fpga[C]//Proc. 26th IEEE Int. Symp. Field-Programm. Custom Comput. Mach.(FCCM). 2018: 9-16.

[184] BERNSTEIN A. Analysis of programs for parallel processing[J/OL]. IEEE Transactions on Electronic Computers, 1966, EC-15(5):757-763. DOI: 10.1109/PGEC.1966.264565.

[185] The OpenCL Specifications ver 2.1.[Z].

[186] Dell networking s6000 spec sheet[EB/OL]. http://dell.com/.

[187] LEE J, LEE S, LEE J, et al. Flosis: a highly scalable network flow capture system for fast retrieval and storage efficiency[C]//2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15). 2015: 445-457.

[188] PAGH R, RODLER F F. Cuckoo hashing[J]. Algorithms - ESA 2001. Lecture Notes in Computer Science 2161, 2001.

[189] JIANG W. Scalable ternary content addressable memory implementation using fpgas[C]// Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems. IEEE Press, 2013: 71-82.

[190] Ethernet switch series[Z]. 2013.

[191] BARBETTE T, SOLDANI C, MATHY L. Fast userspace packet processing[C]//2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS). IEEE, 2015: 5-16.

[192] Strongswan ipsec-based vpn[Z].

[193] Linux virtual server[Z].

[194] MOON S W, REXFORD J, SHIN K G. Scalable hardware priority queue architectures for high-speed packet switches[J]. IEEE Transactions on Computers, 2000.

[195] BAI W, CHEN L, CHEN K, et al. Enabling {ECN} in multi-service multi-queue data centers[C]//13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16). 2016: 537-549.

[196] EICKEN T, CULLER D E, GOLDSTEIN S C, et al. Active messages: a mechanism for integrated communication and computation[C]//Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on. IEEE, 1992: 256-266.

[197] FITZPATRICK B. Distributed caching with memcached[J]. Linux journal, 2004, 2004(124):

5.

[198] SHAO B, WANG H, LI Y. Trinity: A distributed graph engine on a memory cloud[C]// Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013: 505-516.

[199] XIAO W, XUE J, MIAO Y, et al. TuX2: Distributed graph computation for machine learning [C]//NSDI '17. 2017.

[200] LI J, MICHAEL E, PORTS D R K. Eris: Coordination-free consistent transactions using in-network concurrency control[C]//SOSP '17. 2017.

[201] ATIKOGLU B, XU Y, FRACHTENBERG E, et al. Workload analysis of a large-scale key-value store[C]//ACM SIGMETRICS Performance Evaluation Review: volume 40. ACM, 2012: 53-64.

[202] PERRY J, OUSTERHOUT A, BALAKRISHNAN H, et al. Fastpass: A centralized zero-queue datacenter network[C]//ACM SIGCOMM Computer Communication Review: volume 44. ACM, 2014: 307-318.

[203] ZHU Y, KANG N, CAO J, et al. Packet-level telemetry in large datacenter networks[C]// ACM SIGCOMM Computer Communication Review: volume 45. ACM, 2015: 479-491.

[204] GHARACHORLOO K, GUPTA A, HENNESSY J. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors: volume 20[M]. ACM, 1992.

[205] HAN S, JANG K, PARK K, et al. Packetshader: a GPU-accelerated software router[C]// ACM SIGCOMM Computer Communication Review: volume 40. ACM, 2010: 195-206.

[206] ZHANG K, WANG K, YUAN Y, et al. Mega-KV: a case for gpus to maximize the throughput of in-memory key-value stores[J]. Proceedings of the VLDB Endowment, 2015, 8(11):1226-1237.

[207] NARULA N, CUTLER C, KOHLER E, et al. Phase reconciliation for contended in-memory transactions.[C]//OSDI '14: volume 14. 2014: 511-524.

[208] SZEPESI T, WONG B, CASSELL B, et al. Designing a low-latency cuckoo hash table for write-intensive workloads using rdma[C]//First International Workshop on Rack-scale Computing. 2014.

