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

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

Notice of Automatic Translation

This doctoral thesis is an automatically translated copy of the original Chinese text. The translation process was fully carried out by an AI toolchain including GPT-4, without subsequent review or verification by a human. Due to the nuances of automated translation, there might be inconsistencies, inaccuracies, or unclear passages within this version.

In case of any confusion, discrepancies, or doubts arising while reading this translated document, readers are strongly encouraged to refer to the original version of the thesis in Chinese for the most accurate comprehension. Moreover, for in-depth understanding and detailed explanations of the research work, readers can refer to the published papers associated with this thesis, which are available in English.

We appreciate your understanding and encourage you to consult the primary sources mentioned above to get the most accurate information.

University of Science and Technology of China Thesis Originality Statement

I hereby declare that the degree thesis I submitted is the result of my own research conducted under the guidance of my supervisor. Except for the parts that have been clearly marked and acknowledged, the thesis does not contain any research results that have been published or written by others. The contributions of colleagues who worked with me on this research have been clearly stated in the thesis.

Signature: _____ Date: _____

University of Science and Technology of China Thesis Authorization of Use Statement

As one of the conditions for degree application, the copyright holder of the thesis authorizes the University of Science and Technology of China to have partial usage rights to the thesis, namely: the university has the right, according to relevant regulations, to submit copies and electronic versions of the thesis to relevant national departments or institutions, to allow the thesis to be consulted and borrowed, to include the thesis in relevant databases such as the ”China Dissertation Full-Text Database” for retrieval, and to preserve and compile the thesis using reproduction methods such as photocopying, microprinting, or scanning. The content of the electronic document I submitted is consistent with the content of the paper thesis.

Theses classified as confidential will also comply with this regulation after declassification.

☑ Public ☐ Confidential (____ years)

Author’s Signature: _____ Advisor’s Signature: _____

Date: _____ Date: _____

ABSTRACT

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.

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

Finally, this dissertation proposes a co-design of programmable NICs and user-space libraries to provide kernel-bypass socket communication primitives for applications. The dissertation 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 maintains 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 the HTTP latency of web servers to 1/5.5.

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

Contents

Chapter 1 Introduction

1.1 Research Background and Significance

Data centers serve as the ”brains” of the Internet, providing the infrastructure necessary for storing vast amounts of data, performing large-scale computations, and offering Internet services. In the first decade of the 21st century, data centers primarily processed tasks that were easily parallelizable, such as websites and search engines. The rapid improvement in the performance of general-purpose processors also made the advantages of dedicated hardware less apparent. Consequently, Internet data centers often employed a large number of low-cost standard servers for construction[1].

In the past decade, the emergence of big data and artificial intelligence has altered the application load characteristics of data centers. On one hand, big data processing and machine learning workloads demand high computational power. However, due to the slowdown of Moore’s Law and the end of Dennard’s scaling law, the increase in frequency and number of multi-cores of general-purpose processors has been limited by the power wall in the past decade[2]. Therefore, the ”free lunch” of general-purpose processor performance improvement has ended, ushering in the era of architectural innovation. Customized hardware such as GPUs, FPGAs, and TPUs[3] are now widely deployed in data centers. On the other hand, big data processing and machine learning workloads require multiple nodes to work closely together, necessitating high inter-node communication bandwidth and low latency. To efficiently provide message passing and shared memory inter-process communication paradigms for distributed systems, efficient message passing needs to be implemented in the network, and high-performance shared data structure storage needs to be implemented at the storage level. Therefore, in the past decade, data center networks have evolved from 1 Gbps to 40 Gbps, and there is a trend to evolve to 100 Gbps. Dedicated interconnects between customized hardware are also becoming a trend.

At the same time, the operational mode of data centers is also undergoing a transformation towards cloudification. A handful of cloud manufacturers are gradually centralizing the computing power of data centers, each possessing millions of servers. Due to the large scale of cloud data centers, cloud service providers can, on one hand, amortize the design and tape-out costs of servers, boards, and even chips, and on the other hand, improve performance indicators and reduce costs through software optimization, thereby achieving significant economic benefits.

The cloudification of data centers means that a few cloud manufacturers maintain the basic infrastructure of data centers, and IT companies only need to rent computing, network, storage, and other resources from these cloud manufacturers as needed. In cloud data centers, different tenants share a vast pool of computing, storage, and network resources. To achieve resource sharing and performance isolation, data centers require virtualized computing, storage, and networks.

As shown in Figure 1.1, under the Infrastructure as a Service (IaaS) cloud service model, computing nodes need to provide services such as virtual networks, virtual cloud storage, and virtual local storage, while the actual network and cloud storage resources are located on independent network nodes and storage nodes. The virtual network and storage services on the computing nodes virtualize the physically dispersed network and storage resources in the data center into logically unified resources (”multi-virtual one”), akin to a large-scale computer[4].

Network and storage nodes not only need to share physical resources with virtual machines of different tenants on multiple computing nodes (”one virtual multi”), but also need to provide data processing functions and high-level abstractions. For instance, network nodes need to provide *network functions* such as firewalls, load balancing, encrypted tunnel gateways, and Network Address Translation (NAT)①; storage nodes need to perform data structure processing to provide high-level abstractions such as object storage and file system storage, and need to perform replication for disaster recovery.

In addition to persistent storage, data centers also need to provide memory data structure storage to support communication in distributed systems②.

Due to the rapid evolution of cloud services, these virtualized network and storage functions also require flexibility, programmability, and debuggability, which are traditionally often implemented by software running on general-purpose processors. Besides virtualization overhead, the overhead of traditional operating systems cannot be overlooked. The software overhead depicted in the shadowed boxes in Figure 1.1 is referred to as the ”data center tax”[1,4-5]. In the era of 1 Gbps networks and mechanical hard drives, the CPU overhead and latency introduced by network and storage

① In this article, *network function* is a proper noun, not referring to network devices such as switches and routers, but referring to functions in network infrastructure such as firewalls.

② Communication in distributed systems has two paradigms: message passing and shared memory. Message passing can be directly mapped to network communication. The shared memory paradigm is more developer-friendly, can support high availability and scalability, and needs to be implemented through message passing in distributed systems with network interconnection. However, the abstraction level of shared memory is relatively low, so most distributed systems use shared data structure storage to replace shared memory.

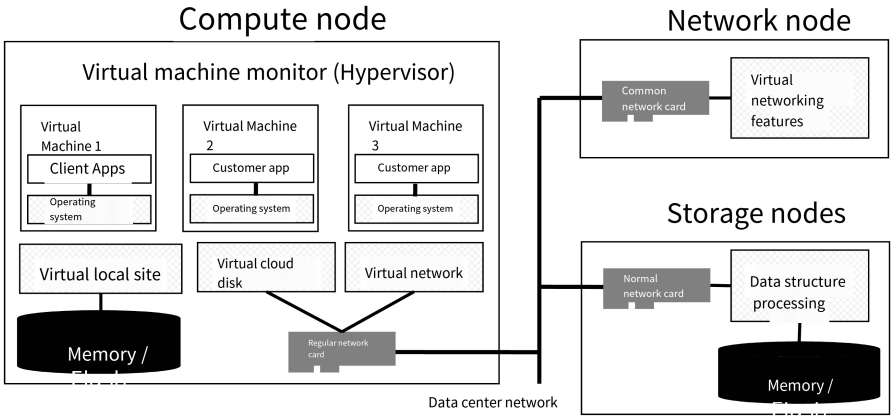


Figure 1.1 Virtualized data center architecture.

virtualization as well as the operating system's network and storage protocol stack were acceptable. With the trend of increasingly faster networks, storage, and customized computing hardware, the data center tax not only squanders significant CPU resources but also hinders applications from fully exploiting the low latency and high throughput of hardware[6]. For instance, as will be elucidated in Chapter 4, computing nodes need to allocate a portion of the CPU cores specifically for implementing network and storage virtualization, and these cores will not be available for sale to customers. Additionally, virtual networks and network functions will add tens to thousands of microseconds of latency. For comparison, the latency of the data center network itself is only a few to tens of microseconds, and the latency added by virtualization is higher than the latency of the network itself. Chapter 5 will explain that the throughput of software-implemented shared memory data structure storage is far from that of memory hardware. Chapter 6 will explain that applications generally use the socket primitives in the operating system for communication. For communication-intensive applications such as web servers, the operating system occupies a large part of the CPU time; moreover, the socket primitives implemented by the operating system have an order of magnitude higher latency than the Remote Direct Memory Access (RDMA) primitives provided by the hardware.

In summary, it is of great importance to reduce the ”data center tax” through full-stack optimization combining hardware and software for the performance and cost of modern data centers, which is also the subject of this paper.

1.2 Research Status at Home and Abroad

In order to reduce the overhead of the ”data center tax”, the academic and industrial communities have proposed many solutions, which can be roughly divided into three categories: optimizing software, utilizing new commercial hardware, and designing

new hardware.

1.2.1 Optimizing Software

Traditional network functions are implemented by dedicated devices deployed in specific locations in the data center. These dedicated network function devices are not only expensive, but also not flexible enough to support multi-tenancy in cloud services. Therefore, cloud service providers have deployed software-implemented virtual network functions. For instance, Ananta[7] is a software load balancer deployed in Microsoft data centers, used to provide cloud-scale load balancing services. Works such as RouteBricks[8] demonstrate that the speed of each server forwarding packets based on multi-core x86 CPUs can reach 10 Gbps, and capacity can be expanded by multi-core and building more network node clusters. Although software-implemented virtual switches and network functions can use a larger number of CPU cores and larger network node clusters to support higher performance, doing so will increase considerable asset and operating costs[7,9]. The profitability of cloud service providers in IaaS business is the difference between the price paid by customers for virtual machines and the cost of hosting virtual machines. Since the asset and operating costs of each server are basically determined at the time of deployment, the best way to reduce the cost of hosting virtual machines is to package more customer virtual machines on each computing node server and reduce the number of servers for network and storage nodes. Currently, the price of a physical CPU core (2 hyperthreads, i.e., 2 vCPUs) is about $0.1 per hour, i.e., the maximum potential income is about $900 per year[10]. In data centers, servers usually serve for 3 to 5 years, so the highest price of a physical CPU core during the server's life cycle can reach $4500[10]. Even considering that some CPU cores are not sold out, and the cloud often offers discounts to large customers, compared with dedicated hardware, it is quite expensive to allocate a physical CPU core specifically for virtual networks.

Most applications access the network through the socket interface provided by the operating system. The sockets of existing operating systems such as Linux were designed for low-speed networks decades ago and have high overhead in today's high-throughput, low-latency data center networks. In recent years, a lot of work has been devoted to providing high-performance sockets. The first type of work is to optimize the TCP/IP network protocol stack of the operating system kernel. However, a lot of kernel overhead still exists, which will be discussed in detail in Chapter 6. The second type of work completely bypasses the kernel TCP/IP protocol stack and imple-

ments TCP/IP in user space, known as user-mode protocol stacks. In this category, some works[11-12] propose new operating system architectures, using virtualization to ensure security and isolation. In addition to these new operating system architectures, many user-mode protocol stacks utilize high-performance packet I/O frameworks that already exist on Linux[13-15]. Among them, some user-mode protocol stacks[16-19] believe that the Linux socket API is the root of performance overhead, and thus propose new APIs, which require modifying applications. Most API changes aim to support zero copy. Some other systems[20-21] go further, believing that the abstraction level of sockets is too low, and applications should use a higher-level remote procedure call (RPC) interface. Since sockets are widely used, it is not realistic to require applications to modify the interface in many cases. Therefore, some systems in the industry[22-24] and academia[25] propose user-mode TCP/IP protocol stacks that comply with the standard socket API. These user-mode protocol stacks provide better performance than Linux, but they are still not close to the performance limit of hardware. Currently, the performance benchmark for host-to-host communication in data center networks is Remote Direct Memory Access (RDMA), and the most efficient method for inter-process communication within a host is shared memory. The host-to-host communication latency of these user-mode protocol stacks is an order of magnitude higher than RDMA, and the intra-host communication latency is one to two orders of magnitude higher than shared memory.

As a fundamental infrastructure for communication and storage in distributed systems, the exploration and advancement of Key-Value Storage systems have persistently been a primary concern in the academic and industrial systems community. The performance of early memory key-value storage systems[26] was not satisfactory. Given that distributed systems are interconnected through networks, the user clients and storage servers of key-value storage systems also need to communicate through networks, introducing the overhead of socket network protocol stacks and network virtualization discussed earlier.

To eliminate the overhead of the operating system kernel, recent key-value storage systems[27-31] employ high-performance network packet processing frameworks, poll network packets from network cards, and utilize the aforementioned user-mode lightweight network protocol stacks to process them. However, even without considering the overhead of the network, the cost of key-value storage systems for data structure processing is also high.

To reduce computing costs, a series of works[32-34] optimize locks, caches, hashes,

and memory allocation algorithms. To reduce the inter-core synchronization overhead of multiple CPU cores processing the same key, a more efficient method is for a fixed CPU core to handle each key's write operations[30].

However, due to the limitations of CPU parallelism, as will be discussed in Chapter 5, even if optimized to the extreme, each CPU core can only handle about 5 million key-value operation requests per second, far lower than the hardware performance that memory random access can provide.

In addition, the keys in real-world workloads often follow a long-tail distribution, that is, a small number of keys are accessed very frequently, and most keys are not frequently accessed. Under long-tail distribution loads, because the same key is always mapped to the same CPU core, it will lead to load imbalance among CPU cores[30].

1.2.2 Utilizing New Commercial Hardware

Due to the pressing performance demands of large-scale web services, big data processing, machine learning, and other computing and networking applications, computing acceleration devices such as Graphics Processing Units (GPUs) and network acceleration technologies such as Remote Direct Memory Access (RDMA)[35] are increasingly being deployed in data centers.

To speed up virtual networks and network functions, previous research has suggested the use of GPUs[36], Network Processors (NPs)[37-38], and hardware network switches ①[9]. GPUs were initially used primarily for graphics processing, but in recent years have been extended to other applications with massive data parallelism. GPUs are suitable for batch operations, but batch operations can lead to high latency. The history of network processors can be traced back to network switches of the 1990s. Network processors consist of a large number of embedded processor cores, each with limited processing power, and stateful connections are usually handled by fixed processor cores, thus limiting the throughput of a single connection. The main problem with hardware network switches is their lack of flexibility and insufficient lookup table capacity[9].

To reduce the overhead of operating system communication primitives, a series of work has offloaded ② part of the operating system network protocol stack to network card hardware. The TCP Offload Engine (TOE)[39] offloads part or all of the TCP/IP protocol stack to the network card. However, due to the rapid growth of general-purpose

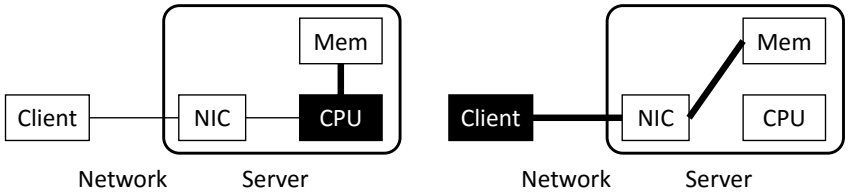
① In this paper, the term *switch* is used interchangeably with router, and the term *switch* is commonly used in data center-related literature to refer to network interconnection devices.

② In this paper, *offload* is a term that refers to implementing functions that are implemented in software on the host CPU using hardware other than the host CPU.

processor performance according to Moore's Law, the performance advantages of these dedicated hardware are limited and have only been successful in dedicated fields. The commercially successful TCP offload functions are mostly stateless offloads ①, such as offloading of TCP checksums. Infiniband[35] technology designed a new set of communication primitives, transport layer protocols, network layer protocols, and physical transmission media. The communication primitives and transport layer protocols are known as Remote Direct Memory Access (RDMA). Infiniband implements the entire network protocol stack in hardware, achieving high throughput and low latency, and is widely used in the field of high-performance computing. Since the transport layer needs to maintain the state of each connection to implement congestion control, packet retransmission, out-of-order rearrangement, etc., Infiniband is a stateful offload. In recent years, due to the hardware trends and application requirements of data centers, the story of stateful offload has begun to revive[40]. RDMA based on Infiniband technology is widely deployed in data centers[41]. In order to be compatible with the existing Ethernet in data centers, data centers do not use Infiniband physical layer networks, but strip the RDMA primitives and transport layer protocols from the Infiniband technology stack, encapsulate RDMA transport layer packets in UDP/IP packets, and transmit them through Ethernet, which is called RoCEv2[42]. Compared with the software-based TCP/IP network protocol stack, RDMA uses hardware offloading to provide ultra-low latency and near-zero CPU overhead.

RDMA communication primitives significantly differ from the socket communication primitives typically used by applications. RDMA is message-based, whereas sockets are stream-based. RDMA necessitates applications to explicitly register and manage send and receive buffers. It also requires applications to manage send, receive, and event queues, and to timely send flow control notifications to the network card. RDMA offers two types of communication primitives. The first type is two-sided operations, similar to sockets, where the sender calls send and the receiver calls recv. Unlike sockets, the RDMA receiver application needs to predeclare the messages to be received and prepare the receive buffer for the network card. The other type is one-sided operations, which provide shared memory primitives, i.e., direct read and write of remote memory, or atomic operations on remote memory. Consequently, RDMA programming is considerably more complex than socket programming. An example program that sends and receives with sockets requires only a few dozen lines of code, while the same functionality with RDMA requires hundreds of lines of code. To enable socket

① *Stateless* means that the internal storage of the network card is read-only during packet processing.



(a) Software / Bilateral RDMA. **(b)** Unilateral RDMA.

Figure 1.2 Architecture of key-value storage systems. Lines represent data paths. A key-value operation (thin line) may require multiple address-based memory accesses (thick line). The box with a black background indicates where key-value data structure processing occurs.

applications to use RDMA, works like RSocket[43-45] convert socket operations into RDMA primitives. They have similar designs, with RSocket being the most actively developed and the de facto standard for converting sockets to RDMA. However, the performance and compatibility of RSocket are not satisfactory. Chapter 6 will propose a socket system that is compatible with existing applications and can fully utilize the performance of RDMA network cards.

In addition to network communication between hosts, inter-process communication within a host is also crucial. Some work[11-12,43] allows inter-process communication within a host to bypass the RDMA network card. However, due to the limitations of the PCIe bus, the latency and throughput of the RDMA network card are much worse than the shared memory within the host. Therefore, Chapter 6 uses shared memory to implement inter-process communication within the host.

To enhance the performance of key-value storage systems, recent research[46,46-47,47] has utilized the hardware-based network protocol stack of the RDMA network card. This approach uses bilateral RDMA as the remote procedure call mechanism between the key-value storage client and server, thereby significantly increasing per-core throughput and reducing latency, as illustrated in Figure 1.2a. Although these studies have further reduced the overhead of network communication, as discussed in Section 1.2.1, these systems still rely on the server-side CPU for data structure processing, which hampers performance.

An alternative strategy is to employ unilateral RDMA, which shifts the data structure processing of the server-side CPU to the client, as shown in Figure 1.2b. The client initiates read and write requests to the server-side shared memory via unilateral RDMA, and the server-side network card directly accesses the memory without involving the CPU. However, using shared memory mode to access data structures often necessitates multiple network round trips (for instance, querying the index first and then accessing the data), which increases access latency and consumes network bandwidth. Moreover,

this mode is not suitable for write-intensive workloads. When multiple clients attempt to manipulate the same data structure (for example, allocating memory or modifying the same key-value pair), they must lock or synchronize between clients, which introduces additional latency and bandwidth overhead.

1.2.3 Designing New Hardware

As the performance improvement of general-purpose processors has hit a wall, major cloud service providers have started to explore the use of custom hardware to reduce the ”data center tax”. This means shifting the overhead of data center virtualization, operating systems, and high-level abstractions from general-purpose processors to custom hardware. The use of custom hardware is not about implementing existing software in hardware as it is, but rather refactoring existing software, dividing it into a data plane and a control plane, optimizing the data plane and implementing it in hardware, and leaving the control plane in software. Although the solution using new hardware has higher performance, it requires not only the assets and operating costs of new hardware, but also the development costs of software and hardware co-design, which are often higher than the one-time research and development (Non-Recurring Engineering, NRE) cost of simply optimizing software. In addition, compared to developing and testing software, designing, verifying, and mass-producing new hardware requires a longer R&D cycle. Programmable hardware has a certain degree of flexibility, but it is limited compared to general-purpose processors, so it requires a certain foresight of future data center application loads and infrastructure. Therefore, not all software is suitable for hardware acceleration, and it is necessary to weigh multiple factors such as cost, benefit, R&D cycle, and flexibility. Due to the massive scale of cloud services, the overhead of the ”data center tax” is also widespread and significant, so it is worth designing programmable hardware for acceleration.

To minimize the CPU cores used for virtual networks on computing nodes, cloud service providers, represented by Microsoft Azure, have deployed a programmable network card on each server in the data center to accelerate virtual networks[10]. To provide high performance while maintaining a certain degree of programmability and flexibility, the industry has proposed programmable network card architectures such as dedicated chips (ASIC), network processors (Network Processor), multi-core general-purpose processor system-on-chip (SoC), and field-programmable gate arrays (FPGA), which will be discussed in detail in Section 2.3. FPGA achieves a balance between performance and flexibility, so Microsoft uses an FPGA-based programmable network

card[48].

FPGAs have been extensively utilized in the implementation of network routers and switches. However, FPGAs are typically programmed in hardware description languages such as Verilog and VHDL. As is widely recognized, hardware description languages are challenging to debug, write, and modify, which presents a significant obstacle for software personnel to utilize FPGAs. To enhance the development efficiency of FPGAs, FPGA manufacturers offer high-level synthesis (HLS) tools[49-50] capable of compiling restricted C code into hardware modules. However, these tools merely supplement the hardware development toolchain. Programmers still need to manually insert the hardware modules generated from C language into the hardware description language project, and they must handle the communication between the FPGA and the host CPU themselves. The academic and industrial communities have proposed efficient hardware development languages such as Bluespec[51], Lime[52], and Chisel[53] [54-56], but these require developers to possess substantial hardware design knowledge. High-level synthesis tools and efficient hardware development languages can enhance the work efficiency of hardware developers, but they are still insufficient for software developers to utilize FPGAs.

In recent years, to enable software developers to use FPGA, FPGA manufacturers have proposed OpenCL-based programming toolchains[57-58], providing a GPU-like programming model. Software developers can offload kernels written in OpenCL language to FPGA. However, in this method, multiple parallelly executing kernels need to communicate through shared memory on the board, and the throughput and latency of DRAM shared memory on FPGA are not ideal, and shared memory can become a communication bottleneck. Secondly, the communication model between FPGA and CPU is similar to the GPU's batch processing model, which results in higher processing latency (about 1 millisecond), which is not suitable for network packet processing that requires microsecond-level latency. Chapter 4 of this paper will propose a modular FPGA programming framework that is available to software developers and has high performance for network packet processing. Based on this, Chapter 5 will use programmable network cards to extend the shared memory read and write primitives of RDMA to key-value operation primitives, and use programmable network cards to implement high-throughput, low-latency memory key-value storage.

1.3 Research Content and Contributions of This Paper

This paper aims to explore high-performance data center systems that are constructed on programmable network cards. It presents a system that enhances the full stack of computing, networking, and storage nodes in cloud computing data centers, utilizing FPGA programmable network cards. As illustrated in Figure 1.3, by replacing the standard network cards on the computing, networking, and storage nodes with programmable network cards, this paper implements virtual network cards and virtual networks, virtual local storage and cloud storage, and lightweight user-state runtime libraries on the computing nodes. Furthermore, the integration of hardware transport protocol communication primitives supplants the software virtualization services and operating system network protocol stack in Figure 1.1. This paper also implements the virtual network functions of network nodes and the memory data structure processing of storage nodes based on the concept of separating the data plane and the control plane, enhances the data plane performance with programmable network cards, and retains the flexibility of the original software control plane.

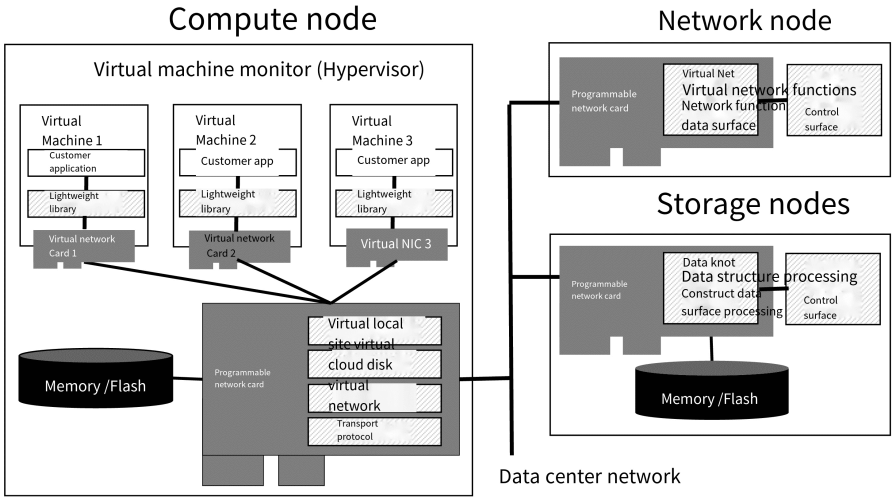


Figure 1.3 Overall architecture of the data center system based on programmable network cards.

Firstly, this paper proposes to accelerate the virtual network functions in cloud computing with programmable network cards. It introduces the first high-flexibility, high-performance network function processing platform ClickNP, accelerated by FPGA in commercial servers. It is well known that FPGA programming is not user-friendly for software engineers. To simplify FPGA programming, a C-like ClickNP language and a modular programming model are designed, and a series of optimization techniques are developed to fully exploit the massive parallelism of FPGA. The ClickNP development toolchain is implemented, which can be integrated with various commercial high-level

synthesis tools. More than 200 network elements are designed and implemented based on ClickNP, and these elements are used to construct various network functions. Compared with CPU-based software network functions, ClickNP's throughput is increased by 10 times, and the latency is reduced to 1/10; and it has negligible CPU overhead, which can save 20% of CPU cores for each computing node in cloud computing.

Secondly, this paper presents a method to accelerate access to remote data structures using programmable network cards. Key-value storage, a fundamental data structure often used, plays a crucial role in numerous distributed systems within data centers. A high-performance memory key-value storage system, KV-Direct, is developed based on the ClickNP programming framework. This system bypasses the server-side CPU and utilizes programmable network cards to directly access the host memory via PCIe. The memory operation semantics of one-sided RDMA are extended to key-value operation semantics, addressing the high communication and synchronization overhead when one-sided RDMA operates data structures. The reconfigurable feature of FPGA is also utilized, allowing users to implement more complex data structures. To address the performance challenges of lower PCIe bandwidth and higher latency between the network card and the host memory, a series of performance optimizations are employed, such as hash tables, memory allocators, out-of-order execution engines, load balancing and caching, vector operations, etc. These optimizations enable 10 times the energy efficiency of the CPU and microsecond-level latency, and for the first time, single-machine performance reaches 1 billion operations per second in a general key-value storage system.

Lastly, this paper presents a method that integrates programmable network cards and user-state runtime libraries to provide system primitives for applications, thereby bypassing the operating system kernel. The socket, the most commonly used communication primitive provided by the operating system, is redesigned and implemented in a user-state socket system, SocksDirect. This system is fully compatible with existing applications, can achieve throughput and latency close to hardware limits, has scalable multi-core performance, and maintains high performance under high concurrent loads. Intra-host and inter-host communications are implemented using shared memory and RDMA respectively. To support a high number of concurrent connections, an RDMA programmable network card is implemented based on KV-Direct. By eliminating a series of overheads such as inter-thread synchronization, buffer management, large data copying, process awakening, etc., SocksDirect increases the throughput by 7 to 20 times compared to Linux, reduces the latency to 1/17 to 1/35, and reduces the HTTP latency

of the web server to 1/5.5.

1.4 Arrangement of Thesis Structure

The structure of this thesis is organized as follows: Chapter 1 serves as the introduction. Chapter 2 presents the development trends of data centers, emphasizes the needs and opportunities for software and hardware co-optimization in data centers, discusses the architecture of programmable network cards, and reviews the application of programmable network cards in data centers. Chapter 3 proposes a data center system architecture that is based on programmable network cards. Chapter 4 is the network function acceleration section, suggesting the acceleration of virtual network functions in cloud computing using programmable network cards. To simplify FPGA programming, the first modular FPGA programming framework ClickNP, which is suitable for high-speed network packet processing and based on high-level languages, is proposed. Chapter 5 is the data structure acceleration section, suggesting the acceleration of remote data structure access using programmable network cards, and designing and implementing a high-performance memory key-value storage system KV-Direct. Chapter 6 is the operating system acceleration section, suggesting a method of integrating programmable network cards and user-state runtime libraries to provide system primitives for applications, and designing and implementing a user-state socket system SocksDirect. Chapter 7 summarizes the entire text and anticipates future research directions.