[209] MITCHELL C, GENG Y, LI J. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store[C]//USENIX ATC '13. 2013: 103-114.

[210] BRESLOW A D, ZHANG D P, GREATHOUSE J L, et al. Horton tables: fast hash tables for in-memory data-intensive computing[C]//USENIX ATC '16. 2016.

[211] SIVARAMAN A, CHEUNG A, BUDIU M, et al. Packet transactions: High-level programming for line-rate switches[C]//Proceedings of the ACM SIGCOMM 2016 Conference. ACM, 2016: 15-28.

[212] COUNCIL T. tpc-c benchmark, revision 5.11[M]. Feb, 2010.

[213] PAGH R, RODLER F F. Cuckoo hashing[J]. Journal of Algorithms, 2004, 51(2):122-144.

[214] HERLIHY M, SHAVIT N, TZAFRIR M. Hopscotch hashing[C]//International Symposium on Distributed Computing. Springer, 2008: 350-364.

[215] BONWICK J, et al. The slab allocator: An object-caching kernel memory allocator.[C]// USENIX summer: volume 16. Boston, MA, USA, 1994.

[216] SATISH N, KIM C, CHHUGANI J, et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort[C]//Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010: 351-362.

[217] BLOTT M, KARRAS K, LIU L, et al. Achieving 10gbps line-rate key-value stores with FPGAs[C]//The 5th USENIX Workshop on Hot Topics in Cloud Computing. San Jose, CA: USENIX, 2013.

[218] BLOTT M, LIU L, KARRAS K, et al. Scaling out to a single-node 80gbps memcached server with 40terabytes of memory.[C]//HotStorage '15. 2015.

[219] COOPER B F, SILBERSTEIN A, TAM E, et al. Benchmarking cloud serving systems with YCSB[C]//Proceedings of the 1st ACM symposium on Cloud computing. ACM, 2010: 143-154.

[220] JIN X, LI X, ZHANG H, et al. NetCache: Balancing Key-Value Stores with Fast In-Network Caching[C]//SOSP '17. 2017.

[221] JIN X, LI X, ZHANG H, et al. Netchain: Scale-free sub-rtt coordination[C]//15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18). USENIX Association, 2018: 35-49.

[222] KAUFMANN A, PETER S, SHARMA N K, et al. High performance packet processing with flexnic[C]//Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems. 2016.

[223] WU M, YANG F, XUE J, et al. Gram: scaling graph computation to the trillions[C]// Proceedings of the Sixth ACM Symposium on Cloud Computing. ACM, 2015: 408-421.

[224] INTEL D. Data plane development kit[Z]. 2014.

[225] LIANG W, YIN W, KANG P, et al. Memory efficient and high performance key-value store on FPGA using cuckoo hashing[C]//2016 26th International Conference on Field Programmable Logic and Applications (FPL). 2016: 1-4.

[226] ISTVÁN Z, ALONSO G, BLOTT M, et al. A flexible hash table design for 10gbps key-value stores on fpgas[C]//23rd International Conference on Field programmable Logic and Applications. IEEE, 2013: 1-8.

[227] CHALAMALASETTI S R, LIM K, WRIGHT M, et al. An FPGA memcached appliance

[C]//Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays (FPGA). ACM, 2013: 245-254.

[228] ISTVÁN Z, ALONSO G, BLOTT M, et al. A hash table for line-rate data processing[J]. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2015, 8(2):13.

[229] LAVASANI M, ANGEPAT H, CHIOU D. An FPGA-based in-line accelerator for memcached [J]. IEEE Computer Architecture Letters, 2014, 13(2):57-60.

[230] TOKUSASHI Y, MATSUTANI H. A multilevel nosql cache design combining in-nic and in-kernel caches[C]//High-Performance Interconnects (HOTI '16). IEEE, 2016: 60-67.