Chapter 2 Introduction to Data Centers and Programmable Network Cards

This chapter introduces the background and related work of the entire text. Figure 2.1 outlines the logical structure of this chapter. Section 1 introduces the four development trends of data centers, namely resource virtualization, distributed computing, customized computing, and fine-grained computing, starting from the application requirements, hardware, and operation modes of data centers. These trends have given birth to two major infrastructures: high-performance data center networks and memory data structure storage. Section 2 analyzes the performance challenges of data centers from the four aspects of virtual networks, network functions, operating systems, and data structure processing, namely the so-called ”data center tax”. Programmable network cards are customized hardware used to reduce the ”data center tax”. Section 3 compares the four architectures of programmable network cards based on dedicated chips, network processors, general-purpose processors, and FPGAs. Section 4 surveys the deployment of programmable network cards in data centers such as Microsoft Azure and Amazon AWS.

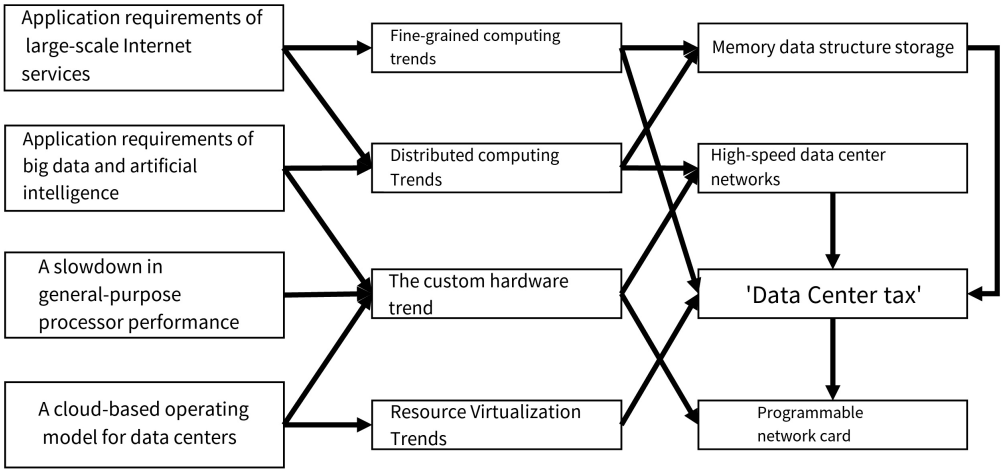


Figure 2.1 Logical structure of this chapter.

2.1 Development Trends of Data Centers

The history of data centers can be traced back to the computer rooms of early computers in the 1940s. Early computers were large precision instruments with high environmental requirements, requiring professional maintenance, and often running confidential military tasks, so they needed strictly controlled data centers for protection. With the development of computer technology, more and more users and businesses need to

use computers. Since early computers were still very expensive, a company or research institution often had only a few computers, and users connected to the computers in the data center through terminals, forming the prototype of the client-server architecture.

From the 1980s to the 1990s, the evolution of hardware, guided by Moore's Law, and advancements in operating system software, spurred the robust development of personal computers (PCs). These PCs, while more affordable, had lower computing and storage capabilities and less stability. Concurrently, mainframes and minicomputers, which were derivatives of early computers, remained the preferred choice for businesses due to their superior computing and storage capabilities, higher stability, albeit at a higher cost. These costly mainframes required dedicated data centers and professional maintenance.

As the 20th century drew to a close, the rapid expansion of the Internet, coupled with its "free" business model and the swift increase in data volume and user numbers, rendered traditional mainframes and minicomputers too expensive and inflexible to accommodate rapidly expanding businesses. Consequently, an increasing number of Internet companies began to utilize standard servers, similar in structure to PCs, to establish Internet services, thereby creating distributed systems. The proliferation of standard servers led to the continuous expansion of data centers. These servers necessitated ample room space, high-speed and stable Internet connections, stable temperature and humidity, and cheaper electricity to minimize energy costs. As a result, the construction of data centers gradually became a specialized industry, and the term "data center" became standardized.

Around 2010, as Internet companies sprouted rapidly, there was a growing demand for highly scalable computing, storage, and network resources. Cloud computing, which offers on-demand rental of computing, storage, and network resources, emerged as an increasingly popular business model. More and more businesses began migrating their traditional IT systems to cloud computing platforms to reduce operation and maintenance costs. Compared to Internet data centers, cloud data centers offer a wider range of application types and user interaction methods, higher network interconnection performance, more customized hardware, and better resource reuse. The following four sections will discuss the four development trends of cloud data centers: resource virtualization, distributed computing, customized computing, and fine-grained computing.

2.1.1 Resource Virtualization

Early computers and mainframes were costly. To allow multiple tasks to fully utilize computing and storage resources, the concept of virtualization was introduced. The time-sharing systems of the 1950s and 1960s[59-60] implemented the time-sharing reuse of hardware by multiple user tasks, evolving into the precursors of modern operating systems such as UNIX in the 1970s[61]. From the 1970s to the 1990s, Virtual Machine Monitors (VMMs, or hypervisors) further implemented the time-sharing reuse of hardware by multiple operating systems[62-63], laying the technical groundwork for the development of cloud computing.

In the first decade of the 21st century, the development of the Internet led to an increasing number of companies needing to provide 24-hour network services, and Internet Data Center (IDC) hosting services gradually emerged. However, IDC hosting requires customers to purchase server hardware in advance and requires maintenance by operations personnel, resulting in high capital expenditure (capex) and operational expenditure (opex). Many companies' network services have high seasonality (such as Amazon's Black Friday promotion), so a large amount of computing resources are idle during off-peak times. On the other hand, the rapid expansion of data and user scale puts time pressure on hardware purchasing and IDC site selection. To this end, virtual machine hosting services provide on-demand virtual machine resources, enabling the slicing of server resources by CPU cores and time-sharing reuse among different customers, and facilitating management and scheduling by internal operations personnel.

Cloud computing is an upgraded version of virtual machine hosting services, with the hallmark change being the decoupling of computing and storage. Virtual host hosting services slice the computing and storage resources on a host into multiple virtual machines. If the host's hardware or virtualization software (hypervisor) fails, the virtual machine also shuts down, and there is a risk of data loss. In cloud computing, the storage resources of virtual machines have multiple replicas in distributed storage systems, so when a computing node fails, the virtual machine can be restarted from other computing nodes, and the failure of storage nodes is generally transparent to customers. The decoupling of computing and storage not only greatly improves service availability and data security, but also facilitates the upgrading of virtualization software and hot migration of virtual machines.

In addition to sharing hardware resources with other companies, IT companies utilize cloud computing for virtualization for another purpose, specifically to repurpose

hardware infrastructure to provide different quality of service guarantees for various types of services within the company. For instance, web front-end servers responding to user requests, online transaction processing (OLTP) databases, online machine learning inference, and so forth, typically require lower latency; offline data processing (OLAP), data mining, distributed machine learning training, and so on, need to access massive data, perform a large amount of computation, and require higher throughput. Low latency and high throughput are somewhat contradictory ①, thus it is necessary to slice computing, network, storage, and other resources to provide different quality of service guarantees (Quality of Service, QoS) for applications with different needs.

As introduced in Section 1.1, the customer's virtual machines in a cloud data center are located on computing nodes, while storage services and network services run on decoupled storage and network nodes. In addition, management nodes are needed for scheduling and monitoring. As shown in Figure 2.2, a data center is usually composed of computing, network, storage, management, and other nodes, as well as the interconnection network between nodes.

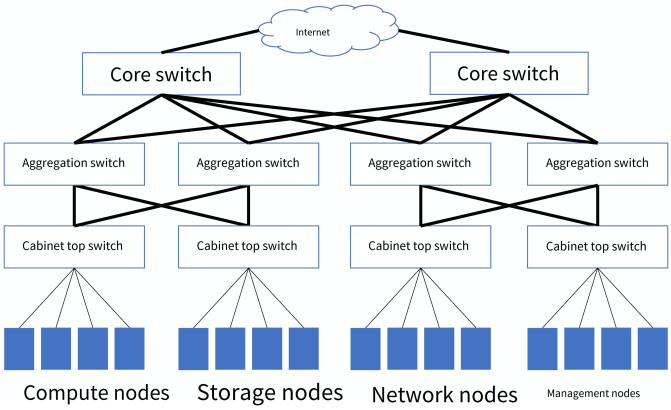


Figure 2.2 Data center architecture.

2.1.2 Distributed Computing

At the close of the 20th century, search engines bridging information islands signified the dawn of the Internet era. Search engines are required not only to gather, process, and index vast amounts of information, but also to respond to a high volume of user information retrieval requests in real time. Traditional mainframes and enterprise-level storage are not only expensive, but also incapable of meeting the scalability of massive information storage and user request processing ②. To address this, Google proposed

①Latency refers to the time difference between the end of processing and the start of processing. Throughput refers to the number of requests processed per unit of time. Latency and throughput are two important indicators of system performance. Roughly speaking, throughput equals the number of parallel processing requests divided by the average latency.

②The scalability of a distributed system refers to the increase in system throughput with the number of nodes. Ideal scalability is a linear increase in throughput with the number of nodes.

the construction of scalable data centers using standard commercial servers, employing software to execute storage partitioning and redundancy, user request distribution, and to build fault-tolerant systems on relatively unreliable hardware foundations[64-66], thus spearheading the wave of large-scale distributed computing.

Workloads such as information retrieval and Web services are relatively straightforward to parallelize. There is virtually no correlation between each user request, hence they can be dispatched to different servers for parallel processing, and increasing the number of servers can almost achieve a linear increase in system throughput. Given appropriate indexing, the amount of data required to process each user request is also minimal, and there is no high demand for communication performance.

With the evolution of search advertising, social networks, and the mobile Internet, Internet companies have amassed vast amounts of user data. In order to extract knowledge from this massive data, big data began to emerge, giving rise to big data processing frameworks represented by Hadoop[67] and Spark[68]. For instance, MapReduce[66] borrowed the concepts of mapping and reduction from functional programming languages, and proposed a programming framework for large-scale data set parallel processing on unreliable hardware clusters. Big data processing typically involves batch operations, requires access to massive data, and is not easy to parallelize and achieve linear acceleration. The fundamental reason for the difficulty in parallelization is the communication overhead between nodes. For example, in graph computing, the classic PageRank algorithm[69] consists of several stages. In each stage, each node needs to update its weight based on the weight of its adjacent nodes. In distributed computing, each node processes a part of the node set, so each stage requires a large amount of communication between nodes.

The MapReduce intermediate results are stored on disk, which results in high I/O overhead. To address this, the Spark[68] big data processing framework suggests maintaining the intermediate calculation state in memory. In-memory computing has emerged as a new paradigm for big data processing[28,70]. Once the disk bottleneck is eliminated, the latency and throughput of the data center network become the new bottleneck of the distributed system. This has led to a performance leap in data center networks from 1 Gbps to 40 to 100 Gbps over the past decade[71] (as shown in Figure 2.3), and the large-scale deployment of high-performance data center network transmission technologies represented by RDMA[41].

In recent years, in-memory computing based on RDMA has significantly improved the performance of many distributed systems, including key-value storage[70], dis-

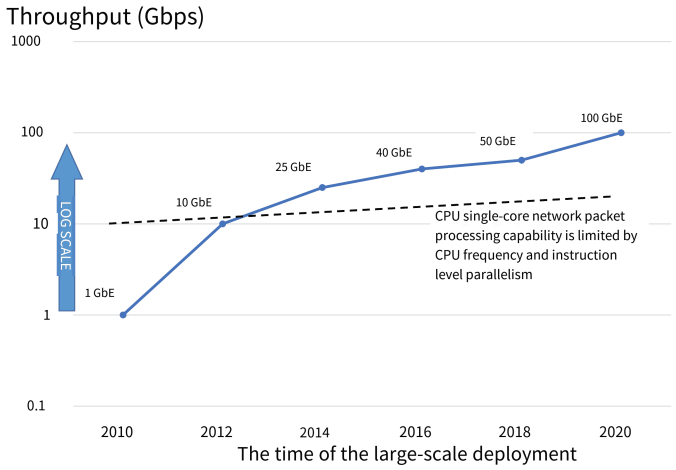


Figure 2.3 Rapid improvement of data center network performance.

tributed transactions[72-74], remote procedure calls[20], graph computing[75], and more. However, having high-performance network hardware does not necessarily lead to improved communication performance in distributed systems. Section 2.2 of this article discusses how the ”data center tax” of traditional operating systems makes it challenging for distributed applications to fully utilize the high performance of data center network hardware.

There are two paradigms for inter-process communication in distributed systems: message passing and shared memory[76]. In the message passing paradigm, the process serializes the data structure to be sent into a string and sends it to another process through a network message. Message passing in distributed systems typically adopts the remote procedure call (RPC) or message queue model, or a combination of the two. In the RPC model, the server side registers a procedure to respond to the client's RPC request. In the message queue model, the producer broadcasts or distributes messages to several consumers. To achieve decoupling of producers and consumers, buffering, and reliable delivery of messages, the message queue model often introduces a broker service, such as Kafka[77]. In terms of programming interfaces, distributed applications usually use RPC libraries and message queue middleware, which rely on the socket interface of the operating system to send and receive messages.

In the shared memory paradigm, multiple processes share memory space, and the content written by one process can be read by all processes. Remote Direct Memory Access (RDMA) technology aims to provide a shared memory abstraction for distributed systems. It has been deployed in more and more data centers in recent years and is a hot research topic in the system academic community. However, in many cases, the abstraction level of shared memory is too low. Applications need to manage memory allocation and release and object layout in memory on their own. Synchronization be-

tween processes is also required when multiple processes write at the same time. On the other hand, the abstraction level of traditional relational databases is too high, too heavyweight, and performance is limited. The abstraction level of shared data structures is between shared memory and relational databases, and it can efficiently and flexibly store structured and semi-structured data. A typical example of shared data structure storage is Redis[78], whose basic abstraction is a key-value mapping table. Users can specify a key to get or modify the corresponding value. The value can be a simple string or a more complex data structure, such as a list, set, priority queue, dictionary, etc. Redis also provides operation primitives for these data structures. Since key-value mapping is a commonly used basic data structure, many data structure storages are also called key-value storages. Shared data structure storage can reside in memory or be persisted to disk or flash.

Memory shared data structure storage is increasingly the choice of distributed applications. For example, the Spark[68] big data processing framework uses Resilient Distributed Datasets (RDD) as the basic abstraction, abstracting big data processing into a data flow graph. Each node in the graph reads data from the RDD, performs transformations, and outputs the results to the RDD. Because RDD storage can achieve high availability and scalability, big data processing programs written with RDD abstraction also have fault tolerance ① and scalability.

Message passing and shared data structure are two paradigms of inter-process communication, each with its own advantages and disadvantages in different scenarios. Both are crucial for the performance of distributed systems. These two communication paradigms can not only be theoretically converted to each other[79], but also often switch to the other paradigm in implementation. For instance, RDMA shared memory and distributed data structure storage send read and write requests and data via network messages; FaSST[20] implements high-performance RPC through RDMA shared memory read and write primitives; Redis[78] key-value storage system can function as a lightweight message queue service. Chapters 4 and 6 of this paper are dedicated to enhancing the performance of message passing. Chapter 5 of this paper is dedicated to enhancing the performance of memory shared data structure storage.

Since 2012, neural networks have been revived, deep learning has emerged as a new paradigm in machine learning, and artificial intelligence has entered another peak that continues to this day. It is widely recognized that data and computing power are

① In distributed systems, resilience refers to the ability of the system to automatically recover and continue running using the remaining nodes when a node experiences unpredictable hardware failures or operating system crashes.

the two main drivers of the revival of deep learning, with the computing power primarily supplied by customized hardware GPUs different from CPUs. The advancement of deep learning demands increasingly high computing power, which in turn stimulates the development of GPUs and various deep learning processors. Because the computing power of a single machine is insufficient to train large models on big data, distributed machine learning training has become mainstream. Distributed machine learning training typically employs the Stochastic Gradient Descent (SGD) method. SGD consists of several stages (epochs). At the start of each stage, all computing nodes share a model, use different training data to calculate the gradient of the model, then aggregate the model gradients of all computing nodes, modify the model, and use it as the shared model of all computing nodes in the next stage. Distributed machine learning training primarily has three architectures: Iterative MapReduce (IMR), parameter server, and data flow[80]. Iterative MapReduce utilizes the infrastructure of big data processing platforms and is suitable for data parallelism ① and synchronous communication.

The issue with synchronous communication is that the performance of the entire system is hindered by the slowest node, and if one node fails, the entire system cannot continue to operate. As a result, asynchronous Stochastic Gradient Descent and semi-synchronous Stochastic Gradient Descent training methods have gained popularity in recent years. In asynchronous communication methods, each node continues to train on local data, but does not need to wait for other nodes to finish training before distributing local model gradient updates to all other nodes. To facilitate the distribution of models and the aggregation of model gradients, distributed machine learning training systems often equip a distributed parameter server[81]. Each worker node retrieves the current parameters from the parameter server and uploads the local gradient updates to the parameter server. A distributed parameter server not only balances the load of each parameter storage node but also reduces communication overhead when worker nodes only access part of the parameters.

The parameter server architecture can support not only data parallelism but also model parallelism ②. For instance, DistBelief[82] utilizes both data parallelism and model parallelism; AlexNet[83] leverages the independence of computations between layers in convolutional neural networks and employs model parallelism. The parameter server is a typical application of key-value storage. Taking Microsoft's Multiverso

① Data parallelism in machine learning refers to different nodes using different data for training. Note the difference from data parallelism within FPGA mentioned later.

② Model parallelism in machine learning refers to multiple nodes calculating part of the model, i.e., the gradient of a training data needs to go through multiple nodes to be calculated.

parameter server[84] as an example, parameters can be vectors, matrices, tensors, hash tables, or sparse matrices. At present, most distributed machine learning systems of large internet companies use the parameter server architecture.

Owing to the growing popularity of data and model hybrid parallel distributed machine learning training methods, distributed deep learning systems based on data flow, such as Tensorflow[85], have emerged in recent years. In data flow systems, each node in the data flow graph represents a data processing operator, which is a finite state automaton. Nodes are connected by control message flows and data flows, adopting the distributed system communication paradigm of message passing. In fact, nodes in the data flow graph can also be parameter servers, which transforms into the distributed communication paradigm of shared data structures.

2.1.3 Customized Hardware

In the era of mainframes, mainframes and even supercomputers that needed to process a large amount of information generally adopted integrated hardware and software systems, that is, the hardware and software were developed by the same company team. Due to the use of high-speed hardware interconnection and hardware redundancy, these systems often have both high performance and high reliability, but the cost increases sharply with the expansion of the system scale.

As mentioned in Section 2.1.2, since the end of the 20th century, Internet data centers are usually composed of standard commercial servers. The reason why these ordinary commercial servers are low in cost is because their architecture is similar to a large number of commercial personal computers (PCs), and the components such as CPU, memory, motherboard, hard disk, and network card are all standard components, independently designed and implemented by various professional companies. The operating system, database, web server and other software are also standardized, either developed by professional companies or open source software. Standard components with large production volumes can better amortize one-time engineering costs such as research and development and wafer production, thereby reducing the price of standard components. Although the system composed of standard components reduces the hardware and software costs of the data center, it also imposes restrictions on the developers of standard components: everyone needs to comply with the interfaces and protocols between standard components and can only innovate within their own boundaries. The builders of the data center system can only combine standard components like building blocks, and it is difficult to consider and optimize globally.

Since 2010, the trend of cloud computing scale, the demand of data center applications, and the performance limitations of general-purpose processors have made customized computing a trend in data centers, and engineers have regained the opportunity for hardware and software co-design. First, in the cloud computing platform, the software and hardware environment are controlled by the service provider. After reaching a certain scale, all forms of customization become possible. As long as it can improve performance, reduce prices, and enhance competitiveness, cloud service providers have enough motivation to customize chips, change network protocols, change server architectures, modify operating systems, and even rewrite applications. Secondly, as mentioned in Section 2.1.2, applications such as big data and machine learning have a high demand for computing power. Finally, the performance limitation of general-purpose processors is the main driving force for customized hardware, which will be discussed in detail below.

Moore's Law predicts that the performance of unit area integrated circuits can be improved by making the storage and processing units smaller and smaller, thereby increasing the number of storage and processing units of unit area integrated circuits. More profound is Dennard's scaling law[86], that is, the performance of integrated circuits can double every two years without consuming more energy and area. Its theoretical basis is that a new generation of semiconductor technology is adopted every two years, the transistor size is reduced by 30%, and the chip area is reduced by 50%. In order to maintain a constant electric field, the voltage is reduced by 30% in proportion to the transistor size. At the same time, because the chip size is reduced, the delay is reduced by 30%, and the clock frequency can be increased by 40%[2,87]. In that era, the dynamic power consumption of integrated circuits accounted for the main part of power consumption, which was directly proportional to capacitance, the square of voltage, and frequency, so it can be calculated that power consumption is reduced by 50%. According to this ideal model, the area and power consumption of integrated circuits are halved every two years, so twice the number of transistors can be stuffed under the original area and power consumption, and the clock frequency is also increased to 1.4 times. For the single-threaded microprocessor of the von Neumann architecture, these additional transistors are mainly used for larger caches, more complex pipelines, superscalar, out-of-order execution, register renaming, branch prediction, etc., to increase the number of instructions that can be executed per clock cycle. According to Pollard's empirical law[88], the computing power per clock cycle is roughly proportional to the square root of the number of transistors. The computing power per unit time is equal

to the clock frequency multiplied by the computing power per clock cycle, so the performance of the microprocessor is doubled every two years without consuming more energy and area.

Regrettably, since the advent of the 21st century, the dividends of Moore's Law and Dennard's scaling law have been gradually diminishing. Firstly, as the feature size of integrated circuits reduces, the voltage also decreases. However, the lower the threshold voltage that controls the transistor, the faster the leakage current of the transistor will increase, becoming a significant part of the power consumption of integrated circuits. To control the leakage current, the threshold voltage cannot be reduced, and may even need to be higher than the previous generation of integrated circuits[87]. Therefore, for each new generation of semiconductor technology, the power consumption of each transistor will not be halved as anticipated.

Secondly, because the area of each transistor is halved and the power consumption is not reduced proportionately, the power consumption of unit area integrated circuits will increase. The heat dissipation problem of the chip has become a major factor limiting the scale of integrated circuits[2].

Furthermore, for the same integrated circuit, within the permissible range, in order to double the performance and double the clock frequency, the voltage must be doubled accordingly to reduce the flip delay of the transistor. Thus, the power consumption is roughly proportional to the cube of the clock frequency. Due to the limitation of heat dissipation, the clock frequency of integrated circuits is also limited, and it is unrealistic to significantly improve the performance of integrated circuits by "overclocking".

Lastly, the 7 nm semiconductor process has been mass-produced, and the radius of a silicon atom is 0.1 nm. As the feature size of integrated circuits is getting closer and closer to the atomic size, quantum effects cannot be ignored, which brings significant technical challenges to lithography technology[2]. In fact, since around 2010, the shrinkage of the feature size of integrated circuits has significantly slowed down and can no longer maintain the speed of one generation every two years.

In conclusion, under the current semiconductor technology framework, the performance of unit area integrated circuits can no longer maintain the speed of doubling every two years, and the improvement of performance also means the increase of power consumption, and the "free lunch" is over.

However, from the standpoint of chip architecture, the traditional von Neumann architecture is unable to fully exploit the computational capacity of each transistor. Consequently, there remains an opportunity to "extract as much as possible from this lemon

that is Moore's Law"[89]. Theoretically, the computational power per clock cycle could be proportional to the number of transistors, but the aforementioned Pollack's empirical law[88] suggests that the computational power per clock cycle of von Neumann microprocessors is empirically proportional to the square root of the number of transistors. For instance, the world's first microprocessor, Intel 4004, manufactured in 1971, utilized a 10-micron process, contained 2300 transistors, had a clock frequency of 108 KHz, and could execute 90 K 4-bit operations per second. The Intel Xeon E5 microprocessor, based on the Broadwell architecture in 2016, utilized a 14-nanometer process (1 M times that of 4004), contained 7.2 billion transistors (3 M times that of 4004), had a base frequency of 2.2 GHz (20 K times that of 4004), and could execute approximately 300 G 64-bit operations per second ① (3 M times that of 4004)[90]. As can be observed, the number of operations that Xeon E5 can execute per clock cycle is approximately 150 times that of 4004, while the number of transistors is 3 million times. Even when considering the complexity of 64-bit computing compared to 4-bit computing, it still implies that the contribution of each transistor in 4004 to computational power is hundreds of times higher than that of Xeon E5. This is because the instruction set and microarchitecture of von Neumann microprocessors are becoming increasingly complex, firstly to enhance single-thread performance, secondly to add deeper cache levels to address the "memory wall" issue, and thirdly to support communication and synchronization between cores, operating systems, and virtualization technologies. The proportion of transistors actually used for computation is diminishing progressively.

Due to the performance bottleneck of von Neumann architecture processors, customized hardware has emerged as a trend. The basic operations of customized hardware do not need to be expressed through instructions, and the data operation process is relatively fixed, eliminating the need for overhead related to von Neumann architecture, instruction decoding execution, and pipeline control. Customized hardware can customize data paths and memory levels, circumventing the "memory wall" problem of von Neumann architecture where all memory addresses share access. Customized hardware can construct numerous processing units to parallel process the same type of data (such as matrix operations), or a deep pipeline to process deep logic level calculations (such as symmetric encryption). As depicted in Figure 2.4, the energy efficiency of Nvidia GPUs such as K80, P100, P40, V100, Intel FPGAs such as Arria 10, Stratix 10, and Google's deep learning processor TPU is significantly higher than that of general-purpose processors (note that the y-axis is a logarithmic coordinate system),

① Assuming the application uses AVX2 instructions, does not use FMA3 instructions, and does not overclock.

and essentially follows Moore's Law's prediction in terms of performance, that is, the energy efficiency of customized hardware doubles every 18 months.

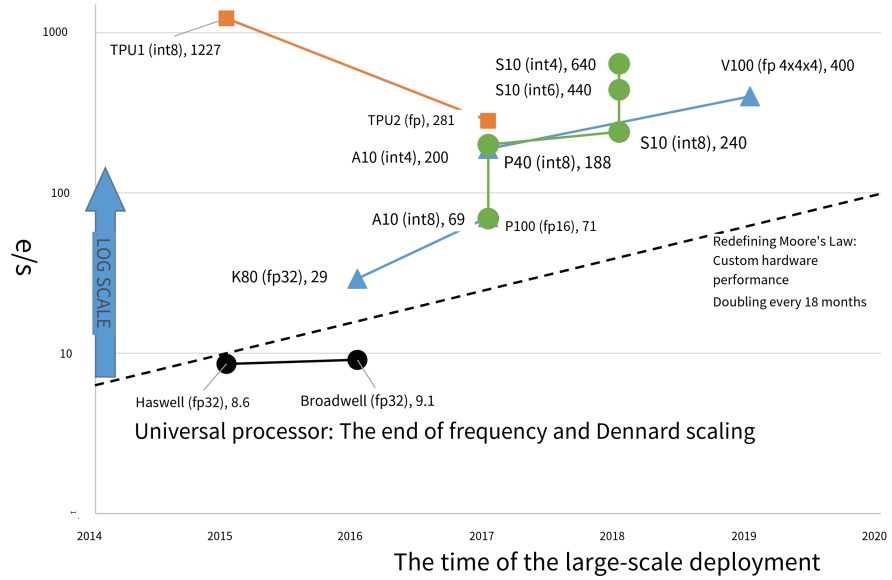


Figure 2.4 The frequency of general-purpose processors and Dennard scaling gradually end, but customized hardware redefines and continues Moore's Law.

2.1.4 Fine-grained Computing

Since the introduction of the Docker framework in 2013, numerous Internet companies have utilized containers to deploy services[91]. Containers are not merely lightweight virtual machines, but more importantly, they redefine the architecture of Internet services, decomposing a few complex macroservices into a large number of simple microservices. Each microservice is deployed in the form of a container, enabling efficient scalable data center scheduling, software dependency management, and isolation between microservices, enhancing the efficiency of development, testing, and operation. The computing granularity of the microservice architecture is finer than the traditional architecture, hence the demand for total request service capability is higher, and the demand for communication between microservices is also high. For instance, WeChat operates more than 3000 microservices on over 20,000 servers. The entrance layer microservices respond to 10 billion to 100 billion user requests per day, and each user request triggers more microservice requests within the system, so the entire WeChat backend needs to respond to hundreds of millions of microservice requests per second[92].

Containers represent a more refined computing paradigm than virtual machines. A single server host can deploy hundreds of containers, while typically only dozens of virtual machines can be deployed.

Firstly, the need to support a larger number of containers presents challenges for data center virtualization. In terms of computing virtualization, traditional virtual machines generally allocate CPU cores directly to the virtual machine. However, the number of containers usually exceeds the number of CPU cores, necessitating the operating system to schedule and allow various containers to time-share CPU cores. This increases scheduling overhead and also complicates performance isolation and quality of service assurance. In terms of network and storage virtualization, the number of containers is an order of magnitude larger than that of virtual machines, exerting pressure on lookup table capacity, queue numbers, cache capacity, and so on.

Secondly, the division of macro services into microservices has led to an increase in communication between containers. Many communications that were originally within the same virtual machine have become communications between containers, putting pressure on the container network within the server. To prevent a significant decline in the performance of microservices, it is necessary to ensure that the performance of the server container network is close to the performance of shared memory communication.

Thirdly, to enhance the fault tolerance and scalability of microservices, stateless microservice design has gradually become a trend. This means that the container itself does not store state, input data, output data, container configuration, and internal state are all stored in the data structure storage service outside the container. This necessitates the data structure storage to have low latency, high throughput, and high availability.

Container-based microservices are not the endpoint of reducing computing granularity. Traditional operating system processes still run inside the container, and these processes consume a certain amount of memory resources even without external requests. To ensure that the container can respond to user requests that may come at any time, computing resources such as CPU also need to be reserved. Containers also need to occupy a certain amount of storage space. Therefore, cloud service providers need to reserve computing, memory, and storage resources for each container, and charge for these reserved resources. The scaling, scheduling, and operation and maintenance of containers also need to be borne by the users of the containers. Although there are open-source frameworks such as Kubernetes, it also increases the burden on container users. In 2015, Amazon AWS launched a serverless computing service called Lambda, which allows users to only write event-driven business code within the programming framework of the cloud service provider, and leave tasks such as execution environment, scaling, scheduling, etc. to the cloud service provider. At present, mainstream cloud service providers such as Amazon, Microsoft, Google, Alibaba, and Tencent all

provide serverless computing services.

In 2019, the University of California, Berkeley's forecast report on serverless computing indicated that despite the numerous advantages of the serverless computing paradigm and its promotion by major cloud service providers, the performance and cost of many applications utilizing serverless computing are not ideal due to issues with cloud storage performance and the absence of temporary storage services. Indeed, serverless computing is not a novel concept. Current big data processing frameworks (such as Spark) and data lake services from cloud computing manufacturers are generally stateless for fault tolerance and scalability, and are widely adopting the serverless computing paradigm. Traditional machine learning frameworks often separate training and inference, while in reinforcement learning, the agent needs to constantly interact with the surrounding environment, and training and inference are constantly cycling, thus necessitating fine-grained computing. The distributed reinforcement learning framework Ray also employs a stateless data processing function programming model and saves the intermediate state of the data stream processing process through key-value storage.

In summary, fine-grained computing presents challenges to the performance of data center virtualization, container networks, data structure storage services, and operating system scheduling.

1. Programmable Switches

The trend towards programmable switches is driven by both demand and hardware. From the demand perspective, the automation of network operations (self-driving network) is becoming increasingly important. To achieve automated network fault detection, diagnosis, and recovery, intelligence must be incorporated into the network, rather than treating the network as a black box. For this reason, programmable packet capture and statistical functions need to be added to the switch. Moreover, by utilizing the high-speed packet processing capabilities of the switch, caching, aggregation, synchronization, transaction processing, etc., in distributed systems can be accelerated, and low-latency, lossless networks also require the support of programmable switch hardware. Many people believe that adding programmability to switches will increase chip area and power consumption, but this is not necessarily the case. Barefoot Company pointed out that about 30% of the area of the switch chip is used for serial IO communication, 50% of the area is used for memory for lookup tables and packet buffering, and only 20% of the area is used for packet processing logic. As the bandwidth of the switch continues to grow, the chip area is also increasing, and at this time, the 20% area

used for packet processing has spare capacity, which can accommodate more logic[93].

Programmable switches can be categorized into three levels based on their programmability, from low to high. The first level includes switches that adhere to the OpenFlow standard. The packet processing logic is a pipeline composed of several match-action tables, but each table has certain restrictions on matching and execution. The second level includes switches that adhere to the P4 standard. Based on the pipeline structure, the matching rules, operations to be performed, and packet header parsing rules for each table entry can be customized. The third level is network processors, which use multi-core CPUs specifically designed for network processing to process each packet. This level can achieve maximum flexibility, but the throughput of a single network connection is limited by the CPU frequency. In reality, fixed-function pipelines, general-purpose programmable pipelines, and network processors are not clearly distinguished. In data center switches, a trend towards gradual integration can be observed, such as using crossbar switches or on-chip networks to flexibly interconnect various packet processing modules and on-chip memory.

To simplify switch operation and maintenance, Microsoft initiated the SONiC white box switch open source project[94]. Firstly, a set of switch chip APIs were defined, allowing the hardware and software of the switch to evolve independently without worrying about compatibility issues. Secondly, a container-based modular switch software architecture was designed. By storing persistent states independently of the container, fine-grained fault recovery and zero service interruption time online upgrades were achieved. Finally, monitoring and diagnostic capabilities were provided to support automatic network operation and maintenance.

2. RDMA Network Cards

Traditional Ethernet network cards have relatively simple functions. The network protocol stack is implemented in the operating system kernel, and software middleware provides higher-level abstractions such as RPC, message queues, etc., to the application program. In cloud computing scenarios, virtual switch software is also needed to implement network virtualization and firewall and other network functions. Therefore, the end-to-end round-trip latency of the data center network is as high as hundreds of microseconds on average, and in extreme cases, it can even reach the millisecond level, which cannot meet the needs of low-latency distributed computing. Among them, the latency of the software accounts for the main part. The Infiniband[35] network commonly used in high-performance computing implements most of the network protocol stack in the network card, which can achieve end-to-end microsecond-level latency,

so the Remote Direct Memory Access (RDMA) technology in it has begun to become popular in data centers. To be compatible with existing data center networks, RDMA network cards in data centers usually use Ethernet and UDP/IP, and then encapsulate the RDMA reliable transmission protocol on top of it. However, the deployment of RDMA in cloud computing data centers is much more complicated than in high-performance computing: firstly, the scale of data centers is larger than that of high-performance computing clusters; virtual machines sharing the same physical host need isolation and QoS; virtual machines need hot migration and fault recovery capabilities; in addition, there are many types of applications in data centers, and communication patterns are complex, RDMA primitives have a low level of abstraction, and socket compatibility and higher-level abstractions such as RPC are needed.

3. Low-latency Network Transmission Protocol

The ideal network and interconnect should be capable of scaling to millions of devices and components within the data center, possess low latency, high bandwidth, and lossless characteristics, and be able to withstand the failure or even malicious attacks of some components. For this reason, network and interconnect designers can draw from a variety of historical designs, such as supercomputers, on-chip networks, interconnects between line cards in core routers, circuit-switched networks, time-division multiplexed networks, etc. On the other hand, they can utilize new physical layer technologies, such as optical switching chips, laser communication, and 60G wireless networks, etc.

The network latency from the sending network card to the receiving network card in the data center primarily includes the propagation delay on the optical fiber, the processing delay of the switch, and the queuing delay in the switch, with the queuing delay accounting for the majority. To reduce the queuing delay, data center switches generally use ECN, RED, and other mechanisms to feedback congestion information to the sending end, but this feedback delay is often long, and it cannot eliminate the occasional queuing caused by packets from different sending ends colliding. For this reason, a series of recent studies are rethinking the existing congestion control algorithms, and switches also need more flexible dynamic control to minimize queue length and ensure quality of service as much as possible. For example, Microsoft Azure Cloud[41], Alibaba Cloud[95], and Huawei Cloud[96] have built large-scale data center RDMA networks, constructing low-latency, high-throughput, and lossless data center networks, providing high-performance network interconnects for big data processing and large-scale machine learning.

2.1.5 In-memory Data Structure Storage

In distributed systems, in-memory data structure storage is of great importance.

Traditionally, in-memory data structure storage has primarily been utilized for caching purposes, such as frequently accessed web page content and real-time updated leaderboards. Memcached[26] and Redis[78] are widely used in-memory data structure storage software, most of which provide key-value mapping data structures and extend more complex data structures such as queues, arrays, and priority queues based on this.

As large-scale distributed computing becomes increasingly significant, in-memory data structure storage, as one of the two paradigms of inter-process communication (message passing and shared memory), has become an essential basic component of distributed computing. For instance, the big data processing framework Spark uses in-memory key-value storage as a method to store and pass intermediate results of data processing. On one hand, it is used to utilize the calculation results of the predecessor nodes in the calculation flow graph as the input of the successor nodes; on the other hand, it is used to recover from the output of the predecessor nodes when the calculation node fails, and re-execute the calculation of the successor nodes. In graph computing[97-98], distributed computing nodes share the same graph, and the points, edges, and other information in the graph can be expressed using key-value mapping.

As a final example, in distributed databases, for the concurrency control of transactions, distributed database systems often assign each transaction an increasing and non-repeating sequence number[99]. This sequence number generator is also an application scenario of key-value storage.

In the future, in-memory data structure storage will also become a basic service for enabling serverless computing. In serverless computing, functions should be stateless, and stateful applications need to store states in external storage. The calculation flow graph composed of multiple functions also needs external storage to pass intermediate results. If persistent storage is used, especially existing cloud storage, its performance and cost are unacceptable[100].

In-memory data structure storage, as a fundamental service within the data center, is becoming increasingly crucial in the context of expanding distributed computing scales and progressively reducing programming granularity.

2.2 “Data Center Tax”

The rise of distributed computing in data centers has not only led to the development of high-performance data center networks but also imposed high demands on the performance of in-memory data structure storage. The communication requirements between customized hardware have also led to the development of high-performance data center interconnects, thereby promoting the integration of networks and interconnects. However, implementing resource virtualization in high-performance data center networks incurs significant overhead.

In addition to resource virtualization, the performance of the network protocol stack in traditional operating systems is also unsatisfactory, largely wasting the low latency and high throughput of data center networks[6]. The trend towards fine-grained computing exacerbates the performance pressure on virtualization and operating systems, and also necessitates high-performance in-memory data structure storage. This article refers to the costs of the data center, excluding the execution of user applications, as the ”data center tax”[4]. This includes the operating system and virtual machine monitor on the computing node, network node, and storage node, as shown in the shaded background box in Figure 2.5. The following sections will discuss the ”data center tax” imposed by virtual networks, network functions, operating systems, and data structure processing.

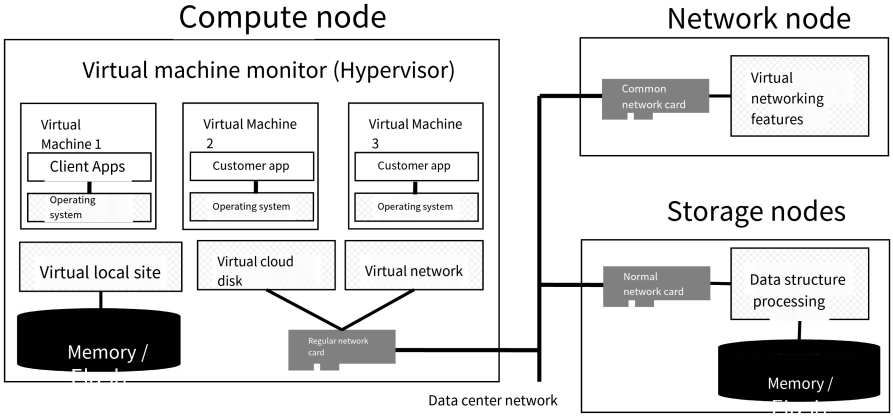


Figure 2.5 Virtualized data center architecture. The boxes with shaded backgrounds represent the costs of the data center, excluding the execution of customer applications, i.e., the ”data center tax”.

2.2.1 Virtual Networks

In public clouds, major customers require more than just virtual machines; they also need the simulation of enterprise network architectures. To support network security features such as security groups and access control lists, as well as to conceal internal network structures on the internet and reduce attack surfaces, public cloud services

provide Virtual Private Cloud (VPC) services. This involves partitioning a logically isolated section in the public cloud, offering rich network semantics, such as private virtual networks with customer-provided address spaces, security groups and access control lists (ACLs), virtual routing tables, bandwidth metering, Quality of Service (QoS), etc. Consequently, the network in the public cloud is virtualized, decoupling the virtual network from the physical network. As shown in Figure 2.6, when two customer virtual machines on two computing nodes communicate, they need to go through the encapsulation and decapsulation of the virtual network software, and may also need to go through the processing of several network functions on the network node. Network functions will be the main topic of the next section.

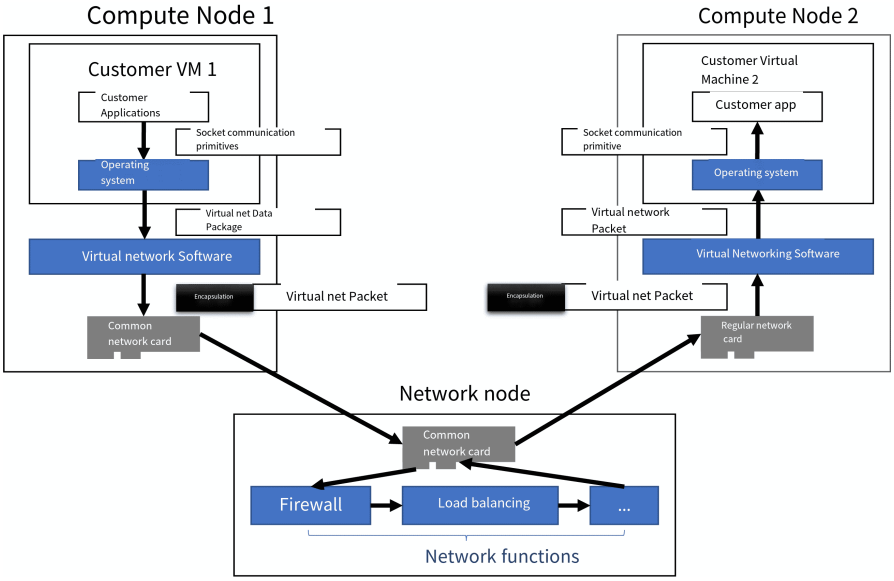


Figure 2.6 Data center virtual network architecture.

The data plane of the virtual network can be described by a Match-Action Table, which can theoretically be implemented on commercial network switches. In 2007, Stanford University proposed OpenFlow[101], which unified the control plane interfaces of different vendor switches, allowing the network to be programmed by software, i.e., Software Defined Networking (SDN). To support the control plane of SDN, Onix[102] proposed a control framework for large-scale switches, and programming languages like Frenetic[103-104] proposed using the Functional Reactive Programming (FRP) paradigm to simplify control plane event handling. Covisor[105] implemented the virtualization of the control plane. As the programmability of switches increases, it becomes possible to define the data plane forwarding behavior of switches from top to bottom using software, rather than adapting to the fixed functions of switches from bottom to top like OpenFlow.

For this purpose, in 2013, Stanford University proposed P4[106], providing programmable packet parsing, stateful match-action pipelines, and other programming ab-

stractions. The academic community has proposed various implementations of the P4 language on programmable switch chips[107], programmable network cards[108], FPGAs[109], and CPU virtual switches[110]. Industrial solutions like the Barefoot Tofino switch chip[111], Mellanox Connect-X network card[112], and the Xilinx SDNet network processor based on FPGA[113] also support the P4 language.

However, virtual networks based on network switches face two fundamental challenges in cloud data centers. First, the semantics of actual cloud data center virtual networks are very complex and change too frequently, making it difficult for the update speed of traditional switch hardware with fixed functions to match the speed of demand changes. Second, a top-of-rack switch connects dozens of rack servers, each of which can virtualize dozens of virtual machines, so the switch needs to support the data plane encapsulation and forwarding rules of up to thousands of virtual machines. The lookup table capacity of existing commercial switch chips is insufficient. For this reason, Microsoft proposed a programming abstraction similar to P4, VFP[114], to support host-based software-defined networking, implementing virtual networks in virtual switch software. Host-based virtual networks can scale well with the number of computing node servers and maintain the simplicity of the physical network.

In this network virtualization model based on virtual switches, every packet sent and received by a virtual machine is processed by the virtual switch (vSwitch) in the virtual machine monitor. Receiving packets usually involves the virtual switch software copying each packet to a buffer visible to the virtual machine, simulating a soft interrupt for the virtual machine, and then letting the network protocol stack of the virtual machine's operating system continue network processing. Sending packets is similar, but in reverse order. Compared to non-virtualized environments, this additional host processing reduces performance, requires additional changes to privilege levels, reduces throughput, increases average latency and latency variation, and increases host CPU utilization.

As shown in Figure 2.3, the improvement speed of data center network performance far exceeds that of general-purpose processors. In a 10 Gbps network, only one CPU core is needed, while in the current 40 Gbps network, about 5 CPU cores are needed, and in the future 100 Gbps network, even 12 CPU cores may be needed. This brings about the ”data center tax”.

2.2.2 Network Functions

In addition to virtual networks, data centers also require a variety of network functions. For example, large-scale Internet services have multiple front-end servers processing user requests concurrently, which requires a highly available, high-performance load balancer to accept user requests and distribute them to the front-end servers. Enterprise-grade load balancers may also support a series of advanced features, such as flexible load balancing rules based on user requests, HTTPS secure connections, log recording and statistics, detection and filtering of possible denial of service (DoS) attacks, Web application injection attacks, etc.[7].

Furthermore, many governments and enterprises already have their own IT information systems. If all are migrated to the public cloud, it does not meet some institutions' requirements for data privacy and security, and the cost and risk of one-time migration are too high. Therefore, it is necessary to connect the customer's existing IT information systems (on-premises) and virtual networks on the public cloud. In addition, in response to customer concerns about data security and privacy, many cloud vendors offer a private cloud (or dedicated cloud) mode, which deploys the software and hardware architecture of the public cloud on the customer's dedicated data center infrastructure. To support the connection between on-premises, private clouds, and public clouds, cloud vendors need to provide virtual private line services, which require a dedicated line gateway capable of implementing basic network functions such as encryption, routing, access control lists, as well as advanced network functions such as caching, TCP acceleration. In some cases, the connections between these clouds and between the cloud and the office are not through SDH or MPLS dedicated lines, but through the Internet public network. At this time, protocols such as IPSec are needed to encrypt and sign the data, using an IPSec gateway[115].

In addition, operator networks' data centers also run a large number of network functions. For instance, AT&T operates over 5,000 Central Offices in the United States, each supporting tens of thousands of users, running Broadband Network Gateways (BNG) and Evolved Packet Core (EPC) gateways in the LTE network[116]. Operators not only aim to reduce the assets and operating costs of the central office but also aspire to lease the edge computing resources of the central office to third parties[116].

Traditionally, these network functions are implemented using dedicated devices, such as F5's load balancer[117], Distributed Denial of Service (DDoS) protection, Web Application Firewall (WAF), etc.; core router or F5 BIG-IP gateway provided by Cisco

offers IPSec gateway functions; operators utilize core network equipment from companies like Huawei, Ericsson, etc. However, these dedicated hardware devices are costly and lack flexibility. Public clouds and 5G networks require flexible network functions to support secure and performance isolation between customers, to meet the Quality of Service (QoS) of different customers. Data center networks and 5G core networks need to support a variety of different service requirements on the same physical network. Such as the three typical application scenarios of 5G: extremely high bandwidth (eMBB), massive scale (MTC), extremely low and stable latency (URLLC). High bandwidth, large scale, low latency requirements are somewhat contradictory to some extent and need to be balanced according to the application's requirements. For high flexibility, data centers such as Amazon, Microsoft, Google, etc., have adopted virtualized software network functions to replace dedicated devices. The 3GPP standard also specifies that the 5G core network adopts a service-based system architecture, and these services need to be implemented using virtualized network functions[118-119].

Through the examples of load balancers, dedicated / IPsec gateways, and operator networks, it can be seen that the complexity of network functions is significantly higher than that of virtual networks. Virtual networks operate at the network layer, data link layer, generally only need to process packet header information, and do not need to maintain complex variable states; while network functions cover the application layer, transport layer, network layer, etc., need to process the payload of data packets, and need to find the network connection to which the data packet belongs based on the data packet, make processing based on the current state of the connection, and then update the state of the connection. The programming flexibility of P4[107] is not enough to implement flexible payload processing and stateful processing based on connections.

In 2000, Professor Eddie Kohler of the Massachusetts Institute of Technology proposed Click[120], a modular router programming framework. Click decomposes network processing into several basic elements (element), each element is implemented with a C++ class; the router's packet processing is a data flow graph composed of these elements, and the Click programming language allows flexible interconnection between these elements. Since the Click open-source project already has a large number of elements, many network researchers only need to interconnect these elements to assemble a complex network function; since Click uses C++ language programming, its flexibility is very high. Therefore, a series of recent research works[121-122] implement network functions based on the Click programming framework. E2[123] proposed a highly available, highly scalable network function scheduling and management framework.

Chapter 4 of this paper will propose a high-performance network function processing platform that implements the Click programming framework on FPGA.

2.2.3 Operating System

The operating system mainly includes three aspects of functionality: resource virtualization, inter-process communication, and high-level abstraction. Resource virtualization refers to the sharing of computing, network, and storage resources among multiple processes in the operating system, which requires ensuring security isolation and performance isolation between processes. Inter-process communication includes message passing, shared memory, and synchronization primitives such as locks and semaphores. High-level abstraction is the abstraction of hardware resources into a unified interface that is easy to use for applications, such as the ”everything is a file” model in Linux. The network is abstracted into sequential read-write socket connections, and storage is abstracted into file systems. This paper focuses on the network functions of the operating system, namely the sharing of network resources among multiple processes, inter-process communication based on message passing, and socket abstraction.

Distributed applications commonly use the socket primitives in the operating system for communication. As shown in Figure 2.7, for communication-intensive applications such as HTTP load balancers, DNS servers, Memcached[26], and Redis[78] key-value storage servers, the operating system occupies 50% to 90% of CPU time, most of which is used to handle socket operations.

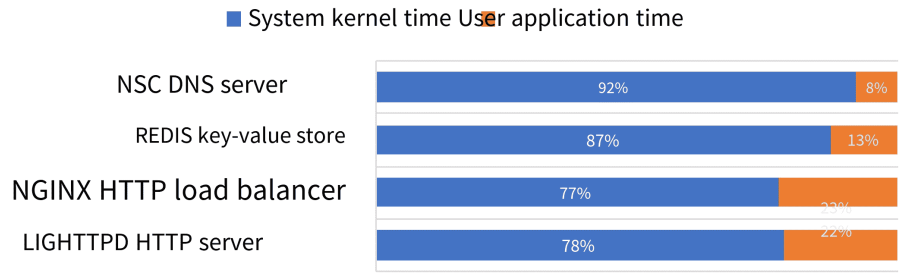


Figure 2.7 Communication-intensive applications consume a lot of CPU time in the operating system kernel.

In the Linux operating system, applications perform I/O operations through File Descriptors (FD). Conceptually, the Linux network protocol stack consists of four layers. First, the Virtual File System (VFS) layer provides a socket API based on file descriptors for applications. Second, at the transport layer, the traditional TCP transport protocol provides I/O multiplexing, congestion control, packet loss recovery, and other functions. Third, at the network and link layer, the IP protocol provides routing functions, Ethernet provides flow control and physical channel multiplexing at the data

link layer, and Linux also implements Quality of Service (QoS) and firewalls based on the netfilter framework. Fourth, at the device driver layer, the network card driver communicates with the network card hardware (or the virtual loopback interface for sockets within the host) to send and receive packets.

It is well known that the virtual file system layer contributes a large part of the overhead of network I/O operations[124-125]. In Chapter 6 of this paper, a simple experiment will verify that the latency and throughput of Linux TCP sockets between two processes in the host are only slightly worse than Linux pipes, FIFOs, and Unix domain sockets. Pipes, FIFOs, and Unix domain sockets bypass the transport layer and network card layer, but their performance is still not satisfactory.

Using the Linux network protocol stack as an example, its overhead is multifaceted. For each I/O operation, a kernel crossing is necessary, and a file descriptor lock must be obtained to safeguard multi-threaded concurrent operations. Each packet's sending and receiving involve a series of operating system overheads such as transport protocol, buffer management, I/O multiplexing, interrupt handling, and process awakening. Each byte's sending and receiving require multiple memory copies. Each establishment of a TCP network connection necessitates the allocation of kernel file descriptors and TCP control blocks, and the TCP server side also needs to schedule new connections. These overheads will be discussed in detail in Chapter 6.

Chapter 6 of this paper will propose a user-space network protocol stack, which transfers the overhead of the operating system's network protocol stack to the user-space library and programmable network card, achieving network transmission close to hardware performance.

2.2.4 Data Structure Processing

Traditional software-based memory data structure storage systems need to access the network through the operating system kernel at both the client (computing node) and server side (storage node), and also need to handle concurrent access to shared data structures through software, bringing a series of overheads.

The overhead of the operating system kernel has been discussed in the previous section. Even if these overheads are completely eliminated, the throughput bottleneck of memory data structure processing is still limited by the computation in data structure operations and the latency in random memory access. On the one hand, CPU-based key-value storage needs to spend hundreds of CPU cycles to perform key comparisons and hash slot calculations. On the other hand, the hash table of key-value storage is

several orders of magnitude larger than the CPU cache, so the waiting time for memory access is determined by the cache miss waiting time of the actual access pattern. Chapter 5 will analyze in detail that even if all CPU cores process key-value operations, their throughput is still far lower than the random access capability provided by the host DRAM memory. For this reason, Chapter 5 will propose a memory data structure storage based on programmable network cards.

2.3 Architecture of Programmable Network Cards

Programmable network cards are required to offer superior cost efficiency compared to host CPUs, while also maintaining the flexibility to adapt to changes in workloads and data center functions. For instance, they may need to extend from network virtualization to storage virtualization, or even to the network functions and data structure processing proposed in this paper. As such, programmable network cards are typically not a single chip with fixed functions, but a system on a chip (SoC). Some programmable network cards also possess control plane processing capabilities similar to those of host CPUs. Therefore, based on the architecture of the data plane, the architecture of programmable network cards can be broadly divided into four categories: dedicated chips, network processors, general-purpose processors, and reconfigurable hardware.

2.3.1 Application Specific Integrated Circuit (ASIC)

An Application Specific Integrated Circuit (ASIC) is an integrated circuit chip specifically developed for certain applications. The threshold for ASIC development is relatively high, and the development cycle is also relatively long. At the current level of technology, the one-time research and development (NRE, Non-Recurring Engineering) cost of a medium-complexity ASIC will be between several million and ten to twenty million dollars, and it requires a development cycle of one to two years. In the past, developing ASICs was often something that professional hardware chip companies could do. However, with the continuous expansion of the scale of cloud computing platforms, cloud computing system companies have also begun to try to design dedicated chips for their own clouds.

Most data center network cards designed by network card manufacturers have a certain degree of programmability. For example, the Mellanox ConnectX-5[126] hardware network card not only has the standard network card's packet sending and receiving functions, Receive Side Scaling (RSS) and Virtual Machine Queues (VMQ), TCP

checksum and Large Send Offload (LSO), RDMA function, QoS network queue and routing table offload functions, but also includes a network switch with a configurable Match-Action Table, which can implement Open vSwitch (OVS) offload function. Although the Match-Action Table can be programmed, it is not Turing complete and lacks the flexibility to parse new packet encapsulation formats, cannot parse application layer protocols, and cannot implement new encryption algorithms. In other words, the data plane programmability of dedicated chips is implemented by configuring the Match-Action Table, not by a general programming language.

Traditionally, Microsoft has partnered with dedicated network chip suppliers (such as Intel, Mellanox, Broadcom, etc.) to offload host networking in Windows. For instance, TCP checksum and Large Send Offload (LSO) in the 1990s, Receive Side Scaling (RSS) and Virtual Machine Queues (VMQ) for multi-core scalability in the 2000s, and stateless offload in 2010 for NVGRE and VxLAN encapsulation in Azure's virtual network solution. In fact, the Generic Flow Table (GFT)[114] proposed by Microsoft was initially intended to be implemented by ASIC suppliers. Microsoft widely shared early design concepts in the industry to see if suppliers could meet the requirements. However, over time, Microsoft's enthusiasm for this approach has diminished, as no design has emerged that meets all the design goals and constraints set by the cloud computing platform[10].

In recent years, the academic community has also proposed a variety of programmable network card architectures, such as FlexNIC[108,127], Emu[128], SENIC[129], sNICh[130], Uno[131], PANIC[132], etc., all based on dedicated chips, primarily aimed at optimizing the Match-Action Table of network switches, implementing richer network virtualization functions, high-performance communication between virtual machines within the host, more flexible Direct Memory Access (DMA), etc.[108,127].

One significant challenge faced by programmable network card suppliers when implementing a dedicated chip architecture is that SR-IOV is an example of all-or-nothing offload. If any required SDN function cannot be successfully handled in the programmable network card, the SDN protocol stack must revert to a software-based SDN implementation, almost losing all the performance advantages of SR-IOV offload.

Custom chip designs for SDN processing offer the highest performance potential. However, over time, they lack programmability and adaptability. In particular, the time span between the proposal of requirement specifications and the arrival of the chip is long, usually 1 to 2 years. During this range, requirements continue to change, mak-

ing the new chip already behind software requirements. The design of custom chips must continue to provide all functions for the server's 5-year lifecycle (at the scale of Microsoft Azure cloud, retrofitting most servers is not feasible). All-or-nothing offload means that the custom chip design specifications set today must meet all SDN requirements for the next 7 years.