[231] LI X, SETHI R, KAMINSKY M, et al. Be fast, cheap and in control with switchkv.[C]// NSDI. 2016: 31-44.

[232] ESCRIVA R, WONG B, SIRER E G. HyperDex: A distributed, searchable key-value store [J]. ACM SIGCOMM Computer Communication Review, 2012, 42(4):25-36.

[233] KEJRIWAL A, GOPALAN A, GUPTA A, et al. SLIK: Scalable low-latency indexes for a key-value store[C]//USENIX ATC '16. 2016.

[234] LIN X, CHEN Y, LI X, et al. Scalable kernel tcp design and implementation for short-lived connections[C]//ACM SIGPLAN Notices: volume 51. ACM, 2016: 339-352.

[235] HAN S, MARSHALL S, CHUN B G, et al. Megapipe: A new programming interface for scalable network i/o.[C]//OSDI: volume 12. 2012: 135-148.

[236] YASUKATA K, HONDA M, SANTRY D, et al. Stackmap: Low-latency networking with the os stack and dedicated nics.[C]//USENIX Annual Technical Conference. 2016: 43-56.

[237] REESE W. Nginx: the high-performance web server and reverse proxy[J]. Linux Journal, 2008, 2008(173):2.

[238] REESE W. Nginx: the high-performance web server and reverse proxy[J]. Linux Journal, 2008, 2008(173):2.

[239] CARLSON J L. Redis in action[M]. Manning Publications Co., 2013.

[240] LIPP M, SCHWARZ M, GRUSS D, et al. Meltdown: Reading kernel memory from user space[C]//27th {USENIX} Security Symposium ({USENIX} Security 18). 2018: 973-990.

[241] CORBET J. Kaiser: hiding the kernel from user space[J/OL]. 2017. https://lwn.net/Articles /738975/.

[242] Thread pools in nginx boost performance 9x![J/OL]. 2015. https://www.nginx.com/blog/thr ead-pools-boost-performance-9x/.

[243] ZHU Y, ERAN H, FIRESTONE D, et al. Congestion control for large-scale rdma deployments[C]//ACM SIGCOMM Computer Communication Review: volume 45. ACM, 2015: 523-536.

[244] MITTAL R, DUKKIPATI N, BLEM E, et al. Timely: Rtt-based congestion control for the

datacenter[C]//ACM SIGCOMM Computer Communication Review: volume 45. ACM, 2015: 537-550.

[245] THOMPSON K, MILLER G J, WILDER R. Wide-area internet traffic patterns and characteristics[J]. IEEE network, 1997, 11(6):10-23.

[246] CORBET J. Large receive offload[EB/OL]. 2007. https://lwn.net/Articles/243949/.

[247] LU Y, CHEN G, LI B, et al. Multi-path transport for RDMA in datacenters[C]//15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18). Renton, WA: USENIX Association, 2018: 357-371.

[248] PFEFFERLE J, STUEDI P, TRIVEDI A, et al. A hybrid i/o virtualization framework for rdma-capable network interfaces[C]//ACM SIGPLAN Notices: volume 50. ACM, 2015: 17-30.

[249] ZHUO D, ZHANG K, ZHU Y, et al. Slim: Os kernel support for a low-overhead container overlay network[C]//16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). Boston, MA: USENIX Association, 2019.

[250] ROGHANCHI S, ERIKSSON J, BASU N. ffwd: delegation is (much) faster than you think [C]//Proceedings of the 26th Symposium on Operating Systems Principles. ACM, 2017: 342-358.

[251] KIM D, YU T, LIU H H, et al. Freeflow: Software-based virtual rdma networking for containerized clouds[C]//16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). Boston, MA: USENIX Association, 2019.

[252] PESTEREV A, STRAUSS J, ZELDOVICH N, et al. Improving network connection locality on multicore systems[C]//Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012: 337-350.