Programmable network cards like the Mellanox ConnectX series incorporate embedded CPU cores to manage new functions, running firmware on the embedded CPU. For instance, the DCQCN congestion control protocol is implemented in firmware on the Mellanox ConnectX-3 network card and is only hardened into hardware on the Mellanox ConnectX-4 network card. However, these embedded CPU cores are not designed for high-speed packet processing and may become performance bottlenecks. Moreover, as new functions are added, these cores may increase processing load over time, exacerbating performance bottlenecks. Lastly, these embedded CPU cores typically need to be programmed through firmware updates of the network card, which often require programming by the network card manufacturer, thus slowing down the deployment of new functions.

Therefore, for cloud service providers, constructing programmable network cards based on dedicated chips is often not feasible, and sufficient programmability needs to be introduced on the data plane.

2.3.2 Network Processor (NP)

As early as the 1990s, network processors were extensively used in routers and switches to provide performance and flexibility. Modern network processors used in data centers generally consist of queues, packet processing cores, flow processing cores, acceleration dedicated circuits, and control cores. Unlike the embedded CPUs running firmware in dedicated chips, the packet processing cores and flow processing cores of network processors are much more powerful.

As shown in Figure 2.8, a typical network processor serving as a switch receives input from the network. To provide Quality of Service (QoS), the network processor supports classifying input packets according to rules, or classifying packets obtained from stateless processing by the packet processing core, determining priority and queue number, and entering the corresponding task queue. Each task queue may be associated with one or more sets of flow processing cores. The flow processing cores take tasks from the associated queue in turn, perform stateful flow processing; the processing results enter the output queue, and after another set of packet processing cores' stateless

processing, they are output to the network. In the scenario of a host network card, the network processor not only needs to process network packets, but also needs to interact with the host. This includes processing data send requests from the host and receive host data requests from the network. When the host receives these requests, they enter the task queue like the input packets and queue up for processing by the flow processing cores. Sometimes, the flow processing core's processing of a packet depends on the result of DMA from the host memory (for example, RDMA one-sided read request needs to fetch the corresponding data from the host memory, then form a packet to send a response) or the processing of the next packet of the same flow (for example, Web Application Firewall (WAF) and seven-layer load balancer that can parse HTTP protocol). For this reason, network processors generally provide a request hang-up function, that is, submit the intermediate state and dependencies of processing to the scheduler, and let the scheduler reprocess the request when the dependencies are satisfied.

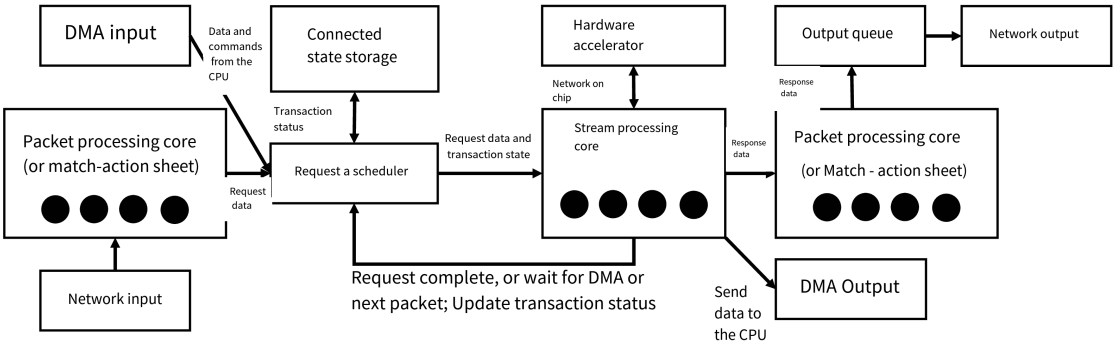


Figure 2.8 General architecture of a network processor.

Network processors can process network packets and host requests more efficiently than general-purpose processors because they have hardened many packet processing functions into hardware logic.

Firstly, in terms of Quality of Service (QoS) schedulers, hardware schedulers can implement a centralized First-Come-First-Serve (c-FCFS) scheduling model, while the Receive Side Scaling (RSS) technology of network cards, which dispatches based on connection hash, attempts to simulate a distributed First-Come-First-Serve (d-FCFS) model[133-134]. In centralized scheduling, the central scheduler maintains a scheduling queue and assigns the head task to the processor that has just completed the previous task. In distributed scheduling, the central scheduler evenly distributes requests to the queues of each processor, and each processor can only process its own queue, so there may be situations where some processor queues are not empty while other processors are idle. Queueing theory shows that c-FCFS has a more balanced load across cores

than d-FCFS, and the average delay and tail latency ① are also lower. Moreover, the RSS technology of network cards often cannot accurately simulate the d-FCFS model, because packet processing may be stateful, and in many cases, the same connection needs to be assigned to the same processor core. Only when the number of connections is large and the number of packets per connection is the same, can the d-FCFS model be simulated; but in reality, the number of packets per connection often shows a long-tail distribution, at which point the load on the processor cores is unbalanced. Theoretical analysis[135] shows that the degree of load imbalance under a long-tail distribution is directly proportional to the number of processor cores. For example, for a network processor with 64 stream processing cores, if the d-FCFS method of allocating connections based on hash is used, the load of the highest-loaded core under a long-tail distribution is 6 times the average core load.

Secondly, many commonly used modules of network processors, such as packet parsing, lookup tables, data structures, timers, DMA engines, etc., are implemented in hardware. If these lookup tables and data structures are implemented in software, they will consume a large number of instructions, affecting data plane processing performance. For example, setting and triggering a timer in software requires about 200 instructions, and parsing a TCP/IP protocol packet requires about 100 instructions[124]. From a latency perspective, in a 1 GHz network processor, if the packet transmission delay needs to be controlled within 1.5 μs, assuming the PCIe delay is 0.3 μs and the network data link layer (MAC) delay is 0.2 μs, and each packet is processed serially within the programmable network card, then the number of instructions for software to process each packet cannot exceed 1000. From a throughput perspective, if line-rate processing of 64-byte small packets under a 40 Gbps network is required, 60 M packets need to be processed per second. Assuming the network processor has 64 processing cores, each core can execute 1 G instructions per second, then the average number of instructions per packet per processing core cannot exceed 1000. Therefore, saving the number of instructions for packet processing is important for both latency and throughput. Network processors greatly alleviate the burden of packet processors and stream processors by implementing common data structures and algorithms in hardware. These hardware modules usually use on-chip network interconnection with the processing cores, allowing the cores to call these modules at any time during processing.

Thirdly, the thread scheduling and context management of the stream processing

① Tail latency refers to the highest latency in a set of latency samples. For statistical stability, latency samples are usually sorted from small to large, and then the latency at the 99%, 99.9% or other percentile is taken as the tail latency.

cores within the network processor are implemented in hardware, so fine-grained latency hiding can be achieved. For example, if a stream processing core calls a DMA operation that takes a long time, or needs to wait for a timer, the hardware will automatically save its context (including registers and the state of the stream being processed), put the thread into the not-ready queue, and switch to the next thread in the ready queue; if the ready queue is empty and the number of concurrent threads has not reached the hardware limit, it can take the next task from the task queue and create a new thread. When the DMA operation returns or the timer is triggered, the thread in the not-ready queue is switched to the ready queue. Most network processors use the non-preemptive cooperative scheduling described above. To support strict priority QoS guarantees and to fully utilize processing capabilities to handle low-priority traffic when high-priority traffic is low, some network processors have preemptive scheduling capabilities. Unlike CPUs on hosts, network processors implement the context management and scheduling functions of the operating system in hardware, greatly reducing the overhead of context switching compared to CPUs. In data center scenarios, the microsecond-level latency hiding of CPU applications has become an increasingly important issue[6], and the ”hardware operating system” design of network processors can also provide some insights for the design of host CPUs.

Fourthly, the memory hierarchy within the network processor is customized, thus the efficiency of memory access surpasses that of general-purpose processors, a feature akin to the architectural advantages of FPGAs. The ”memory wall” problem of general-purpose processors is well recognized. If all stream processing cores read and write stream states from shared memory, the overhead of cache consistency is high, which imposes a significant burden on the processor’s design. Network processors implement the binding of packet content to packet processing cores and stream state to stream processing cores in hardware, and the packet content and stream state are transported and cached through customized data paths, thereby enhancing memory bandwidth under the same chip area and process.

It is worth noting that although network processors are more energy-efficient than general-purpose processors, they are more challenging to program. Early network processors were typically programmed with microcode of a dedicated instruction set, and due to the absence of a compiler, the abstraction level of the programming language was akin to assembly. Modern network processors generally use general-purpose CPU cores (such as ARM and MIPS) as packet processors and stream processors, hence they can leverage mature compilers and development tool chains, and program in C

language. The hardened functions in programmable network cards are invoked in the form of library functions, similar to atomic operations and vector operations on Intel CPUs. Compared with general-purpose processors, the difficulty of programming modern programmable network cards primarily lies in the fact that developers need to invest time in understanding these proprietary library functions and the architecture of the network card, and cannot directly use mature code based on frameworks such as DPDK on general-purpose processors.

In terms of performance, the most significant issue with network processors is that the performance of a single core is low, leading to two implications. First, stateful streams are often mapped to a single processing core or thread to prevent state fragmentation and out-of-order processing within a single stream. Even though some network processors support dividing stateful processing into multiple stages and processing them in a multi-core pipeline, the number of pipeline stages is constrained by hardware and stream processing logic dependencies. Therefore, for stateful processing, the packet throughput of a single stream cannot exceed several times the single-core processing capacity of the network processor. The second implication of low single-core performance is that in order to support higher network bandwidth, the number of processing cores must grow linearly, which not only increases the chip area and power consumption, but also poses challenges to the design of core interconnection, memory hierarchy, and on-chip networks. At network speeds of 40 Gbps and above, the number of cores increases significantly. The on-chip network and scheduler for dispersing and collecting packets become increasingly complex and inefficient. The entire process of delivering packets to the processing core, processing packets, and then sending them to the network often requires 10 μs or more of latency. At this point, the latency is significantly higher than that of dedicated chips, and it exhibits greater variability.

Industry network processor products include Netronome NFP-32xx, Cavium OCTEON, Tilera, Mellanox NP-5, etc. Some of these network processors only have stream processing cores and no packet processing cores, but the overall architecture is similar.

2.3.3 General-Purpose Processor (SoC)

Recognizing the challenges of programming network processors, the industry has proposed a programmable network card architecture based on general-purpose processors. This architecture is akin to the ServerSwitch[136] architecture proposed by Microsoft Asia Research Institute in 2011, comprising a hardware network switch and a

general-purpose processor. For instance, the Mellanox BlueField[137] programmable network card consists of a Mellanox ConnectX-5 hardware network card and a multi-core ARM processor. The hardware network card is the dedicated chip discussed in section 2.3.1, which implements basic packet parsing, classification, queuing, and forwarding functions. As depicted in Figure 2.9, the multi-core ARM processor, Mellanox ConnectX-5 traditional network card, and the network card's on-board DRAM are interconnected through a PCIe switch. The PCIe switch further connects to the host CPU, implementing a three-way interconnection between the multi-core processor, traditional hardware network card, and host CPU. The traditional network card communicates with the multi-core ARM processor through the DRAM on the programmable network card board. The hardware network card can communicate with the host CPU and the multi-core ARM processor through the host DRAM. The typical packet reception process is: the traditional network card sends the received packets to the DRAM on the programmable network card board. The multi-core ARM processor retrieves the packets from the DRAM, processes them, and then sends them to the DRAM of the host CPU.

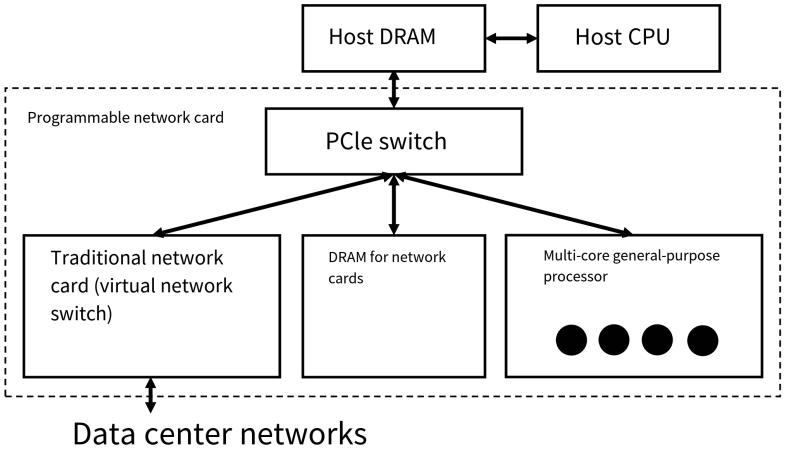


Figure 2.9 Architecture of a programmable network card based on a general-purpose processor.

Network cards based on multi-core SoCs utilize a large number of embedded CPU cores to process packets, trading off some performance to offer better programmability. The SoC architecture, based on multi-core general-purpose processors, shares many similarities with the network processor (NP) architecture, and is thus often categorized as the same type of architecture. However, they have many differences in reality. Compared to network processors, multi-core SoCs are easier to program, meaning they can adopt standard DPDK code and operate in a familiar Linux environment. Existing network function code based on the host CPU can also be easily cross-compiled to run on multi-core SoCs, while the code of network processors generally needs to be rewritten. Nevertheless, the general-purpose processor cores in multi-core SoCs communicate

with traditional network cards using off-chip DRAM, which is less efficient than the access to on-chip high-speed caches in the NP architecture. Table 2.1 compares the SoC and NP architectures of programmable network cards in data centers.

Table 2.1 Comparison of multi-core general-purpose processor and network processor architectures. The numbers come from white papers of programmable network card manufacturers. The actual application performance is affected by the complexity of the application and may not reach the theoretical performance.

Comparison item	Multi-core general-purpose processor (SoC)	Network processor (NP)
Instruction type	Standard ARM / MIPS instruction set	Extended ARM / MIPS instruction set
Operating system	General-purpose operating system (such as Linux)	No operating system or customized operating system
Operating system, paging, etc.	Supported	Generally not supported
Context switch and scheduling	Software operating system	Hardware
Locks, timers, etc.	Software	Hardware
On-board/core communication	Shared memory	Custom data path
Packet buffer	Off-chip DRAM	On-chip high-speed cache
Packet processing framework	General (such as DPDK)	Dedicated
Multi-core queuing model	d-FCFS (hardware dispatch)	c-FCFS (hardware scheduling)
Average processing latency	About 5 μs	Less than 2 μs
Single-core processing capacity	About 3 M pps	About 1 M pps
Number of processor cores	About 8	About 64
Total packet processing capacity	About 24 M pps	About 64 M pps
Power consumption	10 W to 20 W	

As discussed in the preceding section on network processors, both multicore System-on-Chip (SoCs) and network processors are constrained by single-core performance. Although the single-core performance of multicore SoCs surpasses that of network processors, the inter-core communication overhead of multicore SoCs is higher than the hardware pipeline composed of network processor processing cores. Consequently, each packet in multicore SoCs is generally processed to completion on a single processor core (run-to-completion). Therefore, the single-stream performance is typically on the same order of magnitude, around 5 Mpps (packets per second).

In terms of core number increase, since multicore SoCs mostly adopt a distributed first-come-first-serve (d-FCFS) model, if software does not utilize inter-core work stealing techniques, the imbalance of processor load will become increasingly severe with the rise in core number.

In terms of latency, due to the load imbalance among multicore SoC processors, and the non-deterministic latency brought about by the general operating system and general shared memory hierarchy used by multicore SoCs, such as operating system scheduling, interrupts, and cache misses, the latency stability of multicore SoCs is generally poorer than that of network processors, and the tail latency is generally higher than that of

network processors.

Therefore, although the multicore SoC method has a familiar programming model and good application compatibility, its evident weaknesses are single-stream performance, higher latency, and poorer scalability at higher network speeds.

From an architectural perspective, multicore SoCs are most similar to host CPUs, as they both use general-purpose processors. However, as shown in Table **??**, whether in terms of cost or performance power consumption, the general-purpose processors used in programmable network cards have clear advantages. In addition, as discussed in Chapter 1, in cloud computing data centers, host CPUs can be sold, and their potential selling price is much higher than the hardware price of a single CPU component. Therefore, using general-purpose processors embedded in programmable network cards in data centers is still advantageous compared to the traditional method of using host CPUs.

2.3.4 Field-Programmable Gate Array (FPGA)

Intuitively, an FPGA is a large collection of electronic components that can be programmatically reconfigured. These components include logic gates (such as AND, OR, NOT gates), registers, adders, static memory (SRAM), etc., and users can customize their connections to form different circuits. Figure 2.10 illustrates the basic computing unit of an FPGA, which is a logic element composed of programmable logic gates and registers.

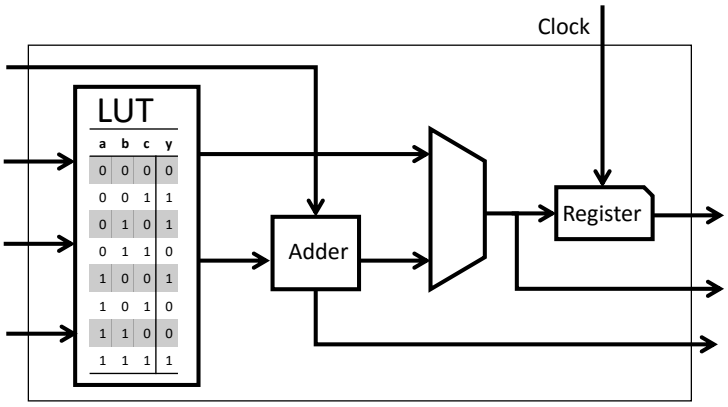


Figure 2.10 The basic computational unit of FPGA – logic element.

Modern FPGAs, in addition to basic elements, are incorporating an increasing number of DSPs and hard cores (hard IP) to enhance the performance of multiplication, floating-point operations, and access to peripheral devices. The hard cores on the

FPGA can support DDR, Ethernet, PCIe, etc., to connect to the on-board DRAM, data center network, host PCIe slot, etc. Figure 2.11 presents the logic diagram of the FPGA board used in this paper.

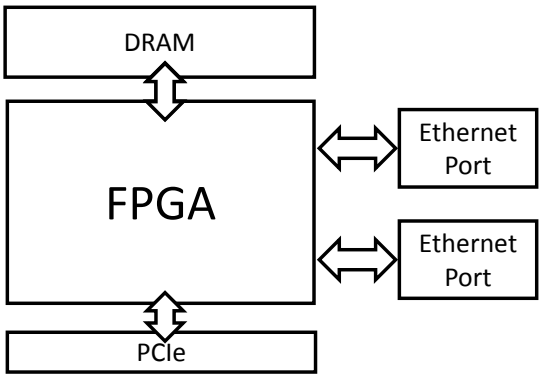


Figure 2.11 Logic diagram of the FPGA board.

General-purpose processors represented by CPUs usually adopt the von Neumann architecture and its variants. In the von Neumann architecture, since the processor (such as a CPU core) may execute any instruction, it requires an instruction memory, decoder, various instruction operators, and branch jump processing logic. Due to the complexity of the instruction flow control logic, it is impossible to have too many independent instruction flows, so both GPUs and CPUs can use SIMD (Single Instruction Multiple Data) to allow multiple processing units to process different data at the same pace. The function of each logic unit in the FPGA is determined at the time of reprogramming (burning), so no instructions are needed.

In the von Neumann architecture, memory is used for two purposes: saving state and communication between processors. In the von Neumann architecture, all processors share memory, so access arbitration is needed; to take advantage of access locality, each processor has a private cache, which requires maintaining consistency between caches of execution units. For the need to save state, there are a large number of on-chip memory (BRAM) modules in the FPGA, each of which can be connected to the logic module that needs to use the corresponding data, without unnecessary arbitration and cache. For the need for communication between logic modules, the connection between each logic module and the surrounding logic modules in the FPGA has been determined at the time of reprogramming (burning), and there is no need to communicate through shared memory.

Because the data path is customized, the FPGA can simultaneously utilize pipeline

parallelism ①, data parallelism ② and request parallelism ③ to reduce latency and increase throughput. The FPGA can organize a large number of processing units into a computation flow graph according to the dependency of data and control. Those with dependencies are pipeline parallel, and those without dependencies are data parallel. This will be discussed in detail in Chapter 4. The FPGA can also build a scheduler to achieve flexible request parallelism, maintain load balance among processing units, and hide external latency in request processing. This will be discussed in Chapter 5.

The utilization of pipeline parallelism by instruction-based GPUs and CPUs ④ is constrained. Although the instruction processing core is pipelined, its depth is limited; due to the high communication overhead between cores, the efficiency of using multiple cores to form a pipeline is often low.

The parallel computing units of the GPU and the vector instructions of the CPU can exploit data parallelism. However, the data parallelism in both the GPU and the CPU is in the SIMD (Single Instruction Multiple Data) mode, the data being processed in parallel must perform the same operation, and there can be no data dependency between parallel computations. But in many applications, the operations required for different data are different (for example, in firewalls, the packet fields and matching methods matched by different rules are different). If you want to implement data parallelism in SIMD mode, each computing unit needs to traverse all possible operations, thereby wasting some resources for unnecessary operations. In addition, many operations in packet processing have data or control dependencies, and the speedup that can be obtained by using data parallelism alone is limited. In the FPGA, as long as the data and control dependencies can be determined at compile time, the computation flow graph can be compiled into a pipeline composed of hardware logic. Finally, many SIMD instructions have restrictions on data alignment and data types, while the FPGA can implement flexible data paths and data types.

Request parallelism cannot reduce the processing latency of a single packet, but it can increase the system's throughput, thereby reducing costs. Both GPUs and CPUs can utilize request parallelism. The computing units within the same group of a GPU

① A pipeline consists of several stages, each task is processed in turn through each stage. There is generally a dependency between the processing of each stage. At any moment, each stage is processing different tasks.

② In this paper, data parallelism refers to parallel processing of unrelated data, such as vector dot product can use data parallelism, different firewall rules can also be processed in parallel. Note that this is different from the term "data parallelism" in distributed machine learning.

③ In this paper, processing a network packet or an operation offloaded by the host CPU to the accelerator card is called a work request, which is a term in RDMA. Request parallelism refers to the concurrent execution of different work requests. For ease of understanding, the paper often uses data packets instead of work requests.

④ In this section, CPU refers to general-purpose processors, including server host CPUs and processors on SoC programmable network cards.

can process different packets separately, however, these processing units in the SIMD architecture must follow a unified pace and perform the same operations. Since different packets require different processing operations, as discussed in the data parallelism above, resources need to be wasted to perform unnecessary operations. The computing units in different groups of a GPU can independently process different packets, but the programming model of the GPU usually requires these packets to be input and output together, increasing the input and output latency. Although each core of a CPU can independently receive and send packets from the network card, due to the high communication cost between the CPU and the network card, packets often need to be received and sent in batches to maintain high throughput. When packets arrive one by one instead of in batches, FPGAs can achieve lower latency compared to GPUs and CPUs. In addition, although different cores of a CPU can asynchronously process different packets, load balancing between cores is a challenge. Table 2.2 summarizes the utilization capabilities of FPGAs, GPUs, and CPUs for pipeline, data, and request parallelism.

Table 2.2 Different architectures' utilization of pipeline, data, and request parallelism.

	Pipeline Parallelism	Data Parallelism	Request Parallelism
Example	Sequentially processing MAC layer, IP layer, TCP layer, application layer of packets; calculating hash	Matching firewall rules; calculating checksum; vector calculation	Processing different packets; key-value operations
FPGA	Can utilize: Custom pipeline	Can utilize: Custom parallel processing units	Can utilize: Custom scheduler
GPU	Limited utilization: Instruction processing pipeline, pipeline composed of multiple cores	Limited utilization: SIMD vector operations	Can utilize, but with significant latency issues
CPU	Limited utilization: Instruction processing pipeline, pipeline composed of multiple cores	Limited utilization: SIMD vector instructions	Can utilize, but with latency and load balancing issues

Therefore, compared to instruction-based processors such as GPUs and CPUs, FPGAs have a latency advantage. For network packet processing, the processing latency of FPGAs can reach microseconds or even nanoseconds. If using a GPU, to fully utilize the computing power of the GPU, the batch size cannot be too small, and the latency will be at the millisecond level, which is more than 1000 times that of the FPGA.① For the host CPU, even using a high-performance packet processing framework like DPDK, the latency is 4 to 5 microseconds, which is an order of magnitude higher than that of the FPGA. A more serious problem is that the latency of the general-purpose CPU is not

①This is an estimate based on the existing GPU batch processing model network function processing framework. If the GPU supports a streaming processing programming model, its latency will be significantly reduced.

stable enough. For example, when the load is high, the forwarding latency may rise to tens of microseconds or even higher; the clock interrupt and task scheduling in modern operating systems also increase the uncertainty of latency. In data centers, latency, especially tail latency, is very important. The latency of FPGA processing network packets is at the nanosecond level, and even if it needs to access the host memory through PCIe, it only requires sub-microsecond PCIe latency.① In summary, for streaming computing tasks, FPGAs have inherent advantages over GPUs and CPUs in terms of latency.

In terms of delay, the network card receives data packets to the CPU, and the CPU sends them to the network card. Even using a high-performance data packet processing framework like DPDK, the delay is still 4 to 5 microseconds. A more serious problem is that the delay of the general CPU is not stable enough. For example, when the load is high, the forwarding delay may rise to tens of microseconds or even higher; the clock interrupt and task scheduling in modern operating systems also increase the uncertainty of the delay.

Although the GPU can also process data packets with high performance, the GPU does not have a network port, which means that the data packets need to be received by the network card first, and then the GPU is allowed to process them. In this way, the throughput is limited by the CPU and/or network card. Not to mention the latency of the GPU itself.

In summary, the main advantage of FPGA in the data center is its stable and extremely low latency, which is suitable for stream-based compute-intensive tasks and communication-intensive tasks.

FPGA is not a panacea, and there are several technical challenges, as shown in Table 2.3.

Firstly, in comparison to CPUs or GPUs, FPGAs typically possess lower clock frequencies and smaller memory bandwidths. For instance, the typical clock frequency of an FPGA is approximately 200MHz, which is an order of magnitude slower than a CPU (2 to 3 GHz). Similarly, the bandwidth of a single on-chip BRAM memory or external DRAM on an FPGA is typically 2 to 10 GBps, while the memory bandwidth of an Intel Xeon CPU is about 60 GBps, and a GPU can reach hundreds of GBps. However, CPUs or GPUs only have a limited number of cores, which restricts parallelism. FPGAs have inherent massive parallelism. Modern FPGAs may have millions of logic units, hundreds of K bits of registers, thousands of on-chip BRAMs (each with a capacity

① In the future, after Intel launches Xeon + FPGA connected via QPI, the latency between the CPU and FPGA can be reduced to below 100 nanoseconds, which is on the same order of magnitude as the latency of the CPU accessing the main memory.

Table 2.3 Challenges of using FPGA as a programmable network card.

Compared architecture	FPGA challenges	Solutions
CPU/GPU/NP/SoC	Low clock frequency	Utilize the massive parallelism inside FPGA
CPU/GPU	Low DRAM memory bandwidth	Customize data path, parallel use of on-chip BRAM memory, reduce DRAM usage
CPU/GPU/NP/SoC	Hardware description language programming is complex and difficult to debug	Programming framework friendly to software developers based on high-level synthesis technology
CPU/GPU/SoC	The software and hardware ecosystem is relatively closed	Open hardware platform, programming framework and IP core
CPU/GPU/NP/SoC	The chip area is limited and not suitable for scenarios with complex logic	Separate control plane and data plane; data plane based on customized instructions
CPU	High PCIe latency when accessing main memory	Design efficient data structures and use out-of-order execution to achieve latency hiding
CPU	Limited PCIe bandwidth when accessing main memory	Design efficient data structures and use on-board cache
CPU/GPU/NP/SoC	Need to rewrite for upgrades, interrupt service	FPGA operating system that supports dynamic reconfiguration and seamless service upgrades
CPU/NP/SoC	High task switching overhead	Spatial multiplexing, not time-division multiplexing
ASIC	Some compute-intensive loads are inefficient	Harden general modules into hard cores

of several MB), and thousands of digital signal processing (DSP) modules. Theoretically, each of these can work in parallel. Therefore, there may be thousands of parallel "*cores*" operating inside an FPGA chip simultaneously. Although the bandwidth of a single BRAM may be limited, if thousands of BRAMs are accessed in parallel, the total memory bandwidth can reach several TBps! Hence, to achieve high performance, programmers must fully utilize this massive parallelism.

Secondly, traditionally, FPGAs are programmed using hardware description languages (HDLs) such as Verilog and VHDL. These languages have a lower level of abstraction, are difficult to learn, and programming is complex. Although high-level hardware description languages like Chisel[53] have gained popularity in recent years, they still require programmers to have a basic knowledge of digital logic design and a hardware design mindset. Therefore, the software programmer community has been distant from FPGAs for many years[54]. To simplify FPGA programming, the industry and academia have developed numerous high-level synthesis (HLS) tools and systems, attempting to convert programs in high-level languages (mainly C) into HDL. However, they either do not fully utilize the massive parallelism in FPGAs, or have high latency, or are merely a supplement to the hardware development toolchain, and are

therefore not suitable for network function processing. A solution to this problem will be proposed in Chapter 4.

Thirdly, traditionally, the FPGA hardware developer community has been relatively closed. Firstly, the open-source ecosystem of FPGA is underdeveloped, leading FPGA developers to often need to implement generic modules (IP cores) from scratch or purchase them from third-party vendors. The cost of developing and purchasing generic modules deters many small and medium-sized enterprises and academic researchers. Secondly, the high price of purchasing FPGA boards in small quantities weakens the cost advantage of FPGA compared to other architectures. In recent years, with FPGA becoming a general-purpose accelerator in data centers, FPGA manufacturers and academia have been continuously promoting the construction of the FPGA ecosystem. For instance, the NetFPGA open network programming platform[138], Xilinx's SDx programming framework[58], and the P4 cross-platform network programming language[106]. Major cloud service providers have also launched on-demand FPGA cloud services and IP core markets, so developers no longer need to spend a high one-time cost to purchase boards and reinvent the wheel. The network element library in Chapter 4 and the key-value data structure in Chapter 5 are beneficial supplements to the FPGA ecosystem.

Fourthly, the logic scale implemented by FPGA with digital circuits is limited by the number of FPGA reconfigurable units, which in turn is limited by the chip area. Therefore, FPGA is not suitable for implementing very complex logic. For complex logic situations, two solutions are generally adopted together. Firstly, distinguish between the control plane and the data plane, implement the data plane in FPGA, and implement the control plane on a general-purpose processor. For example, Microsoft's virtual network accelerator[10] sends new connections to the control plane on the host CPU, where software determines the processing rules and offloads the rules to the FPGA data plane, so that subsequent packets of this connection can be processed by FPGA. The separation of the control plane and the data plane is a design idea that runs through this article. Compared with programmable network cards using general-purpose processors, using FPGA increases the programming complexity of separating the control plane and the data plane. Secondly, extract common operations from complex logic and implement customized instructions with digital logic. Complex logic is implemented through a series of customized instructions, which are stored in memory and do not occupy reconfigurable units. Microsoft's virtual network accelerator[10] has designed customized match-operation tables. Microsoft's neural network processor[139] has customized vec-

tor processing instructions for neural network calculations. The atomic operations and vector operations in Chapter 5 are examples of customized instructions.

Fifthly, to achieve fine-grained cooperative processing between FPGA and CPU, communication via the PCIe bus is necessary. This is because the DRAM capacity on the FPGA board is generally much smaller than the host DRAM, necessitating the storage of large-scale data structures on the host DRAM and access via the PCIe bus. However, the latency of the PCIe bus is in the order of hundreds of nanoseconds, which is an order of magnitude higher than CPU access to main memory. The effective bandwidth of Gen3 x8 is about 6 GB/s, which is an order of magnitude lower than CPU access to main memory. Therefore, accelerated applications on programmable network cards need to design efficient data structures, save memory access times, use out-of-order execution technology to hide latency, and fully utilize the cache of on-chip BRAM and on-board DRAM. This is the theme of Chapter 5.

Sixthly, compared with instruction-based processors, FPGA has a higher task switching overhead. On one hand, although FPGA can achieve data plane task switching without interruption through dynamic reconfiguration, this requires fixing a part of the resources at the physical location on the FPGA chip, which restricts the global optimization of FPGA placement and routing. Additional logic is also needed to assist dynamic reconfiguration, which brings about the area overhead of FPGA. On the other hand, dynamic reconfiguration of FPGA takes tens of milliseconds, much longer than the time for CPU task switching (CPU task switching based on general-purpose operating systems is generally in the order of microseconds, CPU task switching based on dedicated operating systems is generally in the order of hundreds of nanoseconds, and task switching based on dedicated processors can be completed in tens of nanoseconds). This makes FPGA unable to implement fine-grained time-sharing multiplexing like CPUs. Therefore, the current multi-user multiplexing of FPGA is mainly spatial rather than temporal, similar to allocating different CPU cores to different virtual machines. In addition, during FPGA dynamic reconfiguration, user logic cannot work, so the CPU needs to replace FPGA for processing during this period, or the FPGA service needs to be suspended, which affects the quality of service. The FPGA programmable network cards discussed in this article are all for the first-party use of data center infrastructure, not for third-party use that is publicly sold, so there is almost no need for dynamic task switching. The main challenge is to use dynamic reconfiguration to achieve service upgrades and to ensure service quality during the upgrade. The author participated in (but did not lead) a project called Feniks[140] and the recent AmorphOS[141]

aimed to solve this problem.

Seventhly, some types of workloads are compute-intensive, and their efficiency when implemented in FPGA is significantly lower than that of dedicated chips. The first type is standardized operations such as encryption and decryption. For example, the RSA asymmetric encryption based on ASIC of the Intel QuickAssist acceleration card[142] is about 10 times higher in throughput than the FPGA-based implementation in Chapter 4. The second type is common data structures such as lookup tables. For example, Content-Addressable Memory (CAM) is the basis of many concurrent operation schedulers and data structures. CAM can be implemented with tri-state gates in dedicated chips, but its efficiency is lower when implemented in FPGA. The future work outlook in Section 7.2.1 will propose to learn from the architecture of network processors and harden these operations that are not efficient when implemented in FPGA.

Finally, it should be pointed out that from the perspective of business and supply chain, FPGA has certain disadvantages. The retail price of FPGA is high, and the authorization of the development tool chain is also expensive. Therefore, if the application scale is not large enough, the amortized cost of hardware and tool chain may not be advantageous compared to programmable network cards based on network processors or general-purpose processors. For companies with very large application scales, there are only two main manufacturers of FPGAs for data centers at present, and the supply chain security has a large variable. The secondary development cost of switching between the two FPGAs is high, so self-developed network processors or embedded general-purpose processor chips may be a lower and more controllable choice in the long run. FPGA, as a compromise between general-purpose processors and dedicated chips, is suitable for scenarios with medium application scale and high uncertainty of application scenarios.

2.4 Application of Programmable Network Cards in Data Centers

Cloud service providers such as Microsoft Azure, Amazon AWS, Alibaba Cloud, Tencent Cloud, and Huawei Cloud have successively publicized their data center acceleration practices based on programmable network cards.

2.4.1 Microsoft Azure Cloud

At present, Microsoft's primary uses for deploying customized hardware in data centers include computation and infrastructure. In terms of computational acceleration, first, since 2013, it has been used for document selection and sorting algorithms for

Bing search[48]. Since 2016, hardware microservices have been used to consolidate unused resources on multiple FPGAs onto fewer FPGAs, improving FPGA utilization. Second, it is used for compression and encryption algorithms[143], initially only for Bing search, and later extended to Office 365, Cosmos / Azure data lake, Onedrive, cloud storage and other services. Third, since 2015, it has been used for machine learning inference[139,144-146], supporting not only deep learning models but also traditional machine learning models. Fourth, since 2016, it has launched FPGA-supported virtual machine instances, renting FPGA computing power to third-party customers.

The primary uses for infrastructure include networking and persistent storage. In terms of networking, since 2015, it has been used for network virtualization acceleration[10]. For persistent storage, FPGA is used to accelerate Azure cloud storage[147]. On one hand, it is used on the storage backend nodes, using the compression and encryption algorithms of the computational acceleration part to improve throughput, adopt better (but more computationally intensive) compression algorithms to improve compression rate, and save storage space; on the other hand, it is used on the storage frontend nodes and computing nodes' storage services, accelerating the data plane of the storage protocol stack through FPGA, realizing data plane bypassing the hypervisor and being able to share storage resources according to the quality of service guarantee.

Considering the above workloads that require acceleration, Microsoft's choice of custom acceleration hardware for data centers is primarily based on three aspects: the architecture of the custom hardware, the connection range between custom hardware, and the communication method between the CPU and the custom hardware.

In terms of custom hardware architecture, FPGA is suitable for both compute-intensive and communication-intensive workloads, and it has low latency. The cost per unit of computing power is lower in large-scale deployment, but the programming complexity is high; GPU is suitable for accelerating compute-intensive workloads, is simpler to program than FPGA, but has higher latency; even in large-scale deployment, the cost per unit of computing power remains high; dedicated ASIC chips are suitable for both compute-intensive and communication-intensive workloads, have the lowest latency, the highest computing power per unit power consumption, but have poor flexibility after functionalization. For instance, the above workloads expanded from Bing search to compression encryption, network, storage, machine learning, deep learning, etc., which is a gradual development process. The design of dedicated chips is difficult to achieve in one step, and redesigning a dedicated chip requires one to two years and tens of millions of dollars in non-recurring engineering costs (NRE). Based on the above

considerations, Microsoft Azure Cloud uses FPGA as the general custom hardware in the data center to accelerate various workloads.

In terms of the communication method between the CPU and the custom hardware, although coherent memory is easy to program, it is not easy to achieve high efficiency on the architecture based on x86 CPU; the bandwidth and latency of network access are relatively limited; Direct Memory Access (DMA) as a standard communication mode on the PCIe bus, becomes the communication method chosen by Microsoft FPGA. In terms of the connection range between custom hardware, the performance of a single machine is not scalable; the bandwidth of custom interconnects within a rack can be higher, but the cost of adding custom interconnects is high; Using the existing network interconnection in the data center can achieve the maximum scalability, lower cost, and the bandwidth and latency can meet the needs of most applications. Historically, Microsoft's FPGA deployment has tried the above three connection methods between custom hardware, which can be roughly divided into three development stages[148]:

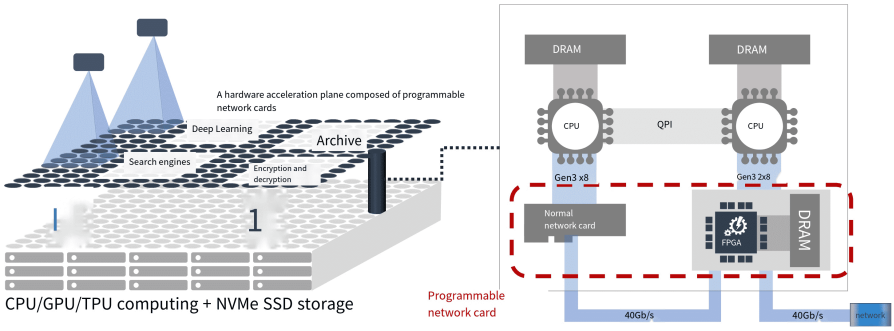


Figure 2.12 Microsoft's FPGA-based programmable network card.

This represents Microsoft's third-generation FPGA deployment architecture, and it is also the architecture currently used for large-scale deployment of "one FPGA per server". The original intention of FPGA to reuse the host network is to accelerate the network and storage, and the more far-reaching impact is to extend the network connection between FPGAs to the scale of the entire data center, making it a truly cloud-scale "supercomputer"[148]. In the second-generation architecture, the network connection between FPGAs is limited to within the same rack, and it is difficult to scale up the dedicated network interconnection between FPGAs, and the overhead of forwarding through the CPU is too high.

In the third-generation architecture, FPGAs communicate with each other through the Lightweight Transport Layer (LTL) protocol. The latency within the same rack is within 3 microseconds; 1000 FPGAs can be reached within 8 microseconds; all FPGAs in the same data center can be reached within 20 microseconds. Although the

second-generation architecture has lower latency within 8 machines, it can only access 48 FPGAs through the network. In order to support a wide range of FPGA communication, the LTL in the third-generation architecture also supports the PFC flow control protocol and the DCQCN congestion control protocol. FPGAs interconnected through high-bandwidth and low-latency networks form a data center acceleration plane between the network switching layer and traditional server software. In addition to the network and storage virtualization acceleration required by each server providing cloud services, the remaining resources on the FPGA can also be used to accelerate Bing search, deep neural networks (DNN) and other computing tasks.

At the NSDI'18 conference, Microsoft published its practice of using FPGA-based programmable network cards for network virtualization acceleration since 2015[10]. With the programmable network card, the throughput between Azure virtual machines can reach up to 31 Gbps. During the same period, the network virtualization implementation of Google Cloud Platform (GCP) based on host CPU software[149] could only reach a throughput of 16 Gbps. Amazon AWS's Nitro network virtualization acceleration based on general-purpose processors[150] can only reach a throughput of 23 Gbps. Due to the limitation of the single-core performance of the general-purpose processor, the throughput of a single TCP stream of AWS can only reach 10 Gbps, while the single-stream throughput of Microsoft Azure and Google Cloud Platform can reach the peak of the virtual machine. This is consistent with the discussion on the architecture of programmable network cards in Section 2.3.

In terms of delay, using the Linux operating system TCP/IP protocol stack, Microsoft Azure achieved an average inter-VM delay of 10 μs, while the kernel bypass DPDK[14] can achieve an average delay of 5 μs. During the same period, the average delay of the Google Cloud Platform was 20 μs, and the average delay of Amazon AWS was 28 μs. After using FPGA acceleration, the tail delay of Microsoft Azure is more obvious than the average delay. For example, at the 99.9% percentile, Azure's delay is 20 μs, the delay of the Google Cloud Platform based on the host CPU is 75 μs, and the delay of Amazon AWS based on the programmable network card general processor is 32 μs. At the 99.99% percentile, Azure's delay is 25 μs, while the Google Cloud Platform and Amazon AWS both reach or approach 100 μs. This is because the delay of hardware pipelining is more controllable than software processing, and in software processing, the latency instability is higher on the host CPU running the customer virtual machine than on the general processor of the programmable network card.

2.4.2 Amazon AWS Cloud

At the Re:Invent conference in December 2017, Amazon AWS Cloud released a computing acceleration architecture called ”Nitro”[151]. According to the information released by Amazon at the 2017 Re:Invent conference and the 2018 AWS Summit[150,152], AWS uses custom ASICs to implement various acceleration and security features. Initially, AWS weighed between FPGA and ASIC architectures and decided to adopt the ASIC solution. For this, in January 2015, Amazon acquired ASIC design company Annapurna Labs for more than $3 billion[153], which is known for designing system-on-chip (SoC) based on ARM cores.

The development of the Nitro project was phased. Similar to Microsoft Azure, the I/O bottleneck of the virtual machine was first manifested on the virtual network. As early as November 2013, AWS's C3 instances introduced a separate network card to implement enhanced networking, using SR-IOV to allow virtual machines to directly access the network card, bypassing the virtual switch software in the virtual machine monitor. This technology helped Netflix achieve a virtual machine network throughput of 2 million packets per second[154].

In January 2015, Amazon Web Services' (AWS) C4 instances started utilizing hardware-accelerated Elastic Block Storage (EBS). The data of the EBS is stored on the storage node, while the customer's virtual machine operates on the computing node, making this a form of remote storage. For the customer's virtual machine, it appears as a virtual storage device, virtualized by the storage management software in the Xen Dom0 of the virtual machine monitor. C4 instances employ high-performance network cards as opposed to traditional ones to connect to remote EBS, thereby enhancing performance.

In February 2017, AWS's I3 instances introduced NVMe local storage and a dedicated storage virtualization chip. Previously, customer virtual machines had to access local storage via the storage management software in the virtual machine monitor. This was necessary as there could be multiple virtual machines on a single physical server, and each virtual machine could only access its own portion of the storage space, necessitating isolation. For NVMe storage with high latency and throughput, the overhead of storage virtualization software is excessive. To address this, the Nitro chip introduced in the I3 instance implements storage isolation in hardware, allowing NVMe storage to be directly passed to the customer's virtual machine through SR-IOV, achieving a storage performance of 3 million I/O operations per second[155].

In November 2017, AWS's C5 instances significantly altered the virtualization architecture of the computing node. Firstly, the remote storage in the C4 instances still required software to implement virtualization. This could also be implemented in hardware like I3 local storage, but the interface of block storage is more complex than local storage, making hardware implementation more challenging. Secondly, after the network, remote and local storage have all utilized hardware virtualization, only the interrupt (APIC) function of the data plane and the management function of the control plane remain in the management software of the virtual machine monitor. The management function of the control plane is relatively complex, and it is clearly impractical to use pure digital logic. To offload all virtual networks (VPC), EBS, and virtualization control planes to the accelerator card, the Nitro ASIC adopts a system-on-chip architecture based on the ARM core, thereby maintaining the programmability and flexibility of the data plane, and also offloading the control plane to the accelerator card.

After implementing the Nitro accelerator card, AWS re-engineered a lightweight virtual machine monitor, Nitro, to replace Xen. The control plane, which was initially running on Xen Dom0, was transferred to the Nitro ASIC, allowing customer virtual machines to achieve performance close to that of bare-metal hosts. AWS later introduced bare-metal instances, where customer code runs directly on physical machines, with all storage and network resources provided by the Nitro card.

The Nitro series chips primarily consist of three chips[150-152]:

1. Cloud network (VPC) and elastic block storage (EBS) acceleration chips, which connect the data center network on one side and connect to the CPU in the form of a PCIe card on the other side;
2. Local NVMe storage virtualization chip, which acts as an intermediary between the CPU and NVMe storage devices;
3. Security chip, used to verify the version of various device firmware in the server, and to re-flash the firmware to erase traces when switching tenants on bare-metal servers.

The functions of the Nitro chip can be categorized into three aspects: cost reduction, performance improvement, and security enhancement. Let's discuss these in detail.

Saving CPU cores. Network and storage virtualization require a significant amount of CPU resources to process each network packet and storage I/O request. According to ClickNP[156], each customer virtual machine's CPU core needs to reserve an additional 0.2 CPU cores to implement virtualization. If these functions can be offloaded to dedicated hardware, the saved CPU cores can be used to install customer

virtual machines. Whether considering the price of each CPU core on a public cloud virtual machine or the hardware cost of each core of a Xeon CPU, significant cost savings can be achieved with dedicated hardware[10].

Increasing the maximum number of cores. Saving CPU cores can not only reduce costs but also increase the maximum number of cores for large virtual machine instances. Since all major public cloud manufacturers purchase CPUs from the same manufacturers such as Intel, the maximum number of CPU cores that can be purchased at the same time is relatively fixed. After virtualization is offloaded to hardware, all CPU cores are used to run customer virtual machines, so AWS's M5 instance can reach up to 96 CPU cores. If hardware offloading is not used, only 80 CPU cores will be available for customer virtual machines, thereby reducing the attractiveness to customers pursuing extreme performance.

Increase single-core frequency. Due to power consumption constraints, the number of CPU cores and the average core frequency are inversely related. Under the same generation of CPU architecture, CPUs with higher core frequencies generally have fewer cores. For virtual machine instances with equal core numbers, if traditional software virtualization is used, the physical machine will require 1.2 times the number of CPU cores, which may decrease the average core frequency. For instance, before the launch of the C5 instance, the 72-core EC2 instance had a CPU base frequency of 2.7 GHz, but the C5 instance virtual machine using the same generation Skylake architecture could achieve a CPU base frequency of 3.0 GHz.

Improve local storage performance. Firstly, on bare metal servers, local NVMe storage can achieve a throughput of up to 400 K IOPS (I/O operations per second) per disk. AWS I3 instances have 8 NVMe SSDs, reaching a throughput of 3 M IOPS. For the common storage virtualization protocol stack, each CPU core can only handle a throughput of about 100 K IOPS, which means that 30 CPU cores need to be occupied to allow the virtual machine to fully utilize the throughput of NVMe storage, which is too costly. Even if there is only one NVMe storage, load balancing among 4 CPU cores is still a problem[135]. As discussed in section 2.3, because hardware allocation and processing tasks are pipeline-style rather than simple parallelism of multiple processing units, hardware can better guarantee Quality of Service (QoS) than multi-core software.

Secondly, in terms of latency, the average latency of NVMe storage on bare metal servers is about 80 microseconds. Virtualization software not only adds an average latency of 20 microseconds, but also due to the impact of factors such as operating system scheduling, interrupts, cache misses, etc., the tail latency under high load can be

as high as 1 millisecond (1000 microseconds). Using hardware offloading can reduce the average latency by 20%, and reduce the tail latency under high load by more than 90%.

Enhancing the security of bare metal servers. Finally, customer code on bare metal servers can directly access various hardware devices within the server, and may even flash firmware of out-of-band server management (BMC) and other components[157]. If malicious code is embedded in the firmware and activated when the next tenant uses the bare metal server, the consequences are unimaginable. In fact, a large portion of customers choose bare metal servers precisely due to concerns about the isolation of virtual machines. To provide a safe and consistent hardware environment when tenants begin using bare metal servers, the Nitro security chip will overwrite the firmware. Nitro will also perform integrity checks at system startup, which is similar to UEFI trusted boot technology, but the verification scope includes not only the operating system bootloader but also hardware firmware.

2.4.3 Alibaba Cloud, Tencent Cloud, Huawei Cloud, Baidu

In 2018, domestic cloud computing service providers in China actively deployed programmable network cards in data centers. The primary purpose of Alibaba Cloud and Tencent Cloud deploying programmable network cards is to support bare-metal servers. Compared with virtual machines, bare-metal servers can eliminate the overhead brought by virtualization, achieving the highest performance under the same hardware conditions; it is convenient to deploy customers' own virtualization software (such as VMWare); it is completely the same as the deployment environment in the customer's own data center (on-premises), reducing the migration cost of customers to the cloud; it is convenient to use hardware that does not support virtualization or has a significant performance loss after virtualization, such as GPU and RDMA network cards; it does not share server hardware with other tenants, and the isolation and security are stronger, which can also meet some customers' compliance requirements.

The main technical challenge of using bare-metal servers in the public cloud is to access resources such as virtual networks (VPC) and remote storage (EBS) in the data center. A simple method is to place several virtual network and storage servers in the same rack, deploy corresponding software, and configure forwarding rules on the top-of-rack (ToR) switch, so that all network packets of the bare-metal server pass through the virtual network and storage server. This method requires additional server resources, which increases the cost. Another method is to offload the data plane of virtual networks

and storage to the top-of-rack switch. However, the programming flexibility of the top-of-rack switch is generally poor[158], which is not enough to support application layer protocols of virtual storage and security rules of virtual networks, etc.

Therefore, incorporating a programmable network card into the server has emerged as the most efficient solution to support bare-metal servers. Alibaba and Tencent have adopted the SoC solution that combines the FPGA data plane and the multi-core CPU control plane. In 2018, Alibaba Cloud launched the ”X-Dragon” bare-metal server, which uses its self-developed MOC card[159-160] to achieve network and storage virtualization similar to AWS Nitro. Tencent Cloud unveiled a programmable network card solution based on FPGA at APNet'18, primarily used for network virtualization[158]. Tencent only required 10 hardware engineers to complete the FPGA logic design in three months, manufacture the programmable network card board in four months, and deploy it within a year[158]. This is an example of the agile development that can be achieved with FPGA programming. Tencent is broadening the application scope of programmable network cards from bare-metal servers to standard virtual machines, and employs a unified programmable network card architecture for both application scenarios.

Leveraging the technological accumulation of HiSilicon Semiconductor, Huawei has launched two programmable network cards, which are also used to accelerate Huawei Cloud's virtual network. The SD100 series programmable network card[161] adopts the SoC architecture based on the multi-core ARM64 CPU, and both the data plane and the control plane operate on the ARM CPU. The IN5500 series programmable network card[162] employs a network processor (NP) to provide programmability of the data plane, which can achieve a performance of 100 Gbps. With the programmable network card, Huawei Cloud launched the C3ne network-enhanced virtual machine instance, which was the first to achieve packet forwarding at the level of tens of millions per second among domestic cloud manufacturers[163].

Although Baidu has not yet launched virtual machine instances accelerated by programmable network cards, it is a pioneer in accelerating compute-intensive applications in data centers with FPGA. As early as 2010, Baidu utilized FPGA for data compression[164]. In 2014, Baidu launched the SDA framework for using FPGA for deep learning inference[165], and subsequently used FPGA for database SQL processing[166]. In 2017, Baidu proposed the XPU, a full-stack accelerator for data centers based on FPGA, which is used for various computing acceleration scenarios[167].

Chapter 3 System Architecture

This paper proposes a high-performance data center system architecture based on programmable network cards. As shown in Figure 3.1, this paper upgrades ordinary network cards to programmable network cards, offloads the high-performance data plane required by virtualization, network and storage functions, and the operating system to the programmable network card, in order to reduce the ”data center tax” and allow the CPU to focus on the client’s applications.

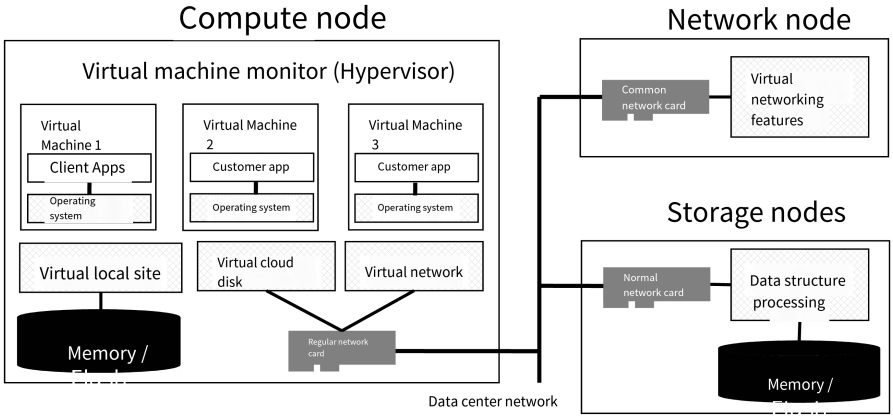


Figure 3.1 Review: Data center architecture with virtualization.

As discussed in Chapter 1, a virtualized data center can mainly be divided into computing, network, and storage nodes. In network and storage nodes, a design concept of separating the control plane from the data plane is adopted. The data plane handles relatively frequent and simple operations, while the control plane handles relatively infrequent and complex operations. The data plane is implemented in the programmable network card, and the control plane is implemented on the host CPU, achieving a data plane that does not pass through the host CPU at all. This includes the virtual network functions in Chapter 4 and the data structure processing in Chapter 5. Accelerating virtual network functions and remote data structure access are also the most important innovations of this paper.

In the computing node, that is, the server host where the client virtual machine is located, the programmable network card implements the virtualization functions of the virtual machine monitor (hypervisor) and the operating system primitives. Virtualization is divided into ”one-to-many” and ”many-to-one” aspects. ”One-to-many” means that the programmable network card virtualizes the hardware resources within the computing node into multiple logical resources, achieving multiplexing of other computing nodes and multiple local virtual machines. For example, ClickNP in Chapter 4 virtu-

alizes the hardware network card and network link into a virtual network card for each virtual machine; KV-Direct in Chapter 5 allows multiple clients to concurrently access shared key-value storage while ensuring consistency. ”Many-to-one” means that the programmable network card virtualizes physically dispersed resources within the data center into a logical resource, achieving mapping and routing from logical resources to physical resources. For example, ClickNP in Chapter 4 virtualizes network functions within the data center into logically unified network functions; the KV-Direct client in Chapter 5 virtualizes distributed key-value storage into logically unified key-value mapping; it can also achieve disaggregation of storage and memory. In order to accelerate operating system primitives and control hardware complexity, operating system primitives are divided into reliable communication protocols on the programmable network card and user-space libraries and user-space management programs running on the host CPU, such as the socket communication primitives implemented by SocksDirect in Chapter 6.

Figure 3.2 shows the overall architecture of the system based on the programmable network card. The overall design of the subsequent chapters of this paper will be briefly introduced in the order of network, storage, and high-level abstraction.

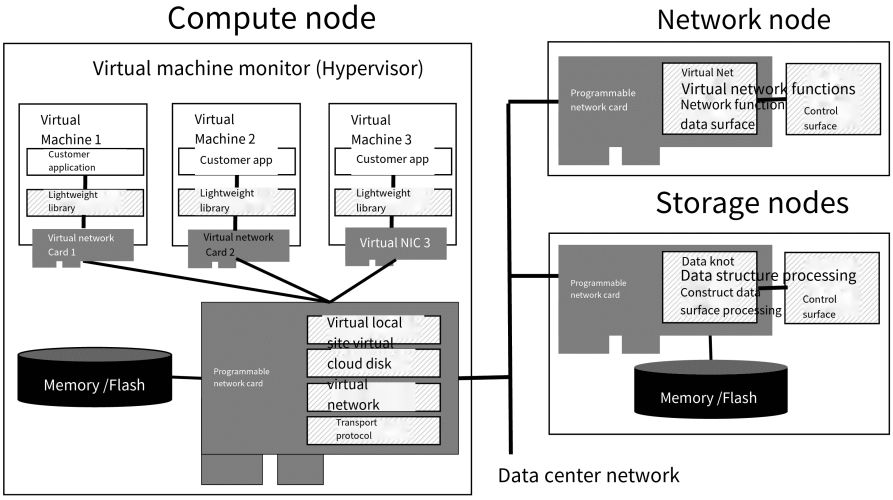


Figure 3.2 Overall architecture of the data center system based on the programmable network card.