[253] SOARES L, STUMM M. Flexsc: Flexible system call scheduling with exception-less system calls[C]//Proceedings of the 9th USENIX conference on Operating systems design and implementation. USENIX Association, 2010: 33-46.

[254] THADANI M N, KHALIDI Y A. An efficient zero-copy i/o framework for unix[M]. Sun Microsystems Laboratories, 1995.

[255] CHU H K J. Zero-copy tcp in solaris[C]//Proceedings of the 1996 annual conference on USENIX Annual Technical Conference. Usenix Association, 1996: 21-21.

[256] CORBET J. Zero-copy networking[J/OL]. 2017. https://lwn.net/Articles/726917/.

[257] DUNKELS A. Design and implementation of the lwip tcp/ip stack[J]. Swedish Institute of Computer Science, 2001, 2:77.

[258] NEUGEBAUER R, ANTICHI G, ZAZO J F, et al. Understanding pcie performance for end host networking[C]//Proceedings of the 2018 Conference of the ACM Special Interest Group

on Data Communication. ACM, 2018: 327-341.

[259] KAUFMANN A, STAMLER T, PETER S, et al. Tas: Tcp acceleration as an os service [C/OL]//EuroSys '19: Proceedings of the Fourteenth EuroSys Conference 2019. New York, NY, USA: ACM, 2019: 24:1-24:16. http://doi.acm.org/10.1145/3302424.3303985.

[260] Apache http server[EB/OL]. 2019. https://httpd.apache.org/.

[261] Fastcgi process manager for php[EB/OL]. 2019. https://php-fpm.org.

[262] Python wsgi http server for unix[EB/OL]. 2019. https://gunicorn.org.

[263] vsftpd[EB/OL]. 2019. https://security.appspot.com/vsftpd.html.

[264] WIKIPEDIA. iscsi host adapter (hba)[EB/OL]. 2019. https://en.wikipedia.org/wiki/Host_adapter.

[265] CLEMENTS A T, KAASHOEK M F, ZELDOVICH N, et al. The scalable commutativity rule: Designing scalable software for multicore processors[J]. ACM Transactions on Computer Systems (TOCS), 2015, 32(4):10.

[266] HINTJENS P. Zeromq: messaging for many applications[M]. " O'Reilly Media, Inc.", 2013.

[267] RABBITMQ A. Rabbitmq-messaging that just works[J]. URL: https://www.rabbitmq.com, 2017.

[268] SEWELL P, SARKAR S, OWENS S, et al. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors[J]. Communications of the ACM, 2010, 53(7):89-97.

[269] CORPORATION I. Intel 64 and ia-32 architectures software developer manual, volume 3 [Z]. 2019.

[270] MITTAL R, SHPINER A, PANDA A, et al. Revisiting network support for rdma[J]. arXiv preprint arXiv:1806.08159, 2018.

[271] CORBET J. Tcp connection repair[J/OL]. 2012. https://lwn.net/Articles/495304/.

[272] ARISTA. Arista 7060x series[EB/OL]. 2019. https://www.arista.com/en/products/7060x-series.

[273] rpclib - modern msgpack-rpc for c++[EB/OL]. 2019. http://rpclib.net.

[274] PANDA A, HAN S, JANG K, et al. Netbricks: Taking the v out of nfv.[C]//OSDI. 2016: 203-216.

[275] KAMINSKY A K M, ANDERSEN D G. Design guidelines for high performance rdma systems[C]//2016 USENIX Annual Technical Conference. 2016: 437.

[276] TECHNOLOGIES M. Mellanox innova(tm)-2 flex open programmable smartnic[EB/OL]. 2019. http://www.mellanox.com/page/products_dyn?product_family=276&mtag=programmable_adapter_cards_innova2flex.

[277] LU Y, CHEN G, RUAN Z, et al. Memory efficient loss recovery for hardware-based transport in datacenter[C]//Proceedings of the First Asia-Pacific Workshop on Networking. ACM,

2017: 22-28.

[278] PHOTHILIMTHANA P M, LIU M, KAUFMANN A, et al. Floem: A programming system for nic-accelerated network applications[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad, CA: USENIX Association, 2018: 663-679.

[279] WONG H, BETZ V, ROSE J. Comparing fpga vs. custom cmos and the impact on processor microarchitecture[C]//Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays. ACM, 2011: 5-14.

[280] VISSERS K. Keynote 2: Versal: The new xilinx adaptive compute acceleration platforms (acap)[C]//2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3). IEEE, 2018: 10-10.

[281] VISSERS K. Versal: The xilinx adaptive compute acceleration platform (acap)[C]// Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2019: 83-83.

[282] GAIDE B, GAITONDE D, RAVISHANKAR C, et al. Xilinx adaptive compute acceleration platform: Versal tm architecture[C]//Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2019: 84-93.

[283] SWARBRICK I, GAITONDE D, AHMAD S, et al. Network-on-chip programmable platform in versal tm acap architecture[C]//Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2019: 212-221.

[284] WU C, FALEIRO J M, LIN Y, et al. Anna: A kvs for any scale[J].

[285] KIM D, MEMARIPOUR A, BADAM A, et al. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems[C]//Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. ACM, 2018: 297-312.

[286] TRADER T. Why nvidia bought mellanox: ‘future datacenters will be like high performance computers’[EB/OL]. 2019. https://www.hpcwire.com/2019/03/14/why-nvidia-bought-mellanox-future-datacenters-will-belike-high-performance-computers/.

[287] DULLOOR S R, ROY A, ZHAO Z, et al. Data tiering in heterogeneous memory systems [C]//Proceedings of the Eleventh European Conference on Computer Systems. ACM, 2016: 15.

[288] GU J, LEE Y, ZHANG Y, et al. Efficient memory disaggregation with infiniswap[C]//14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17). 2017: 649-667.

[289] AGARWAL N, WENISCH T F. Thermostat: Application-transparent page management for

two-tiered main memory[C]//ACM SIGARCH Computer Architecture News: volume 45. ACM, 2017: 631-644.

[290] LAMPORT L. Fast paxos[J]. Distributed Computing, 2006, 19(2):79-103.

[291] KEMME B, PEDONE F, ALONSO G, et al. Processing transactions over optimistic atomic broadcast protocols[C]//Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on. IEEE, 1999: 424-431.

[292] MORARU I, ANDERSEN D G, KAMINSKY M. There is more consensus in egalitarian parliaments[C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 358-372.

[293] PEDONE F, SCHIPER A. Optimistic atomic broadcast[C]//International Symposium on Distributed Computing. Springer, 1998: 318-332.

[294] PORTS D R, LI J, LIU V, et al. Designing distributed systems using approximate synchrony in data center networks.[C]//NSDI. 2015: 43-57.

[295] LI J, MICHAEL E, SHARMA N K, et al. Just say no to paxos overhead: Replacing consensus with network ordering.[C]//OSDI. 2016: 467-483.

[296] DANG H T, SCIASCIA D, CANINI M, et al. Netpaxos: Consensus at network speed[C]//Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research. ACM, 2015: 5.

[297] DANG H T, CANINI M, PEDONE F, et al. Paxos made switch-y[J]. ACM SIGCOMM Computer Communication Review, 2016, 46(1):18-24.

[298] DANG H T, BRESSANA P, WANG H, et al. Network hardware-accelerated consensus[J]. arXiv preprint arXiv:1605.05619, 2016.

[299] YANG T, GIFFORD R, HAEBERLEN A, et al. The synchronous data center[C/OL]//HotOS '19: Proceedings of the Workshop on Hot Topics in Operating Systems. New York, NY, USA: ACM, 2019: 142-148. http://doi.acm.org/10.1145/3317550.3321442.

[300] ZUO G. Near-optimal total order message scattering in data center networks[J]. 2017.