3.1 Network Acceleration

3.1.1 Network Virtualization Acceleration

Starting from the traditional data center architecture in Chapter 1 (Figure 1.1), this paper gradually eliminates or offloads the ”data center tax” to the programmable network card. As shown in Figure 3.3, the first step is to replace the original ordinary net-

work card with a programmable network card and offload the software-implemented virtual network to the programmable network card.

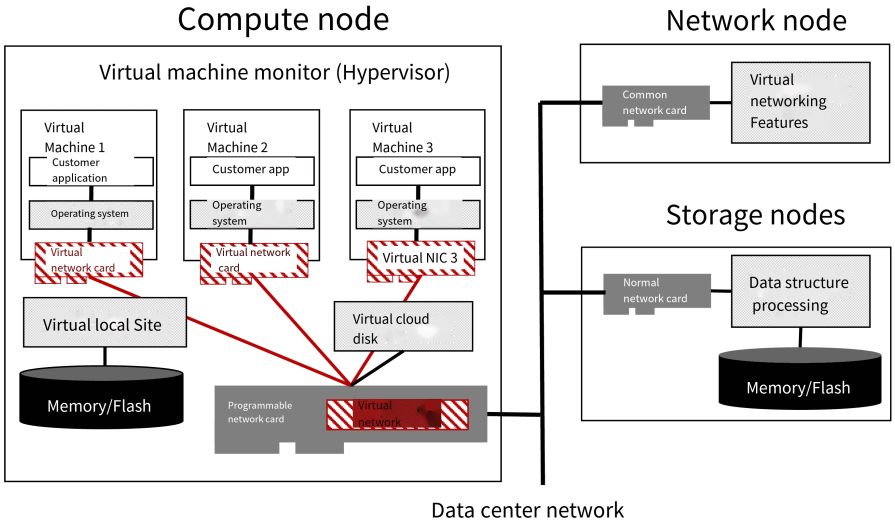


Figure 3.3 Architecture after accelerating the virtual network with a programmable network card.

In order for the operating system network protocol stack on the virtual machine to use the virtual network to send and receive data packets, the programmable network card uses SR-IOV (Single Root I/O Virtualization) technology[168] to virtualize into multiple PCIe virtual devices (VF, Virtual Function), and assigns a PCIe virtual device to each virtual machine. The original virtual network card driver in the virtual machine (such as those based on virtio technology[169]) needs to be replaced with the FPGA driver and FPGA-based virtual network card driver implemented in this paper. ClickNP in Chapter 4 virtualizes the hardware network card and network link into the virtual network of multiple tenants.

If we can bypass the network protocol stack of the virtual machine operating system and directly replace the standard library (i.e., system call interface libc) used by the application program on the virtual machine, there is no need to implement SR-IOV hardware virtualization. Chapter 6 of SocksDirect intercepts the standard library calls about network sockets of the application program and implements the container overlay network (suitable for container virtual network) in user mode. For efficient communication between the user mode runtime library and the programmable network card, the FPGA driver is installed in the virtual machine, and a part of the PCIe address space of the programmable network card is mapped to the user mode, thereby bypassing the virtual machine kernel and the Virtual Machine Monitor or Hypervisor.

3.1.2 Network Function Acceleration

As shown in Figure 3.4, the second step is to divide the software-implemented virtual network function into a data plane and a user plane on the network node, and unload the data plane into the programmable network card. It should be noted that the division of network nodes and computing nodes is logical. It is possible that the virtual network function is orchestrated to the same server host as the virtual machine, at which time the functions of the network node and the computing node are combined into one, and the connection between the virtual network and the virtual network function is simplified from the data center network to the connection between modules in the programmable network card.

After the data packet from the source computing node (or the previous network node) is received by the programmable network card of the network node, it is processed in the data plane of the network card. In most cases, the control plane on the CPU does not need to be involved, and the processed data packet can be sent back to the data center network, reaching the destination computing node (or the next network node). Chapter 4 will introduce how to use high-level language modular programming to implement network functions, and implement the collaborative processing of the FPGA data plane and the CPU control plane.

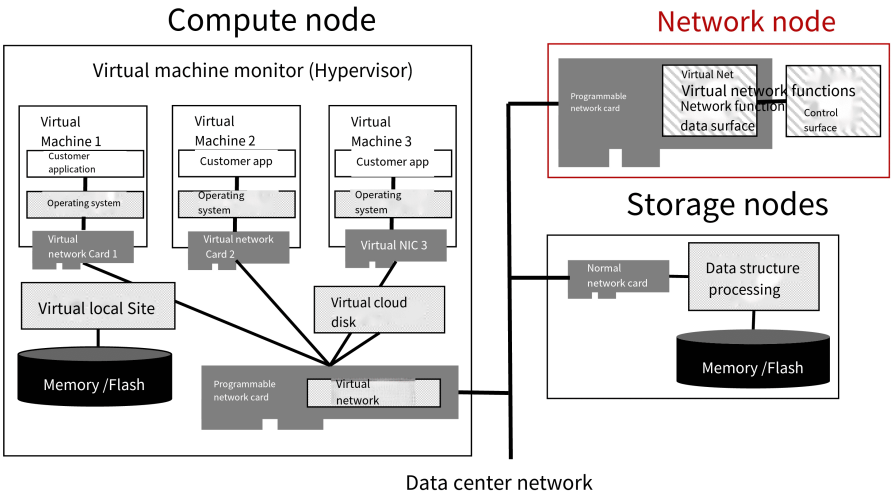


Figure 3.4 Architecture after accelerating network functions with a programmable network card.

3.2 Storage Acceleration

3.2.1 Storage Virtualization Acceleration

After network acceleration, the third and fourth steps are storage acceleration. As the third step, the storage virtualization function of the computing node is first offloaded

to the programmable network card, as shown in Figure 3.5 ①.

To support distributed storage composed of multiple storage nodes, the virtual cloud storage service needs to map logical addresses to storage node addresses. For example, in the distributed key-value storage in Chapter 4, the client needs to map the key to the storage node according to consistent hashing[170], and then route the request to that node.

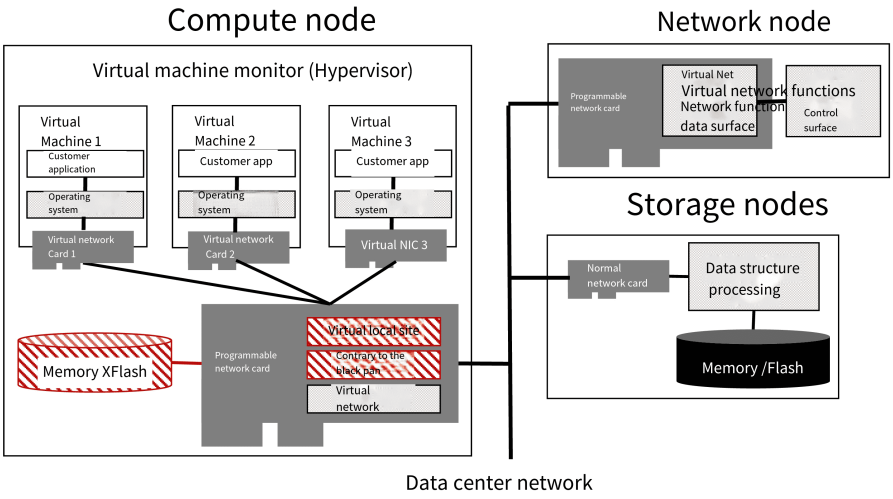


Figure 3.5 Architecture after accelerating local storage and cloud storage with a programmable network card.

3.2.2 Data Structure Processing Acceleration

The fourth step is to offload the data structure processing on the storage node to the programmable network card. Take the key-value storage detailed in Chapter 5 as an example. After the programmable network card on the storage node receives a request to query (GET) or write (PUT) a certain key from the network, it needs to query the corresponding key-value pair from the local memory or flash memory, process the request, and then send the result to the requester on the network. As shown in Figure 3.6, this process is called the data plane of data structure processing, and usually does not require the intervention of the control plane. However, since complex logic is not suitable for running on the programmable network card, the memory allocator is divided into two parts: the network card and the host CPU. The network card caches several fixed-size free memory blocks. When the free memory blocks are insufficient, the control plane on the host CPU needs to supplement the free memory blocks by splitting larger memory blocks; when there are too many free memory blocks, the host CPU needs to perform garbage collection and merge into larger memory blocks. Another challenge of directly accessing memory data structures through the network card is the lower PCIe bandwidth

①This paper does not make contributions in the area of storage virtualization, it is included in the system architecture for completeness.

and higher latency between the network card and memory. For this, Chapter 5 designed a series of optimization methods to save bandwidth and hide latency through concurrent processing. Despite the concurrent processing of requests, the design of Chapter 5 can still ensure the strong consistency of concurrent access by multiple clients, that is, the requests are logically processed in the order they are received by the network. If the storage node also runs a virtual machine as a computing node, in order to solve the consistency problem when accessing the same storage area locally and remotely, whether it is local or remote access, it is processed through the programmable network card.

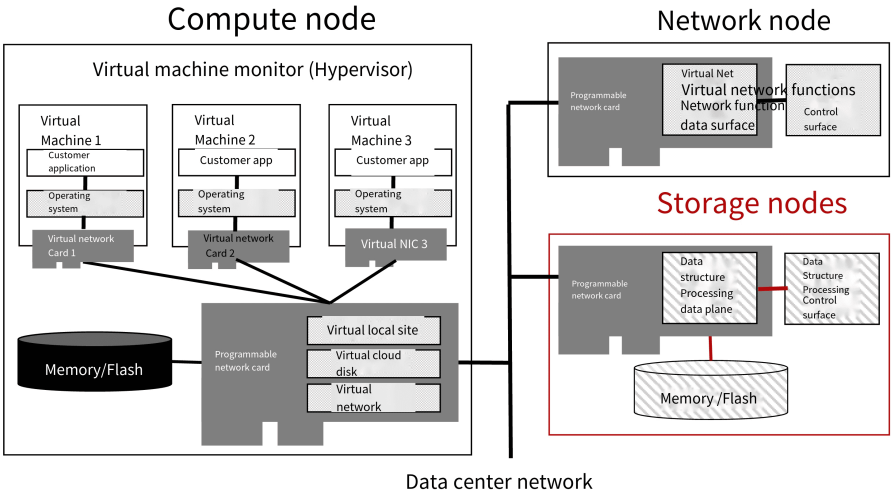


Figure 3.6 Architecture after accelerating data structure processing with a programmable network card.

3.3 Operating System Acceleration

The final step is to divide the high-performance functions in the operating system into three parts, which are processed in the programmable network card, the user-mode runtime library of the host CPU, and the user-mode daemon of the host CPU, respectively. As shown in Figure 3.7, the operating system is replaced by a user-mode runtime library in the figure, and the function of the transport protocol is added in the programmable network card. The user-mode runtime library intercepts the system calls of the application program by replacing the standard library (such as libc), so that part of the operating system functions can be implemented in user mode, and the other part of the functions can be offloaded to the programmable network card. The user-mode daemon is mainly used for control plane operations, which are not shown in Figure 3.7 for simplicity.

The operating system includes subsystems such as communication and storage. This article takes the acceleration of the communication system as an example. The

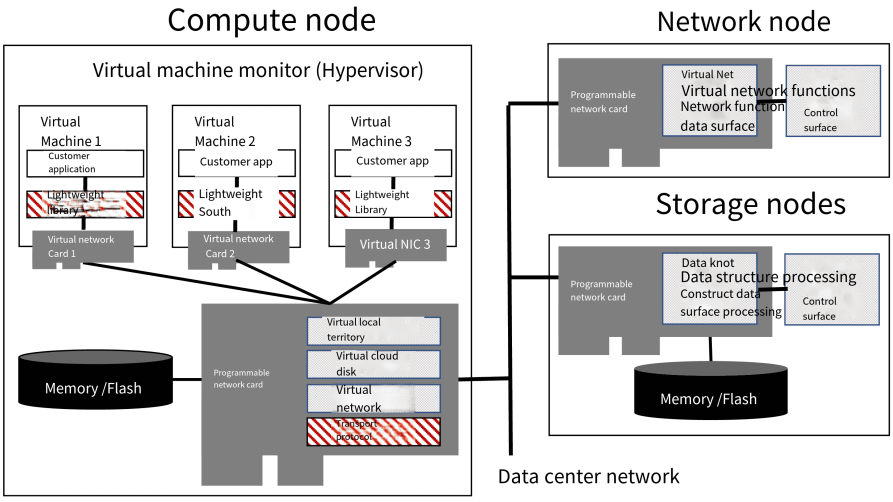


Figure 3.7 Architecture after accelerating operating system communication primitives with a programmable network card.

socket is the most commonly used communication primitive for application programs, but its performance is not satisfactory due to various overheads of the operating system. Chapter 6 designs and implements a high-performance user-mode socket system, which is fully compatible with existing applications, and can achieve low latency and high throughput close to the hardware limit for both intra-host process communication and cross-host communication. The system is composed of a reliable communication protocol on the programmable network card and a user-mode library and user-mode daemon running on the host CPU. For cross-host communication, the data plane is composed of the programmable network card and the user-mode library. The programmable network card is responsible for low-level semantics such as multiplexing and reliable transmission, and provides Remote Direct Memory Access (RDMA) primitives; the user-mode library is responsible for encapsulating RDMA primitives into the socket semantics of the Linux Virtual File System, providing high-level semantics such as lock-free message queues, buffer management, waiting events, and zero-copy memory page remapping. For intra-host communication, the data plane is composed of the CPU's hardware memory coherence protocol and the user-mode library. The user-mode library establishes shared memory queues between processes and relies on the CPU's memory coherence protocol for automatic synchronization. The functions of the user-mode library are similar to those of cross-host communication. The user-mode daemon is responsible for the control plane, that is, initialization, process creation and exit, connection establishment and closure, establishing queues with the RDMA network card, and establishing shared memory queues between processes.

In the design of Chapter 6, the client application accesses the RDMA function in

the programmable network card directly through the SocksDirect runtime library, without going through the operating system kernel and the virtual machine monitor, so the programmable network card does not need to support SR-IOV hardware virtualization.

3.4 Programmable Network Card

After introducing the data center system architecture based on the programmable network card, this section introduces the software and hardware architecture inside the programmable network card. As shown in Figure 3.8, the logic inside the programmable network card is mainly composed of the programming framework in Chapter 4, the basic service middleware in Chapter 5, and the application layer in Chapters 5 and 6. The corresponding software on the host CPU includes the FPGA communication library and driver in Chapter 4, the key-value operation library in Chapter 5, and the socket communication library compatible with the Linux operating system in Chapter 6.

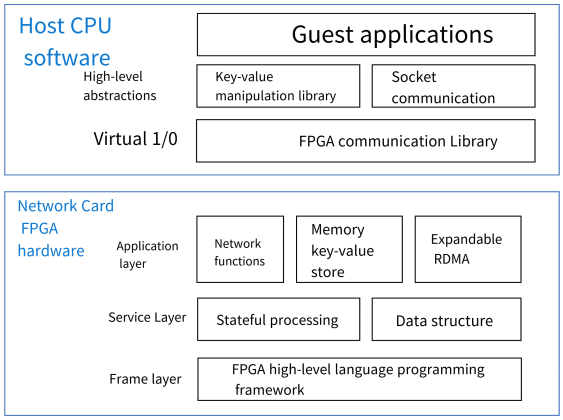


Figure 3.8 Programmable network card architecture with software-hardware co-design.

Figure 3.9 shows the hardware architecture of the Catapult programmable network card[48] used in this paper. The Catapult programmable network card consists of a Stratix V FPGA and a Mellanox ConnectX-3 commercial network card. The FPGA has two QSFP interfaces connecting to the 40 Gbps Ethernet network, one connecting to the data center switch, and the other connecting to the commercial network card inside the programmable network card. Since the FPGA used in this paper does not have a PCIe Gen3 x16 hard core, the FPGA is connected to the host through two PCIe Gen3 x8 interfaces, which share a PCIe Gen3 x16 physical slot. The commercial network card has two 40 Gbps Ethernet interfaces, one connecting to the FPGA and the other idle. The commercial network card is connected to the host through a PCIe Gen3 x16 interface.

The FPGA used in this paper has 172,600 logic elements (ALM), 2,014 on-chip

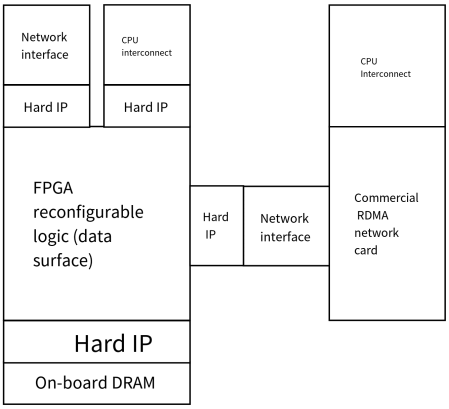


Figure 3.9 The Catapult programmable network card used in this paper.

memories (BRAM) of 20 Kb size, and 1,590 digital signal processing units (DSP) that can perform 16-bit multiplication. The FPGA board also has 4 GB of DRAM, which is connected to the FPGA via a DDR3 channel.

Chapters 4 and 5 of this paper use the reconfigurable logic of the FPGA to implement network functions and data structure processing; Chapter 6 uses the commercial RDMA network card to implement the hardware transport protocol part of socket primitives. Data packets from the data center network are received by the programmable network card from the network interface in the upper left corner of the FPGA. If it is a network function or data structure processing requirement, it is directly processed in the FPGA reconfigurable logic, and the FPGA needs to use the on-board DRAM and access the host DRAM through the CPU interconnect (such as PCIe) during the processing. If the data packet is for socket communication in Chapter 6, it will be decapsulated in the FPGA's virtual network and sent to the commercial RDMA network card through the network interface between the FPGA and the commercial RDMA network card. The commercial RDMA network card will DMA the content of the data packet to the user-space socket library on the host through the CPU interconnect (such as PCIe).

The next three chapters will discuss in detail the three main innovations of this paper, namely the acceleration of network functions based on programmable network cards (ClickNP), storage data structures (KV-Direct), and operating system communication primitives (SocksDirect).

Chapter 4 Acceleration of ClickNP Network Functions

4.1 Introduction

The focus of this chapter is network virtualization and network function acceleration, as depicted in Figure 4.1.

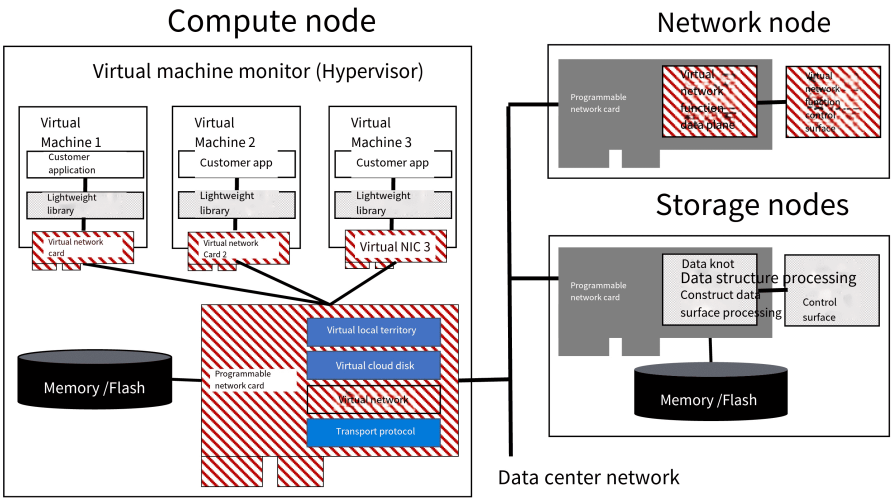


Figure 4.1 The focus of this chapter: network virtualization and network function acceleration, highlighted with a bold slanted background.

This chapter, serving as the foundation of the entire text, will introduce an FPGA high-level language programming framework and the corresponding runtime on the host CPU, and implement hardware-accelerated network functions based on this, as illustrated in Figure 4.2.

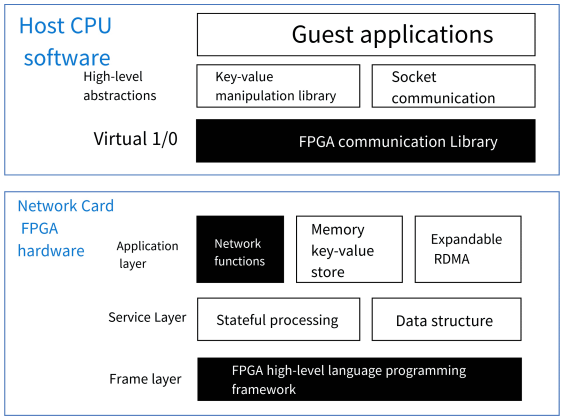


Figure 4.2 The placement of this chapter within the programmable network card software and hardware architecture.

This chapter presents ClickNP , an FPGA acceleration platform for highly flexible and high-performance network function processing on commercial servers. ClickNP addresses the programming challenges of FPGA in three steps. Firstly, it offers a mod-

ular architecture, akin to the Click model introduced in Section 2.2.2[120], where complex network functions can be composed of simple elements. ① Secondly, ClickNP elements are written in high-level C language and are cross-platform. ClickNP elements can be compiled into hardware description language and hardware modules on FPGA by utilizing commercial High-Level Synthesis (HLS) tools[49,57-58], or compiled into machine instructions on CPU using standard C++ compiler. Lastly, high-performance PCIE I/O channels provide high-throughput and low-latency communication between elements running on CPU and FPGA. PCIE I/O channels not only enable joint processing of CPU-FPGA – allowing programmers to freely divide tasks between CPU and FPGA, but also greatly assist in debugging, as programmers can easily run problematic elements on the host and use familiar software debugging tools.

ClickNP employs a series of optimization techniques to effectively harness the extensive parallelism inherent in FPGA. Initially, ClickNP arranges each element into a logic block in FPGA and links them with First-In-First-Out (FIFO) buffers. Consequently, all these element blocks can operate in full parallel. For each element, this chapter meticulously crafts processing functions to minimize dependencies between operations, enabling high-level synthesis tools to generate maximum parallel logic. Furthermore, *delayed write* and *memory scatter* techniques have been developed to address read-write dependencies and false memory dependencies, issues that existing high-level synthesis tools cannot resolve. Finally, by carefully balancing operations at different stages and aligning their processing speeds, the overall throughput of the pipeline can be maximized. Through these optimizations, ClickNP achieves a packet throughput of up to 200 million packets per second ②, and exhibits ultra-low latency (for most packet sizes, the latency is less than 2μs). Compared to the most advanced software network functions on GPU and CPU, this represents about 10 times and 2.5 times the throughput gain[36], while reducing latency by 10 times and 100 times respectively.

This chapter implements the ClickNP toolchain, which can be integrated with various commercial high-level synthesis tools[49,57], including Intel FPGA OpenCL SDK and Xilinx SDAccel. This chapter also implements approximately 200 commonly used elements, 20% of which have the same functionality as the corresponding elements in Click, and are re-implemented with reference to Click's code. This chapter will utilize these elements to construct five demonstration network functions: (1) High-speed packet sending and capturing tools, (2) Firewalls that support exact matching and wild-

①This is also the origin of the system name *Click Network Processor* (ClickNP).

②The actual throughput of ClickNP network functions may be limited by the data rate of the Ethernet port.

card matching, (3) IPSec gateways, (4) A four-layer load balancer capable of handling 32 million concurrent streams, (5) pFabric scheduler[171] performs strict priority flow scheduling, with 4 billion priorities. The evaluation results demonstrate that all these network functions can be significantly accelerated by FPGA, and can saturate the line speed of 40Gbps at any packet size, while maintaining extremely low latency and negligible CPU overhead.

In summary, the contributions of this chapter are: (1) The design and implementation of the ClickNP language and toolchain; (2) The design and implementation of high-performance packet processing modules that run efficiently on FPGAs; (3) The design and evaluation of FPGA-accelerated network functions. To the best of the author's knowledge, ClickNP is the first FPGA-accelerated packet processing platform for general network functions, completely written in a high-level language and capable of achieving 40 Gbps line speed.

4.2 Background

4.2.1 Performance Challenges of Software Virtual Networks and Network Functions

Virtual networks are typically implemented with virtual switch software, such as Open vSwitch[172]. To enhance the performance of virtual networks and meet programmability requirements, cloud service providers have redesigned software virtual switches. By leveraging high-speed network packet processing technologies such as DPDK[14], and running CPU cores in polling mode, the cost of packet processing can be significantly reduced by bypassing the OS network protocol stack. However, even for simple network packet forwarding that does not do any processing, each CPU core can only handle 10 M to 20 M packets per second[121-122], which still requires 3 to 6 CPU cores for a 40 Gbps line speed of 60 M packets per second.

Traditional network functions are implemented by dedicated devices deployed at specific locations in the data center, such as F5 load balancers[117]. These dedicated network function devices are not only expensive, but also not flexible enough to support multi-tenancy in cloud services. To support flexible network functions, cloud service providers also deploy software-implemented virtual network functions. For instance, Ananta[7] is a software load balancer deployed in Microsoft data centers, used to provide cloud-scale load balancing services. To support multiple network functions on a single server, RouteBricks[8], xOMB[173] and COMB[174] adopt the program-

ming model of the Click modular router[120], implementing each network function as a C++ class, allowing each packet to be processed by various network functions in sequence on a CPU core, the so-called "run-to-completion" model. These works show that under actual network functions, the speed of packet forwarding per server based on multi-core x86 CPUs can reach 10 Gbps, and capacity can be expanded through multi-core and building more network node clusters. To achieve isolation between network functions, NetVM[175], ClickOS[121], HyperSwitch[176], mSwitch[177] and other works put each network function in a (lightweight) virtual machine, and let the virtual switch distribute packets to these virtual machines for processing in sequence, the so-called "pipeline" model. In the pipeline model, packets are repeatedly passed between cores, which is costly. NetBricks[122] returned to the "run-to-completion" model, but implemented network functions in high-level languages, and ensured isolation between network functions at the compiler and runtime framework level. NFP[178] improves packet processing performance by using multiple parallel network function pipelines.

Although virtual switches and network functions implemented via software can support higher performance with more CPU cores and larger network node clusters, this will significantly increase asset and operating costs[7,9]. The profitability of Infrastructure as a Service (IaaS) cloud service providers is the difference between the price customers pay for virtual machines and the cost of hosting these virtual machines. Given that the asset and operating costs of each server are essentially determined at the time of deployment, the most effective way to reduce the cost of hosting virtual machines is to package more customer virtual machines on each computing node server and decrease the number of servers for network and storage nodes. Currently, the price of a physical CPU core (2 hyperthreads, i.e., 2 vCPUs) is approximately $0.1 per hour, meaning the maximum potential income is around $900 per year[10]. In data centers, servers usually serve for 3 to 5 years, so the highest price of a physical CPU core during the server's life cycle can reach $4500[10]. Even considering that some CPU cores are always unsold, and the cloud often offers discounts to large customers, compared with dedicated hardware, even dedicating a physical CPU core to virtual networking is quite costly.

To accelerate virtual networks and network functions, previous work has proposed using GPUs[36], Network Processors (NP)[37-38] and hardware network switches[9]. GPUs were initially used primarily for graphics processing, but in recent years have expanded to other applications with massive data parallelism. PacketShader[36] demonstrates that using GPUs can achieve a packet switching speed of 40Gbps. GPUs are

suitable for batch operations, but batch operations can lead to high latency. For instance, the forwarding latency reported by PacketShader[36] is about 200μs, which is two orders of magnitude higher than ClickNP . As discussed in section **??**, compared with GPUs, FPGAs can fully utilize pipeline parallelism, data parallelism and request parallelism to achieve low-latency, high-throughput packet processing. Network processors are specifically used for processing network traffic and have many hard-wired network accelerators. NP-Click[179] implemented the Click programming framework on network processors. The year after the Click modular router[120] was proposed, NP-Click[179] implemented the Click programming framework on network processors. As discussed in section 2.3.2, the main problem with network processors is that single-flow throughput is limited by single-core performance. As discussed in section 2.2.2, the main problem with hardware network switches is insufficient flexibility and lookup table capacity[9].

As discussed in Section **??**, utilizing FPGA for network function processing poses a series of challenges. This paper concentrates on harnessing massive parallelism and programming toolchains. Compared to CPUs or GPUs, FPGAs typically have lower clock frequencies and smaller memory bandwidth. For instance, the typical clock frequency of an FPGA is about 200MHz, an order of magnitude slower than a CPU (2 to 3 GHz). Similarly, the bandwidth of a single block of memory or external DRAM on an FPGA is usually 2 to 10 GBps, while the DRAM bandwidth of an Intel Xeon CPU is about 60 GBps, and the GDDR5 or HBM bandwidth of a GPU can reach hundreds of GBps. However, CPUs or GPUs only have a limited number of cores, which restricts parallelism. FPGAs have a large amount of built-in parallelism. Modern FPGAs may have millions of LEs, hundreds of K-bit registers, tens of M-bit BRAMs, and thousands of DSP modules. Theoretically, each of them can work in parallel. Therefore, thousands of parallel "*cores*" may be running simultaneously inside an FPGA chip. Although the bandwidth of a single BRAM may be limited, if thousands of BRAMs are accessed in parallel, the total memory bandwidth can reach several TBps! Therefore, to achieve high performance, programmers must fully utilize this massive parallelism.