[301] CORBETT J C, DEAN J, EPSTEIN M, et al. Spanner: Google's globally distributed database [J]. ACM Transactions on Computer Systems (TOCS), 2013, 31(3):8.

致　谢

我要感谢我的母校中国科学技术大学和微软亚洲研究院能给我宝贵的学习机会，让我能在联合培养博士期间接触到世界领先的可编程网卡实验平台和数据中心应用场景，在数据中心系统领域开展前沿研究。

我要感谢中国科学技术大学的导师陈恩红教授。从本科四年级开始的六年里，陈恩红老师一直支持我在微软的联合培养实习，帮助我确定了研究方向和博士课题。读博期间，陈老师帮助我确定培养计划，资助我参加国际学术会议，推荐我申请微软学者奖、国家奖学金等诸多奖励，还帮助我修改开题报告和毕业论文。无论是生活还是科研，陈老师都竭尽所能给予我支持和帮助，让我免除后顾之忧，集中注意力于科研中的学术问题。我能取得一些小小的学术成果，不仅有赖于陈老师在大方向上的指导，也跟陈老师在背后默默的支持和帮助是密不可分的。衷心地感谢陈恩红老师对我的支持和帮助。

我要感谢我在微软亚洲研究院的导师，首席研究员张霖涛博士。相处的三年里，张霖涛老师带领我走进系统研究的大门，不仅教会了我计算机系统的知识和思维方式，而且锻炼了我独立思考、发现问题和主持研究的能力。张霖涛老师指导我完成了第二个研究项目 KV-Direct，在键值存储领域若干可能的创新点里，选定了加速内存数据结构访问这个最能突出可编程网卡作用的创新点。在我和阮震元同学合作实现的过程中，他帮助我们提炼总结系统设计与优化技巧，从头到尾修改论文、讲稿，并发表在系统领域的顶级学术会议上，让我在博士中期有较好的科研成果。随后，张霖涛老师给我足够的空间让我独立思考、自由探索，带领我开阔视野，培养对系统的大局观，并帮助我招聘实习生来合作实现我的创新，研究了几个新的课题，论文被 SIGCOMM 等会议接收。不管是组内还是组外的报告，张霖涛老师总是能敏锐地理解并提出深刻的问题。他给我在知名教授面前讲故事和听取反馈的宝贵机会，提醒我不要陷入技术细节而忘记听众的背景和系统的大局。张霖涛老师指导我理清了博士期间研究的主线，认识到自己所做研究更深刻的内涵、更广阔的外延以及与高影响力工作的差距。张霖涛老师带我在微软总部进行了第一次美国之旅，平时经常给我分享系统研究、职场和人生的经验，是我的良师益友。衷心地感谢张霖涛老师对我的指导和帮助。

我要感谢我在微软亚洲研究院的前导师，前资深研究员谭焜博士。谭焜老师是我的科研启蒙导师，不仅教会了我计算机网络的知识和思维方式，而且教会了我科研“分析型思考”的方法论和做学问“去伪存真”的态度。谭焜老师确立了网络研究组在数据中心领域的研究方向，搭建了世界领先的数据中心网络和可编程网卡实验平台。谭焜老师手把手指导我完成了第一个研究项目 ClickNP。他提

出了用可编程网卡加速网络功能这个学术问题，确定了高级语言编程的基本框架和技术路线，帮助我撰写论文，并发表在网络领域的顶级学术会议上，让我在科研上有一个较高的起点。谭焜老师一方面让我给不同领域的研究员讲解以锻炼大局观，另一方面注重细节，在组会上讨论代码风格、实验数据和讲稿字句。在我思维过于发散时，他及时让我收敛得出结论，让我能持续高效产出。ClickNP 项目完成后，我在 FPGA 编程和系统、网络方向之间纠结时，谭焜老师指导我定位在系统领域，专注于能产生实际影响的项目。谭焜老师还给我分享了很多对研究的哲学思考，亦是我的良师益友。衷心地感谢谭焜老师对我的指导和帮助。

我要感谢与我合作论文的老师和同学们。他们除了我的导师以外，还有微软亚洲研究院的研究员和实习生同学。在 ClickNP 项目中，我要感谢研究员罗腊咏博士在 FPGA 编程框架方面的早期探索，上海交通大学彭燕庆同学、中国科学技术大学罗人千同学合作开发编译器、网络元件和应用，资深研究员徐宁仪博士、资深研究员熊勇强博士、研究员程鹏博士的讨论与帮助，以及北京航空航天大学的贺同同学对 FPGA 与 CPU 间通信管道的开发。在 KV-Direct 项目中，我要感谢共同第一作者、中国科学技术大学阮震元同学与我一起设计和实现系统，北京航空航天大学肖文聪同学撰写引言，中国科学技术大学陆元伟同学的讨论与帮助，资深研究员熊勇强博士和首席硬件工程师 Andrew Putnam 博士的讨论与硬件实验环境的支持。在 SocksDirect 项目中，我要感谢共同第一作者、中国科学技术大学崔天一同学与我一起设计和实现系统，研究员白巍博士撰写引言、梳理论文逻辑，中国科学技术大学王子博同学实现了第一版系统的原型。在 TOMS 项目中，我要感谢中国科学技术大学左格非同学与我一起设计和实现系统，研究员白巍博士与我讨论修改了论文的大部分内容。感谢尚未发表的研究项目中，微软亚洲研究院系统组研究员任晶磊博士、陈亮博士，电子科技大学的王阳同学、KAIST 的 Taekyung Heo 同学以及陆元伟、肖文聪、阮震元、崔天一、李弈帅、曹士杰等同学的合作与帮助。还要特别感谢微软亚洲研究院研究员陈果博士、张建松博士与我多次合作论文。陈果博士的 FUSO 是我作为第四作者在学术生涯的第一篇论文，严谨的治学态度令我受益终生。张建松博士指导我进行 FPGA 开发，我有幸参与了张建松博士提出的 Feniks FPGA OS 和 SSD 加速项目。衷心感谢所有博士论文合作者对我的指导与帮助，没有你们，我是不可能做出这些成果的。

除了以上论文合作者外，我还要感谢微软亚洲研究院副院长周礼栋博士，系统组刘云新博士、张宸博士，网络组舒然博士、牛治雄博士，南京大学的王晓亮副教授在科研项目中给予我的指导和帮助。感谢微软美国总部的 Andrew Putnam，Tanj Bennett，Derek Chiou，Qi Luo，Daniel Firestone 和微软亚洲研究院的崔巍、王文强等同事对我的支持与帮助。感谢首席研究员张永光博士，给我进入中国科学技术大学和微软亚洲研究院联合培养的机会。感谢会议论文的各位匿名审稿

人和博士论文评审委员会的宝贵意见。感谢在学术会议和面试中给予我宝贵指导的老师、专家和同学。

我要感谢在中国科学技术大学和微软亚洲研究院遇到的老师、同学和朋友们。特别是以微软亚洲研究院联合培养博士生为核心的“饭团”同学，谢谢你们陪伴我度过丰富多彩的博士生活。感谢微软亚洲研究院学术合作部的同事们在我的博士培养过程与实习生活中的帮助。感谢中国科学技术大学 Linux 用户协会的张焕杰老师和各位小伙伴们，让我在本科期间有机会开发和运维各种网络服务，锻炼了计算机系统的技术能力和处理故障的心理素质，培养了我对云计算的兴趣。感谢中学的肖世康同学教我做网站，区块链领域的朋友们培养我对定制化硬件的兴趣。我要特别感谢我的女朋友，我们不仅因为科研相识，一起合作论文，她还非常支持我的学术研究和职业发展规划。我的女朋友让我在生活上变得更成熟，也使我们在一起的生活充满了快乐。她就像一束光，照到的地方就充满了光亮。