Traditionally, FPGAs are programmed using hardware description languages such as Verilog and VHDL. These languages are too low-level, difficult to learn, and complex to program. Therefore, the large software programmer community has been far from FPGAs for many years[54]. To simplify FPGA programming, the industry and academia have developed many high-level synthesis tools and systems, aiming to convert programs in high-level languages (mainly C) into hardware description languages.

However, as the next subsection will discuss, they are not suitable for network function processing, which is the focus of this work.

4.2.2 FPGA-based Network Function Programming

The aim of this chapter is to leverage FPGAs to construct a multi-functional, high-performance network function platform. Such a platform should meet the following requirements.

Flexibility. The platform should be *fully programmed in high-level languages.* Developers should be able to program using high-level abstractions and familiar tools, just like programming on multi-core processors. This is a prerequisite for most software programmers to use FPGAs.

Modularity. The network function platform should support a *modular architecture* for packet processing. Previous experiences with virtualized network functions have shown that the correct modular architecture can capture many common functions in packet processing[120-121], making them easy to reuse in various network functions.

High performance, low latency. Network functions in data centers should process a large number of packets at line rates of 40 / 100 Gbps, with ultra-low latency. Previous work has shown[180] that even a few hundred microseconds of latency added by network functions can negatively impact the service experience.

Support for CPU/FPGA joint packet processing. FPGA is not a panacea. As discussed in Section **??**, not all tasks are suitable for FPGAs. Larger logic cannot be accommodated in FPGAs. Therefore, fine-grained processing separation between the CPU and FPGA should be supported. This requires high-performance communication between the CPU and FPGA.

FPGAs have long been used to implement network routers and switches. NetFPGA[181] proposed an open hardware platform for implementing routers on FPGAs. In recent years, FPGAs have also been used to accelerate network functions[182-183]. As early as the second year after the Click modular router[120] programming framework was proposed, Xilinx proposed Cliff[184] for implementing Click with FPGA, requiring hardware developers to implement a Click element as a hardware module using hardware description languages. Subsequently, CUSP[185] and Chimpp[182] proposed a series of improvements, simplifying the interconnection of hardware modules and enhancing the capabilities of software-hardware co-processing and software simulation. However, the above work programs FPGAs using Verilog, VHDL, and other hardware description languages. It is well known that hardware description languages are diffi-

cult to debug, write, and modify, posing significant challenges for software personnel to use FPGAs.

To enhance the development efficiency of Field-Programmable Gate Arrays (FPGAs), manufacturers of these devices offer high-level synthesis (HLS) tools[49-50] capable of compiling restricted C code into hardware modules. However, these tools merely supplement the hardware development toolchain, and programmers are still required to manually insert the hardware modules generated from C language into the hardware description language project. Manual handling is also necessary for the communication between the FPGA and the host CPU. Efficient hardware development languages such as Bluespec[51], Lime[52], and Chisel[53] [54-56] have been proposed by the academic and industrial communities, but their use requires developers to possess substantial hardware design knowledge. Gorilla[183] proposed a domain-specific high-level language for packet switching on FPGAs. While high-level synthesis tools and efficient hardware development languages can enhance the work efficiency of hardware developers, they are still insufficient for software developers to utilize FPGAs.

Click2NetFPGA[186] employs high-level synthesis tools to directly compile the C++ code of Click modular routers[120] to FPGA, thereby providing a modular architecture. However, several bottlenecks exist in the system design of Click2NetFPGA's performance (such as memory and packet I/O), and the code has not been optimized to ensure fully pipelined processing, resulting in a performance that is two orders of magnitude lower than that of this paper. Moreover, Click2NetFPGA does not support FPGA/CPU joint processing, thus it cannot update configurations or read states during data plane runtime.

In recent years, to enable software developers to use FPGA, manufacturers of these devices have proposed OpenCL-based programming toolchains[57-58], offering a GPU-like programming model, as depicted in Figure 4.3. Software developers can offload kernels written in OpenCL language to FPGA.

Nonetheless, this method has its limitations. Multiple parallel executing kernels need to communicate through on-board shared memory, and the DRAM throughput and latency on the FPGA are not ideal, making shared memory a communication bottleneck. Furthermore, the communication model between FPGA and CPU resembles a GPU-like batch processing model. The communication between the host program and the FPGA kernel must always go through the on-board DDR memory. This results in higher processing latency (about 1 millisecond), which is not suitable for network packet processing that requires microsecond-level latency. Thirdly, OpenCL kernel functions

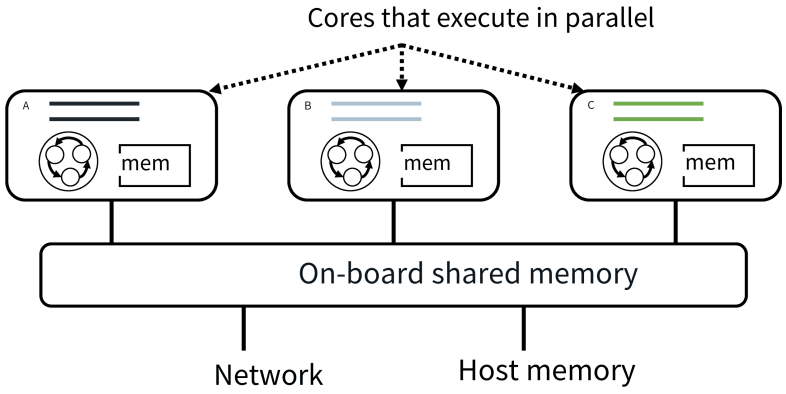


Figure 4.3 Communication method between OpenCL kernels and between kernels and network, host: shared memory.

require explicit calls from software programs on the host machine. Before the kernel terminates, the host program cannot control the kernel behavior, such as setting new parameters, nor can it read any kernel state. However, network functions face a continuous stream of data packets and should always be running. Lastly, OpenCL does not support joint packet processing between CPU and FPGA, and packet processing on the CPU can only be performed outside the OpenCL framework.

The following will introduce ClickNP , a novel FPGA-accelerated network function platform that meets the requirements of flexibility, modularity, high performance, low latency, and CPU/FPGA joint processing.

Network packet processing belongs to stream processing. ST-Accel[187] pointed out that the efficiency of stream processing through FIFO in FPGA is higher than that of shared memory, which can achieve lower latency and higher throughput. For this reason, within the ClickNP framework, the processing logic modules of FPGA and the communication between the network and the host should also be through the FIFO pipeline, as shown in Figure 4.4.

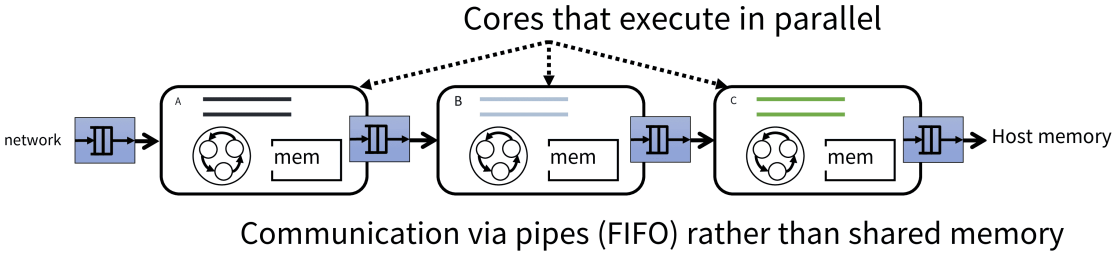


Figure 4.4 Communication method between ClickNP kernels and between kernels and network, host: pipeline (FIFO).

4.2.3 Architectures for Network Processors

Network processors based on general-purpose CPUs such as ClickOS[121] offer excellent programmability, modularity, and composability, but the packet forwarding

performance of a single core cannot keep up with a 10 Gbps line rate for minimum-sized packets, even before any network function is plugged in. Because CPU instructions are executed sequentially and have low parallelism, packet processing performance would drop further as more network functions are added. If a CPU-based network processor is added bump-in-the-wire, there will be tens of microseconds additional end-to-end latency[121], which is one magnitude higher than the switching fabric. In a network virtualization scenario, if packet encapsulation and decapsulation are done at end hosts, as in the case of a virtual switch, network card offloading mechanisms including Large Send Offload (LSO) and Large Receive Offload (LRO) have to be disabled, which has a significant impact on TCP performance[188].

ASICs are known to be high-performance, but the network functions are fixed. Commodity switching ASICs typically have a pipeline of network functions[189], where each function can be configured via registers and a match table based on TCAM or memory. Some ASICs provide flexible OpenFlow-like match-action tables[190], but the packet parser is fixed (we could not support new packet header and shim layer formats), actions are not extensible, and the order of network functions in the pipeline is not reconfigurable.

GPUs are extensively utilized as co-processors for tasks that require intensive computing, but their SIMD (Single-Instruction Multiple-Data) programming model is not suitable for network processing, where different packet types may follow various execution flows. The high power consumption, the high latency of batch processing, and the inability to send and receive network packets without CPU intervention also make GPU-based network processors impractical in data centers.

Fortunately, reconfigurable hardware is an architecture that offers both programmability and high performance, as well as power efficiency for certain workloads. The most notable example of reconfigurable hardware is FPGA (field programmable gate arrays). FPGAs can implement any logic function and use distributed on-chip registers and SRAM to exploit bit-level and task-level parallelism, so stream processing pipelines do not ”hit the memory wall” as in Von Neumann architecture[54]. FPGAs have shown potential in accelerating many workloads in the cloud[48]. Moreover, Moore’s law is still applicable in the FPGA industry, as the fabrication technology of FPGA is currently several generations behind the CPU industry [citation required].

4.2.4 FPGA Programming Challenge

Despite the potential of FPGA in network processing, FPGA's programmability is traditionally provided by hardware description languages (Hardware Description Languages) such as Verilog, which require hardware knowledge and are much more difficult to program and debug than higher-level languages like C/C++. Therefore, existing FPGA-based network processors such as NetFPGA[181] are challenging to program for software engineers.

Many works, for example, OpenFlow[101], P4[106], and Software Defined Network et[191], provide programmability by abstracting a set of primitives in network processing and defining a high-level programming language to compose the primitives. This approach has proven effective, but the programmability is limited to a set of predefined actions, which cannot keep up with the rapid development of data center network functions. Our work aims to make the primitives extensible for software engineers.

Fortunately, several frameworks have been proposed to provide abstractions for generic FPGA programming. Examples of such works include Xilinx Vivado High Level Synthesis[192] based on C/C++, Altera SDK for OpenCL[193] based on C-like OpenCL and IBM Lime[52] based on Java.

However, FPGA has a completely different architecture than general-purpose CPUs. For software programmers that bear Von Neumann model in mind, the compilers may generate surprisingly poor hardware logic for reasonable code in high-level language. For example, Click2NetFPGA[186] uses LLVM and high-level synthesis tools to compile optimized Click C++ code into hardware description language, but the resulting FPGA-based router can only process 178 K pps (packets per second) for 98B packets, and 215 Mbps for large packets, which is 30 – 50x slower than a CPU core in ClickOS[121]. The bottleneck for small packets is the IP header checking stage[186] because this stage is not fully pipelined; the bottleneck for large packets is the byte-wide shared memory[186], indicating a shared-memory design suitable for Von Neumann model would yield poor performance on FPGA.

FPGA has millions of logic gates with 10x slower clock rate than CPU, thousands of distributed fast SRAMs each with only KB capacity, and a large DRAM with 10x lower throughput than DRAMs in CPU architecture. Consequently, exploiting both spatial and temporal parallelism is crucial to unleashing the performance of FPGA. In network stream processing, most operations are independent of each other and therefore can be either parallelized (spatial) or pipelined (temporal), so that each stage of the

pipeline can process different packets in parallel.

4.2.5 Design Goals

We highlight several design goals for our ClickNP framework to enable software engineers to write efficient network applications.

Modularity. Modularity is one key feature that improves parallelism, since modules do not have shared state and can run in parallel by nature. Borrowing the concepts from Click modular router[120], *elements* are basic building blocks of network functions. Elements run asynchronously and are connected via uni-directional *channels*. The network processing pipeline is a data flow graph of elements and channels, starting from Ethernet receivers and ending at Ethernet transmitters.

Line-rate throughput. To allow efficient processing of packet content, an Ethernet packet is split into 32-byte *flits* before feeding into elements. In the worst case, when 69-byte packets are received back-to-back, the line rate would be 40G / 8 / (69+20) = 56.18 Mpps, which splits into 56.18M * 3 = 168.54M flits. Every clock cycle an element reads at most one flit and outputs zero or one flit. This means any FPGA pipeline with clock frequency lower than 168.54 MHz would not be able to achieve line rate. If we waste a cycle between every two packets, the minimum clock frequency would be 224.72 MHz. However, on Stratix V FPGA platform[194], non-trivial hardware logic that accesses registers and local memory can hardly run higher than 200 MHz. Therefore no idle cycles are allowed in elements processing packet content. First, the framework should provide abstractions for programmers to develop fully pipelined network functions. Second, as full compilation of a FPGA program may take hours, the framework should give performance warnings in an early compilation stage if the code cannot be fully pipelined.

Code reuse. Many network applications share a common set of elements, for example packet parser, lookup tables and packet modifications. Code of these elements should be reusable and elements should be composable. Software engineers should be able to write many network applications simply by connecting elements in the library.

Debugging support. First, as hardware description language (e.g. Verilog) simulation and debugging is both time consuming and requires extensive hardware knowledge, the framework should be able to compile OpenCL-based ClickNP programs to native x86 code for emulation, and provide traffic generators and receivers to test functionality. Second, as CPU is neither capable of sending or receiving packets at 60 Mpps, we

need a FPGA-based network benchmark suite to perform stress testing on the network processor.

Separation of control plane and data plane. On one hand, our throughput requirement requires most network packets to be processed through the reconfigurable hardware without any CPU intervention. On the other hand, software-defined networking and network function virtualization applications are usually complicated and have external dependencies. Therefore a clear interface between the control plane and the data plane is mandatory, where data plane programs are written within ClickNP framework and target massive parallelism, and control plane programs need only slight modifications to call our host library and perform on-the-fly reconfigurations.

Host communication. Network processors require low-latency and high-throughput interactions with the host machine. In Software Defined Networks and Network Function Virtualization applications, FPGA needs to send unknown packets to the controller and request a new forwarding rule to be inserted into FPGA. The round-trip time should be as low as possible to reduce end-to-end flow establish time. In packet replay and capture applications, FPGA needs to receive or send Gigabytes of packets from or to the host machine without using the network adapter.

We design ClickNP to meet the above design goals with Catapult FPGA[48] and Altera OpenCL[55]. In the next section, we will describe the FPGA and OpenCL components, and how we build a toolchain that abstracts away hardware specific details.

4.3 System Architecture

4.3.1 ClickNP Development Toolchain

Figure 4.5 depicts the architecture of ClickNP . ClickNP is built upon the Catapult Shell architecture[48]. The Catapult *shell* consists of numerous reusable logic modules that are common to all applications. The shell abstracts them into a set of well-defined interfaces, such as PCIe, Direct Memory Access (DMA), DRAM Memory Management Unit (MMU), and Ethernet MAC. The FPGA program written with ClickNP is compiled into Catapult *user logic* (role). The user logic calls the interfaces provided by the shell to access external resources. Since ClickNP relies on a commercial high-level synthesis toolchain to generate FPGA hardware description language, a *high-level synthesis-specific runtime* (Board Specific Package, BSP) is required to perform the conversion between the high-level synthesis-specific interface and the shell interface.

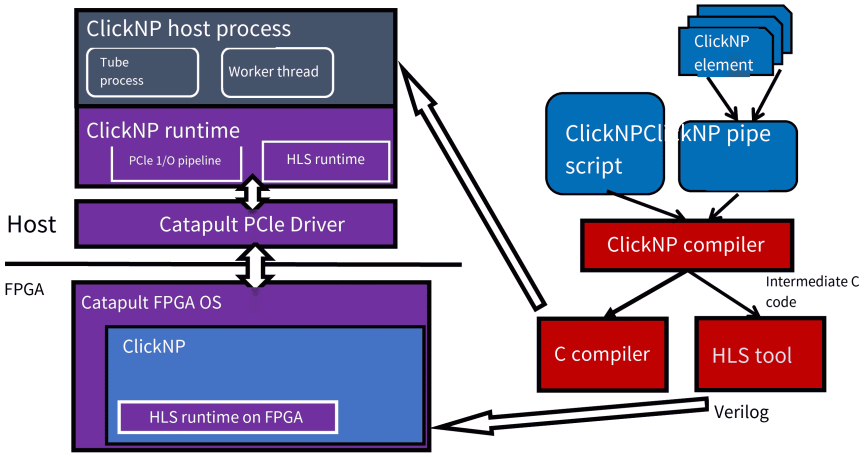


Figure 4.5 ClickNP Architecture.

The ClickNP host process communicates with the ClickNP user logic through the ClickNP runtime library, which further relies on the services in the Catapult PCIe driver to interact with the FPGA hardware. The ClickNP runtime library implements two important functions: (1) It exposes a PCIe channel API to achieve high-speed and low-latency communication between the ClickNP host process and the role; (2) It calls several high-level synthesis-specific libraries to pass initial parameters to the modules in the role and control the start/stop/reset of these modules. The ClickNP host process has a manager thread and zero or more worker threads. The manager thread loads the FPGA image into the hardware, starts the worker threads, initializes the ClickNP components in the FPGA and CPU according to the configuration, and controls their behavior by sending *signals* to the components at runtime. If each worker thread is assigned to a CPU, each worker thread can handle one or more modules.

4.3.2 ClickNP Programming

1. Abstraction

ClickNP provides a modular architecture, where the basic processing module is known as an *element*. As shown in Figure 4.6, ClickNP elements have the following features:

- Local state. Each element can create a set of local variables that can only be accessed within the element itself.
- Input and output ports. Elements can have any number of input or output ports.
- Handler functions. Elements have three handler functions: (1) initialization handler, invoked once when the element starts, (2) processing handler, continuously called to inspect the input ports and process available data, and (3) signal handler, which receives and processes commands (*signals*) from the manager thread in the

host program.

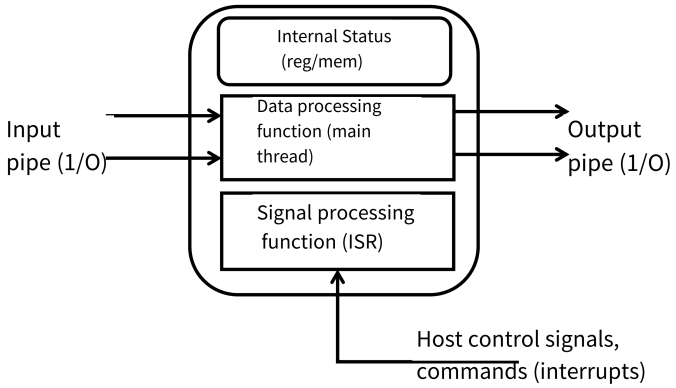


Figure 4.6 Composition of ClickNP elements.

The output port of an element can be connected to the input port of another element through a *channel*, as depicted in Figure 4.7 (a). In ClickNP , a channel is essentially a FIFO buffer, written at one end and read from the other. The data unit of read/write operations on the channel is known as a *flit*, which has a fixed size of 64 bytes. The format of a flit is shown in Figure 4.7 (b). Each flit contains a header of metadata and a payload of 32 bytes. When moving between ClickNP elements, large amounts of data (e.g., full-size packets) are divided into multiple flits. The first flit is marked with sop (start of packet), and the last flit is marked with eop (end of packet). If the size of the data block is not 32, the pad field of the last flit indicates the number of bytes filled into the payload. The reserved fields in the flit have been optimized by the hardware description language synthesis tool. Dividing large data into flits not only reduces latency but also allows different segments of a packet to be processed simultaneously at different elements, increasing parallelism. Finally, to implement network functions, multiple ClickNP elements can be interconnected to form a directed processing graph, known as a ClickNP *configuration*.

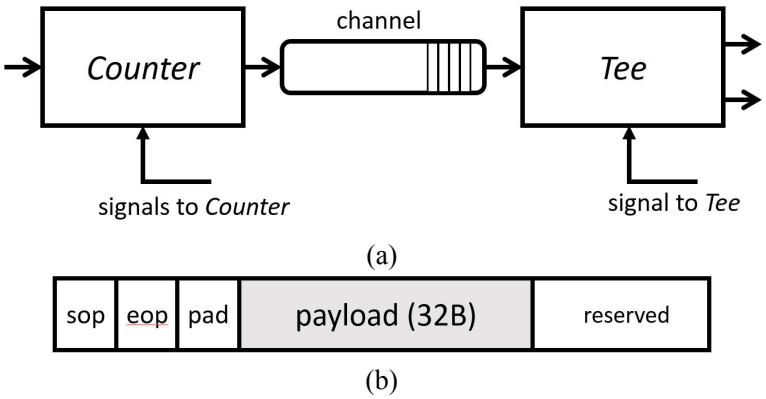


Figure 4.7 (a) Two ClickNP elements connected by a pipeline. (b) The format of Flit.

Obviously, the ClickNP programming abstraction is similar to the Click software

router[120]. However, there are three fundamental differences that make ClickNP more suitable for FPGA implementation: (1) In Click, the edges between elements are C++ function calls, and a *queue* element is needed to store packets. But in ClickNP , the edges actually represent FIFO buffers that can hold actual data. Moreover, ClickNP pipelines can break data dependencies between elements and allow them to run in parallel. (2) Unlike Click, where each input/output port can *write (push) or read (pull)* data, ClickNP has unified these operations: an element can only *write (push)* to the output port, and the input port can only perform *read (pull)* operations. (3) Click allows elements to directly call the methods of another element (through the context of a stream-based router), in ClickNP , coordination between elements is *message-based*, for example, the requester sends a request message to the responder and gets a response through another message. Compared with coordination through shared memory, message-based coordination allows more parallelism and is more efficient in FPGAs, because access to shared memory can become a bottleneck.

2. Language

ClickNP elements can be declared as an object in an object-oriented language (such as C++). Unfortunately, many existing high-level synthesis tools only support the C language. To take advantage of commercial high-level synthesis tools, a compiler can be written to convert an object-oriented language (such as C++) to C, but this effort is not easy. This paper proposes a domain-specific language (DSL) based on a subset of the C language to support element declarations.

The translation of your LaTeX content is as follows:

Figure 4.8 displays a code snippet of the *Counter* element, which solely counts the number of packets that have passed through. The element is defined by the .element keyword, followed by the element name and input/output port declarations. The .state keyword defines the state variables of the element, and .init, .handler, and .signal specify the initialization, data processing, and signal processing function elements. Table 4.1 enumerates the built-in functions for operating on input and output ports.

Similar to Click, ClickNP also utilizes simple scripts to specify the configuration of network functions, as depicted in Figure 4.9. The configuration comprises two parts: *declaration* and *connection*, adhering to a syntax akin to the Click language[120]. It is noteworthy that in ClickNP , the keyword host can be employed to annotate elements, which will result in the elements being compiled into CPU binary files and executed on the CPU.

For elements that some high-level synthesis tools struggle to generate efficient

```
.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);
  }
}
```

Figure 4.8 Code of the packet counter element.

Table 4.1 Built-in operations on ClickNP pipelines.

uint get_input_port()	Acquire a bitmap of all input ports with available data.
bool test_input_port(uint id)	Test the input port indicated by id.
flit read_input_port(uint id)	Read the input port indicated by id.
flit peek_input_port(uint id)	Retrieve the data of the input port indicated by id, but do not remove it.
void set_output_port(uint id, flit x)	Set flit as the output port. Upon the handler's return, flit will be written to the pipeline.
ClSignal read_signal()	Read the signal from the signal port.
void set_signal(ClSignal p)	Set the output signal on the signal port.
return (uint bitmap)	The return value of .handler specifies the bitmap of the input ports to be read in the next iteration.

hardware logic for, ClickNP supports Verilog elements written in hardware description language. To integrate Verilog elements into the system, developers are required to write an element with the same interface as a placeholder (or a high-level language element with the same function for CPU debugging and testing), and declare it in the ClickNP configuration file with the verilog keyword. The compilation toolchain will replace the Verilog module generated by the high-level synthesis tool for the placeholder element with the developer's implementation.

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

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

Figure 4.9 The ClickNP configuration file for the interconnection of packet capture tool elements. Elements annotated with the host keyword are compiled and executed on the CPU. Elements annotated with "@" need to receive control signals from the manager thread. "From_tor" and "to_tor" are two built-in elements, representing the input and output of the Ethernet port on the FPGA.

4.4 Internal Parallelization in FPGA

Fully utilizing the internal parallelism of FPGA is crucial for performance. ClickNP thoroughly exploits the parallelism within and between elements in FPGA.

4.4.1 Inter-element Parallelization

The modular architecture of ClickNP makes it natural to exploit parallelism between different elements. The ClickNP toolchain maps each element to a hardware module in the FPGA. These hardware modules are interconnected through FIFO buffers and can work in complete parallel. Therefore, each element in a ClickNP configuration can be viewed as a tiny independent core with custom logic. Packets flow from one element to another along the *processing pipeline*. This type of parallelism is referred to as *pipeline parallelism*. Furthermore, if a single processing pipeline does not have enough processing power, multiple such pipelines can be replicated in the FPGA, and the data can be divided into these pipelines using load balancing elements, thereby utilizing *data parallelism*. For network traffic, there is data parallelism (at the packet level or flow level) and pipeline parallelism, which can be used to accelerate processing. ClickNP is very flexible and can easily configure both types of parallelism, as shown in Figure 4.10.

Developers can manually specify the number of times inter-element parallelism occurs (i.e., the number of times a certain element is repeated), or they can specify the throughput or area target of the entire network function pipeline or a certain element, and the ClickNP toolchain will automatically calculate the number of times each element is parallelized. ClickNP obtains the average amount of data read from the input pipeline per clock cycle based on the dependency analysis report output by the high-level synthesis tool, and estimates the throughput of the element based on the clock frequency after FPGA synthesis ①; the area is obtained based on the synthesis results of the FPGA.

① Assuming there is no blocking inside the element, i.e., data can be read from the input pipeline in each iteration.

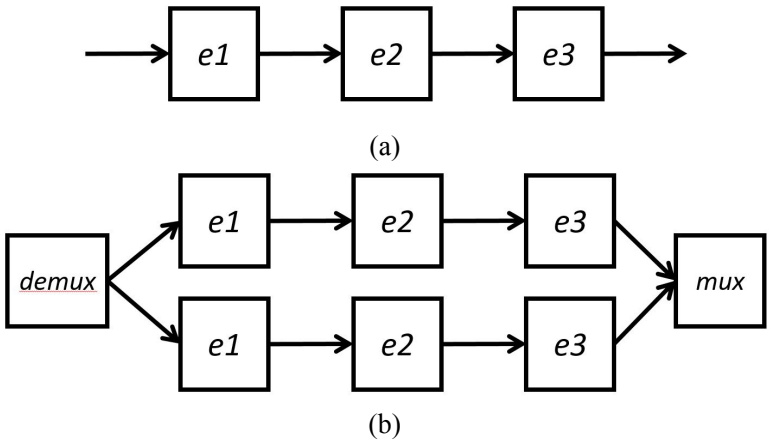


Figure 4.10 (a) Inter-element parallelism. (b) Intra-element parallelism.

The ClickNP toolchain automatically balances the replication times of each element, so that the processing throughput of each element in the pipeline is roughly balanced.

4.4.2 Intra-element Parallelization

Unlike CPUs that execute instructions in memory with limited parallelism, FPGAs synthesize operations into hardware logic, thereby eliminating instruction loading overhead. If data requires multiple related operations within a processing function, the high-level synthesis tool will schedule these operations to pipeline stages in a synchronous manner. In each clock, the result of one stage moves to the next stage, and at the same time, new data is input to this stage, as shown in Figure 4.11 (a). In this way, the processing function can process data in each clock cycle and achieve maximum throughput. However, in reality, pipeline processing may become inefficient in two situations: (1) there is *memory dependency* in the operations; (2) there are *unbalanced* pipeline stages. The following two sections will discuss these two issues in detail and propose solutions.

1. Reducing Memory Dependency

If two operations access the same memory location, and at least one of them is a *write operation*, these two operations are said to be mutually dependent[195]. Because each memory access has a cycle delay, and the semantic correctness of the program largely depends on the order of operations, operations with *memory dependency* cannot be processed simultaneously. As shown in Figure 4.11 (b), S1 and S2 are mutually dependent: S2 must be delayed until S1 ends, and only after S2 is completed, S1 can operate on new input data. Therefore, this function will need two cycles to process a data. For some packet processing algorithms, memory dependencies can be quite complex, but due to the modular architecture of ClickNP , most elements only perform

If the element cannot do this, developers can specify the average number of iterations required to read data each time.

simple tasks, and the memory dependency between *read and write operations* is the most common situation, as shown in Figure 4.11 (b).