最后，我要感谢我的父母、爷爷奶奶和家人。是你们在背后默默地支持着我，让我有一个强有力的后盾。感谢我的父母和爷爷奶奶二十多年的养育之恩，以及其他家人的帮助和支持。没有你们，我不可能做出现在的成绩。你们是最伟大的，谢谢你们的付出。

在读期间发表的学术论文与取得的研究成果

已发表第一作者论文

1. B. Li, K. Tan, L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, E. Chen, "ClickNP: highly flexible and high performance network processing with reconfigurable hardware", Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16), Florianopolis, Brazil, Aug. 2016. （CCF 推荐 A 类会议）
2. B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, L. Zhang, "KV-direct: High-performance in-memory key-value store with programmable NIC", Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17), Shanghai, China, Sept. 2017. （CCF 推荐 A 类会议）
3. B. Li, T. Cui, Z. Wang, W. Bai, L. Zhang, "SocksDirect: Datacenter Sockets can be Fast and Compatible", Proceedings of the 2019 ACM SIGCOMM Conference (SIGCOMM '19), Beijing, China, Aug. 2019.（CCF 推荐 A 类会议，已被接收，将于 2019 年 8 月正式出版）

其他已发表论文

1. G. Chen, Y. Lu, Y. Meng, B. Li, K. Tan, D. Pei, P. Cheng, L. Luo, Y. Xiong, X. Wang, Y. Zhao, "FUSO: Fast Multi-Path loss recovery for data center networks", IEEE/ACM Transactions on Networking (TON) 2018. (CCF 推荐 A 类期刊)
2. Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, T. Moscibroda, "Multi-path transport for RDMA in datacenters", NSDI 2018. (CCF 推荐 A 类会议)
3. G. Chen, Y. Lu, Y. Meng, B. Li, K. Tan, D. Pei, P. Cheng, L. Luo, Y. Xiong, X. Wang, Y. Zhao, "Fast and Cautious: Leveraging Multi-Path diversity for transport loss recovery in data center", ATC 2016. (CCF 推荐 A 类会议)
4. Z. Ruan, T. He, B. Li, P. Zhou, J. Cong, "ST-Accel: A high-level programming platform for steaming applications on FPGA", Proceedings of 26th IEEE International Symposium on Field-Programmable Custom Computing Machines 2018 (FCCM '18). (CCF 推荐 C 类会议)
5. Y. Lu, G. Chen, Z. Ruan, W. Xiao, B. Li, J. Zhang, Y. Xiong, P. Chen, E. Chen, "Memory efficient loss recovery for hardware-based transport in datacenter", Proceedings of the 1st Asia-Pacific workshop on networking 2017 (APNet '17).

6. J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, T. Moscibroda, “The Feniks FPGA Operating System for Cloud Computing”, Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys ’17).
7. B. Li, Z. Wang, T. Cui, L. Zhang, “Fast and Compatible User-Space Container Networking with Programmable NIC”, SOSP Student Research Competition 2017 (poster), Final Presentation Rounds.
8. R. Li, B. Li, G. Zhang, J. Jiang, Y. Luo, “A High-Performance and Flexible Chemical Structure & Data Search Engine Built on CouchDB & ElasticSearch”, Chinese Journal of Chemical Physics, Volume 31, Number 2.
9. J. Meng, H. Tan, C. Xu, W. Cao, L. Liu, B. Li, “Dedas: Online Task Dispatching and Scheduling with Bandwidth Constraint in Edge Computing”, Proceedings of the 2019 IEEE International Conference on Computer Communications (INFOCOM ’19). (CCF 推荐 A 类会议)

待发表论文

1. B. Li, G. Zuo, W. Bai, L. Zhang, “Efficient and Scalable Total-Order Message Scattering in Data Center Networks”.