One approach to eliminate this memory dependency is to store data exclusively in registers. Given that registers are sufficiently fast to perform read, compute, and write back operations within a single cycle, there is no *read-write* dependency provided the computation process can be completed within one clock cycle. Compared to CPUs, FPGAs possess a significantly larger number of registers, such as the Altera Stratix V which has 697Kbit of registers, thus registers can be utilized to minimize memory dependency as much as possible. When the variable is a scalar, or the variable is an array but all accessed offsets are constants and the array size does not exceed the threshold, the ClickNP compiler implements the variable in registers. Programmers can use the ”register” or ”local / global” keywords to explicitly instruct the compiler to place a variable (which can also be an array) in the register, BRAM, or on-board DDR memory.

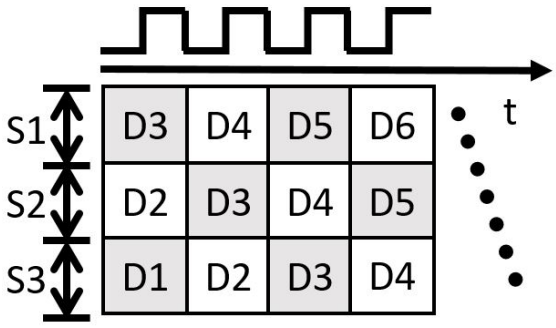
For larger data, they must be stored in BRAM or DDR memory. Variables declared within the component are stored in BRAM by default. Fortunately, a technique called *delayed write* can still be used to mitigate the memory dependency caused by *read-write operations* in Figure 4.11 (b)①. The core idea of delayed write is to alleviate memory dependency by adding temporary storage. Delayed write buffers the new data to be written in the register until the next read operation②. If the next read accesses the same location, it will directly read the value from the buffered register. In this way, read and write operations can be performed in parallel, because read and write operations must access different memory locations.③ Figure 4.11 (c) shows a code snippet of delayed write. Since there is no memory dependency in the code, the component can process one data per cycle, achieving full pipelining. By default, the ClickNP compiler automatically applies *delayed write* to arrays with read-write dependencies, generating code similar to Figure 4.11 (b). If an array with read-write dependencies has multiple read operations, ClickNP will generate code as shown in Figure 4.11 (c) P1 for each read operation. If the array has multiple write operations and these write operations are in mutually exclusive branch conditions, ClickNP will generate intermediate register variables, transforming it into a situation with only one write operation. If the array has

①The result of the write operation in Figure 4.11 (b) may be used by the read operation in the next loop iteration. After expanding adjacent loop iterations, read-after-write and write-after-read are essentially the same dependency.

②The name *delayed write* comes from the transformed code pattern, the physical write operation to memory is not delayed. The hardware logic generated by the delayed write technique is the register forwarding pattern commonly used in pipeline processor design.

③Most BRAMs in FPGAs have two ports, one for read operations and one for write operations, i.e., one random read and one random write can be performed each clock cycle, with a one clock cycle delay for read and write operations.

multiple non-mutually exclusive write operations, ClickNP currently cannot automatically generate delayed write.[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)

Figure 4.11 Examples of memory dependency. (a) No dependency. S*n* represents a stage of the pipeline, D*n* is a data. (b) Memory dependency occurs when state is stored in memory and needs to be updated. (c) Solving memory dependency using delayed write.

A subtle issue arises when using the struct data structure. Figure **??**(a) illustrates such an example, where a hash table is used to maintain the count for each stream. There will be a memory dependency between S2 and S1, even though they are accessing different fields of the struct. The reason is that almost all current high-level synthesis tools treat the struct data structure as a single data with a larger bit width – equal to the size of the struct, and use only one arbiter to control access. This type of memory dependency is termed *pseudo-dependency*. Physically, the two fields *key* and *cnt* can be located at different memory locations. To address this issue, ClickNP employs a technique called *memory scattering*, which automatically converts the struct array into several indepen-

[①]Theoretically, it can be implemented as follows: if there are *N* write operations, *N* registers can be used to save these written values, and all *N* registers are compared when reading. If read and write operations are interspersed, and the generated memory read and write instructions are concentrated in one place, additional registers need to be added to save the intermediate state.

dent arrays, each used to store a field in the struct. Each array is allocated different BRAM, so they can be accessed in parallel (Figure **??**(b)). After implementing *memory scattering*, S1 is no longer dependent on S2, thus eliminating the pseudo-dependency. In general, if all accesses to an array can be divided into several disjoint equivalence classes, with the access address range of each class not overlapping, *memory scattering* can be applied, converting the address range accessed by each equivalence class into an independent array [①]. It is worth noting that memory scattering is only applicable to components in FPGA, and is disabled if the component runs on the host CPU.

2. Balancing Pipeline Stages

Ideally, each stage in a processing pipeline should operate at the same speed, meaning it processes data in one clock cycle. However, if the processes of each stage are unbalanced and some stages require more clock cycles than others, these stages will limit the overall throughput of the pipeline. For instance, in Figure 4.12 (a), S1 is a loop operation. Since each iteration requires one cycle (S2), the entire loop will take N cycles to complete, significantly reducing the pipeline throughput. Figure 4.12 (b) presents another example, where a BRAM cache is implemented for the global table (*gmem*) in DDR. Although the ”else” branch rarely hits, it creates a fat stage in the pipeline (requiring hundreds of clock cycles). The high-level synthesis compiler we use reserves the worst-case number of clock cycles for each stage, so even if the fat stage is rarely used, it greatly affects the processing speed of the entire pipeline.

ClickNP employs two strategies to balance the stages within the pipeline. Firstly, it unrolls loops to the maximum extent possible. Loop unrolling can break down a loop into a series of minor operations that can be executed in parallel or in a pipeline. It's important to note that unrolling a loop will duplicate the operations in the loop body, thereby increasing the area cost. Hence, it may only be suitable for loops with simple loop bodies and a small number of iterations. In network functions, such small loops are quite common, such as calculating checksums, shifting packet payloads, or iterating over various possible configurations. The ClickNP compiler provides an unroll directive to unroll loops.

While many high-level synthesis tools support unrolling loops with known iteration counts, many real-world applications have loops with variable iteration counts. However, in network functions, the upper limit of the loop iteration count can often be determined, such as the maximum length of a packet. Since the iterator of a loop is

[①]For example, in the case of Figure **??**, S1 accesses addresses divisible by 16 remainder 0 7, S2 accesses addresses divisible by 16 remainder 8 15.

```
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)

Figure 4.12 Unbalanced pipeline stages.

often used as an array index, in this case, the upper and lower bounds of the iterator can be determined based on the size of the array ①. There are also some loops where the upper limit of the iterator is a constant or a simple expression composed of other loop iterators, in which case the maximum value of the upper limit expression can be calculated. For cases where the compiler cannot automatically determine the loop iteration count and while loops without explicit iterators, ClickNP allows programmers to specify the upper limit of the loop count through pragma. Once the upper limit of the loop count is determined, ClickNP wraps the loop body with an if statement representing the loop condition, replaces the continue and break statements in the loop body, and then duplicates it.

For the computation shown in Figure 4.12 (a), memory dependencies need to be resolved after loop unrolling, because the variable sum is written and read multiple times. After loop unrolling, ClickNP expands scalar variables into *static single assignment* form, so that each variable is assigned only once, which can eliminate this kind of memory dependency. Essentially, static single assignment and delayed writing both increase memory space to improve parallelism.

① ClickNP does not support dynamic memory allocation, so the size of all arrays can be statically determined at compile time.

The second technique involves separating different types of operations within a single component if it has both fast and slow operations. For instance, in a cache component's implementation, as depicted in Figure 4.12 (b), the slower "else" branch is relocated to another component. This allows the fast path and slow path to operate asynchronously. If the cache miss rate is extremely low, the entire component's processing speed is determined by the fast path. As shown in Figure 4.13, the ClickNP compiler offers an "async" primitive. Users can insert code blocks enclosed in async { } in the handler. The code inside will be compiled into a new component and connected to the original component through a pipeline. The original component serializes and sends the variables used in the asynchronous component, then waits for the asynchronous component to complete. Once the asynchronous component is finished, it sends the written variables that the original component will continue to use back to the original component.

```
.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);
}
```

Figure 4.13 Example of Async primitive.

4.5 System Implementation

4.5.1 ClickNP Toolchain and Hardware Platform

This chapter implements a ClickNP compiler as the front end of the ClickNP toolchain (§**??**). For the host program, Visual C++ is utilized as the backend. Further integration of the Altera OpenCL SDK (i.e., Intel FPGA SDK for OpenCL)[57] and Xilinx SDAccel[49] as the backend for FPGA programs. The core part of the ClickNP compiler contains about 20,000 lines of C++, flex, and bison code, which parse configuration files and component declarations, perform the optimizations described in Section

4.4, and generate code specific to each commercial high-level synthesis tool.

For components running on the FPGA, each component is compiled into intermediate C code, which is then compiled into a logic module by a high-level synthesis tool. When using the Altera OpenCL high-level synthesis tool, each ClickNP component is compiled into a *kernel*, the connections between components are compiled into Altera extended channels (channel), which are then implemented with the Avalon ST interface; the components communicate with the on-board DRAM (i.e., global memory) using the Avalon MM interface. When using the Xilinx SDAccel high-level synthesis tool, each component is compiled into an IP core, and the connections between components are implemented using AXI streams, and the AXI memory-mapped interface is used to access the on-board DRAM. Components running on the host are compiled into CPU binary files, and the management process creates a worker thread for each host component. Each pipeline between the host and FPGA components is mapped to a *slot* of the PCIe I/O channel (§4.5.3).

The hardware platform of this paper is based on the Altera Stratix V FPGA and Catapult shell[48]. The Catapult shell also includes an OpenCL-specific runtime (BSP). ClickNP user logic communicates with the shell through the BSP. ClickNP user logic runs in an independent clock domain, and the BSP converts interfaces such as PCIe DMA and DRAM in the shell in different clock domains to the user logic's clock domain through asynchronous FIFO. The BSP also provides management functions such as OpenCL kernel start and stop. At the time of writing this paper, the author has not yet obtained the Xilinx hardware platform. Therefore, the system evaluation is mainly based on the Altera platform using ClickNP + OpenCL, and the reports generated by Vivado HLS (such as frequency and area costs) are used to understand the performance of ClickNP + Vivado.

1. Intermediate C Code Suitable for High-Level Synthesis Tools

Upon powering on the host or online reconfiguration of the FPGA, each kernel or IP core commences parallel operation. As depicted in Figure 4.14, each kernel initially executes the initialization (init) function, then enters an infinite loop, checks the input pipeline and executes the event handling (handler) function, checks the signal and executes the event handling function.

High-level synthesis tools convert the intermediate C code into a hardware description language. Each loop in the intermediate C code is either fully unrolled into a pipelined logic module or implemented as a state machine, with each clock cycle executing one iteration of the loop. Loop unrolling is only applicable when the number of

```
void kernel() {
  Call init function;
  Declare and initialize input and output buffers;
  while (true) {
    if (input buffer is free and input pipeline is not empty) {
      Move data from input pipeline to input buffer;
    }
    Call handler function;
    if (output pipeline is free or output buffer is full) {
      Move data from output buffer to output pipeline;
    }
    if (input event pipeline is not empty) {
      Read input event from input event pipeline;
      Call signal function;
      Write event handling response to output event pipeline;
    }
  }
}
```

Figure 4.14 Pseudocode of Kernel Intermediate C Code.

loop iterations is statically known at compile time and the number of iterations is small. For loops implemented as state machines, there may be data dependencies between different iterations of the loop, and the pipelined logic of one iteration may take several clock cycles to complete, so there may be several clock cycles of interval between two consecutive loop iterations. High-level synthesis tools need to calculate the minimum interval (initiation interval, II) between two consecutive iterations based on dependency relationships and pipeline delay information of the loop body. The smaller the II, the higher the throughput. Therefore, we aim to minimize II as much as possible.

Currently, high-level synthesis tools are primarily designed for compute-intensive operations. These operations often consist of multi-level nested loops, and the loop transformation methods used by compilers are typically applicable to programs with static control parts and perfectly nested loops during control flow compilation. However, the number of times the while loop in the program in Figure 4.14 is executed is unknown at compile time, and due to the presence of input/output buffer movement code, the loops in the handler and signal functions are not perfectly nested loops. Early versions of high-level synthesis tools may not only fail to compile, take too long to compile, and other errors for more complex source programs due to completeness issues, but also analyze too many unnecessary memory dependencies, leading to a large II, or even cause II static analysis failure, and the inner loop cannot be parallelized.

This paper aims to circumvent the nested loop optimization of existing high-level synthesis tools. Observing that the component calculation logic in network functions is relatively simple, and the amount of data processed per clock cycle is also very limited

(such as one flit), many loop execution times can be statically determined at compile time (for example, processing each byte of a flit). Therefore, ClickNP unrolls or flattens all loops in the handler and signal functions, so that the generated intermediate C code only has a while loop, and there are no nested loops, as shown in Figure 4.15. The default strategy of ClickNP is to unroll loops that can be determined at compile time and flatten all other loops. Users can also specify the unrolling and flattening strategy through compilation options (pragma) embedded in the source program. In this way, high-level synthesis tools only need to analyze a single loop, reducing the possibility of errors.

Unrolling the loop body also has a significant advantage: it facilitates compiler optimization. Traditional compilers often find it difficult to optimize vector operations represented by loops. After ClickNP unrolls the loop, the vector operation is decomposed into point-by-point operations, and high-level synthesis tools can perform a series of optimizations such as constant propagation and dead code elimination. In addition, after unrolling the loop, ClickNP can perform static single assignment transformation, and can also expand the array with access addresses determined by loop variables into several discrete registers, thereby eliminating memory dependencies.

The ClickNP can generate performance analysis reports. Within each component, the analysis report includes the storage method of each variable, the unrolling or flattening strategy of each loop, the minimum interval (II) between two adjacent iterations, and the dependency chain causing the II bottleneck. At the computational graph level, the analysis report includes the delay, throughput, and clock frequency of each component.

2. Optimization of Compilation Speed

One limitation of FPGA programming is the relatively long compilation time. A simple network packet forwarding function requires about 3 hours to compile. The compilation time is mainly composed of several stages such as high-level synthesis, IP core generation, hardware description language logic synthesis, FPGA layout routing, and timing analysis. This chapter adopts several techniques to shorten the FPGA compilation time. First, the OpenCL programming framework generates IP cores from the Verilog modules generated by high-level synthesis and inserts them into the shell part, which requires copying a large amount of IP core and shell part code. Noting that the peripheral interface of user logic is fixed, this paper pre-generates IP cores and only needs to replace the high-level synthesis Verilog module files into the project; for the shell part code, file references are used instead of copying. Second, in order to shorten the logic synthesis time, the fixed shell part is synthesized into a netlist file through

```
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) Original C intermediate code (schematic code).

```
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) The result after loop unrolling and flattening.

Figure 4.15 Loop unrolling and flattening. The i loop is unrolled according to the parallelism 2 and flattened; the j loop is fully unrolled: the result of the transformation is that all bytes of 2 flits are calculated per clock cycle.

logic synthesis, and the synthesis result is retained by using design partition, which can reduce the logic synthesis time of the shell part by about 35 minutes. Third, in order to speed up the convergence speed of the FPGA layout algorithm, after the initial compilation is completed, the layout constraints of each module of the shell part are added to basically fix its layout. Most of the modules in the shell part interact with the hard IP at fixed positions on the chip, so it is reasonable to fix the layout near the hard IP. However, for better performance, this paper does not use design partitions to completely fix the layout and routing of the shell part. This optimization can save about 20 minutes. Fourth, distinguish between debug and release compilation modes. The debug mode aims to verify the correctness of the logic, not to pursue performance. In debug mode, the clock frequency of user logic is fixed at 50 MHz, which greatly reduces the diffi-

culty of layout and routing; the layout and routing of the shell part are solidified using design partitions, and the layout and routing of this part of high-frequency logic takes a long time. The debug mode can save 25 minutes of compilation time compared to the release mode, and more time will be saved when the user logic is more complex. Fifth, unnecessary timing analysis models are deleted. The OpenCL framework defaults to analyze timing constraints under four conditions, but as long as the most stringent one is met, the other three can also be met, so we only retain the most stringent timing constraint model. Sixth, in order to achieve the highest possible performance, the OpenCL framework first compiles with a higher user logic clock frequency (such as 250 MHz), then calculates the longest delay and the highest clock frequency that can work correctly based on the timing analysis results, and then uses this clock frequency for secondary layout and routing and timing analysis. This can achieve the highest possible throughput for compute-intensive workloads. However, this paper focuses on network packet processing, and only needs to achieve network line speed processing capabilities, so it can fix the clock frequency at 180 MHz, most user logic can reach this frequency, so there is no need to re-layout and route.

Table 4.2 FPGA compilation acceleration technology.

Compilation stage	Optimization method	Compilation time before optimization (min)	Compilation time after optimization (min)
ClickNP compilation	–	0.1	0.1
High-level synthesis	–	1	1
Generate IP core	Pre-generate IP core; use file reference instead of copying	10	0
Logic synthesis	Retain the synthesis result of the shell	50	15
Layout and routing	Add layout constraints of the shell; in debug mode, lower the clock frequency of user logic, retain the layout and routing results of the shell	60	40 (15)*
Timing analysis	Delete unnecessary timing analysis models	15	5 (0)*
Secondary layout and routing	Fix the clock frequency, no need to re-layout and route	30	0
Secondary timing analysis	Delete	15	0
Total	–	180	60 (30)*

* The number in parentheses is the compilation time in debug mode.

Table 4.2 summarizes the above compilation acceleration techniques. Before optimization, developers could only debug 3 rounds per working day, but after optimization, they can debug about 10 rounds in debug mode, and system tests that require performance can also be conducted about 6 rounds, greatly improving work efficiency.

4.5.2 ClickNP Component Library

This paper implements a ClickNP component library containing nearly 200 components. Some of them (about 20These components cover a large number of basic operations of network functions, including packet parsing, checksum calculation, encapsulation/decapsulation, hash table, longest prefix matching (LPM), rate limiting, encryption, and packet scheduling. Due to the modular architecture of ClickNP , the code size of each component is moderate. The average line of code (LoC) of the components is 80, and the most complex component *PacketBuffer* has 196 lines of C code①.

The table referred to as **??** showcases a selection of key elements implemented in ClickNP . Along with the name of each element, the table also denotes the demo network functions that employ each element (these are elaborated further in section 4.6). The optimization techniques previously discussed in section 4.4.2 are utilized to minimize memory dependencies and balance pipeline stages. The third column of the table provides a synopsis of the optimization techniques employed by each element.

For the elements listed at the top of Table **??**, the processing logic within the elements necessitates access to every byte of the packet. The throughput for these elements is provided in Gbps. However, the elements listed at the bottom of the table only process packet headers (metadata), hence it is more suitable to measure their throughput in packets per second. It is crucial to note that the throughput measured in Table **??** represents the maximum throughput that the corresponding elements can attain. When these elements are employed in real network functions, other components, such as Ethernet ports, may become the bottleneck.

For reference, Table **??** compares the optimized FPGA version, a simple FPGA implementation that does not apply the techniques described in section 4.4, and a CPU implementation. The table clearly demonstrates that, post optimization, all of these elements can process packets very efficiently, achieving speeds that are 7 to 117 times faster than the initial FPGA implementation and 21

Considering the power consumption of the FPGA (approximately 30W) and the CPU (approximately 5W per core), the energy efficiency of the ClickNP elements is 4 to 120 times greater than that of the CPU. Table **??** also provides information on the processing delay of each element. As can be observed, this delay is very low, with an average of 0.19μs and a maximum of only 0.8μs (LPM _Tree).

The final two columns of the table provide a summary of the resource utilization

① The number of lines of code refers to the ClickNP component code, excluding the host control program and test code.

of each element, with utilization normalized to the capacity of the FPGA chip. Most elements only use a small number of logic elements, which is to be expected given that most packet operations are simple. The HashTCAM and RateLimiter elements have moderate logic resource usage due to their larger arbitration logic. However, BRAM usage largely depends on the configuration of the element. For example, the rate of BRAM usage increases linearly with the number of entries supported in the flow table.

In summary, the FPGA chip used in this study has sufficient capacity to support meaningful network functions that contain a small number of elements.

4.5.3 PCIE I/O Channel

As previously stated, a key characteristic of ClickNP is its support for combined CPU / FPGA processing. This chapter accomplishes this objective by designing a high-throughput, low-latency PCIe I / O channel. ClickNP supports flexible I/O operations. As depicted in Figure 4.16, ClickNP offers two abstractions for communication with the host CPU based on slots and work queues, and also provides an interface for raw DMA operations.

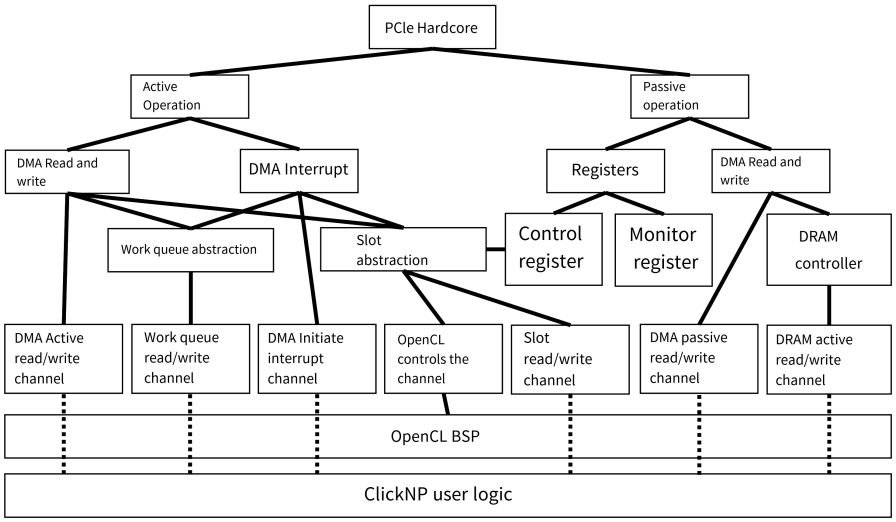


Figure 4.16 Architecture of the PCIe I/O channel.

In the slot-based abstraction, the PCIe physical link is divided into 64 logical sub-channels, or *slots*. Each slot has a pair of send and receive buffers for DMA operations. Of the 64 slots, 33 are utilized by OpenCL BSP for managing ClickNP kernels and accessing on-board DDR (i.e., OpenCL control channels), and one slot is used to transmit signals to ClickNP elements. The remaining 30 slots are used for data communication between FPGA and CPU elements. The slot abstraction on the CPU can operate in either interrupt or polling mode.

Each data sent in the slot abstraction necessitates at least 4 DMA operations ①, and it needs to wait for the device on the other side to complete processing before it can send the next data in the same slot. To amortize DMA overhead and increase the concurrency of message sending, the work queue is an extension of the slot abstraction. Each slot no longer has only one pair of buffers, but a pair of ring buffer queues for sending and receiving. Each ring buffer queue has 128 slots and is accessed in a first-in, first-out manner. When the sender finds that there is still data in the ring buffer queue that has not been taken away, there is no need to notify the other party, saving the overhead of the CPU sending the doorbell through PCIe MMIO and the FPGA sending the interrupt.

In addition to slots and work queues, more flexible communication methods are required between the FPGA and CPU. Firstly, in the key-value storage in Chapter 5, the FPGA needs to directly read and write the key-value in the host memory without the involvement of the host CPU. This necessitates the FPGA to be capable of directly issuing raw PCIe DMA read and write requests. Secondly, in memory disaggregation based on programmable network cards, the FPGA directly maps remote memory to the host memory space via PCIe MMIO. The host CPU directly accesses this memory space, generating PCIe DMA read and write requests sent to the FPGA. The user logic in the FPGA needs to handle these DMA passive read and write operations. Lastly, some applications (such as traditional OpenCL applications) may prefer that the host CPU and FPGA use the DRAM on the FPGA board as shared memory, so the DRAM on the FPGA board is mapped to the host memory space via PCIe MMIO, and is sent to the DRAM controller by the PCIe passive read and write logic in the shell. Since the efficiency of the host PCIe MMIO reading and writing large blocks of data is low, it also supports the host CPU through the control register, allowing the FPGA shell to actively initiate DMA to move data between the board DRAM and the host memory.

Figure 4.17 displays the benchmark results of the PCIe I/O channel with varying numbers of slots and batch sizes. As a baseline, the performance of OpenCL global memory operations is also measured – to date, this is the only method provided by OpenCL[196] for communication between the CPU and FPGA. In Figure 4.17, it can be observed that the maximum throughput of a single slot is approximately 8.4 Gbps.

① The process of sending data from the host CPU to the FPGA is: the host CPU writes the downlink control register in the FPGA (also known as the doorbell); the FPGA DMA reads data from the host memory. When the FPGA has processed the data in the slot, it writes the downlink completion register in the host memory and sends an interrupt to the host CPU. The process of sending data from the FPGA to the host CPU is: the FPGA reads the internal uplink control register and judges it to be empty; the FPGA DMA writes data to the host memory and sends an interrupt to the host CPU. The process of the host receiving data sent by the FPGA is: read the uplink control register in the FPGA and judge it to be non-empty; read the data in the host memory; write the uplink control register in the FPGA, indicating that the processing is complete.

Through 4 slots, the total throughput of the PCIe I/O channel can reach 25.6 Gbps ①. However, the throughput of OpenCL is surprisingly low, even less than 1 Gbps. This is because the global memory API is designed to transfer GB-level large amounts of data. This may be suitable for applications with large data sets, but it is not suitable for network functions that require strong stream processing capabilities. Similarly, Figure 4.17 (b) shows the communication latency. Since OpenCL is not optimized for stream processing, the OpenCL latency is as high as 1 ms, which is generally unacceptable for network functions. In contrast, the PCIe I/O channel has a very low latency of 1 μs in polling mode (a CPU core continuously polls the status register), and the latency in interrupt mode is 9 μs (almost no CPU overhead).

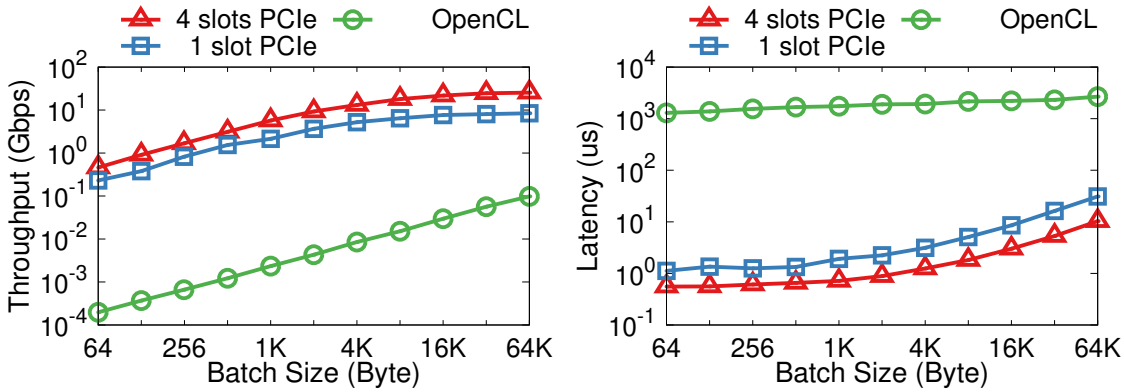


Figure 4.17 Performance of PCIe I/O channels. The Y-axis is a logarithmic coordinate system.

In order to send signals to FPGA components, the ClickNP compiler automatically generates a special component called *CmdHub* in the FPGA. *CmdHub* distributes the control signals issued by the host management program to FPGA components through pipelines, and the FPGA components return the results of the signal processing functions to *CmdHub* through pipelines, which in turn return to the host management program. To avoid the complexity of layout and wiring brought about by one-to-many pipeline connections, *CmdHub* forms a daisy chain with all components, starting from *CmdHub*, passing through all components in the topological order of the component connection diagram, and finally returning to *CmdHub*. In order to identify the target component in the daisy chain, the component ID is embedded in the signal message, and each component only processes the signal message matching the component ID, and directly forwards other signal messages.

①This is the actual maximum performance of the PCIe Gen2 x8 hard core used at the time of writing this chapter. In fact, this FPGA supports the PCIe Gen3 x8 hard core. Chapter 5 achieves 2 times the PCIe I/O channel throughput by replacing the hard core and optimizing the shell.

4.5.4 Debugging

ClickNP provides two methods for debugging.

CPU function simulation. ClickNP components are written in class C high-level language, so a component can be compiled into a thread running on the CPU, and the pipeline is the queue between threads. Developers can use familiar software debugging tools for function simulation.

Actual FPGA operation. CPU function simulation has limitations. Firstly, there is a possibility of deadlock in the communication pipeline between components. During CPU function simulation, due to the inconsistency of timing and hardware logic, deadlock problems may not be discovered; secondly, CPU simulation speed is slow, it cannot reflect actual performance, and it is difficult to test the interaction with PCIe DMA and the network; finally, function simulation cannot discover errors in the compiler. Therefore, in actual applications, after the function simulation passes, it is generally debugged by the method of actual FPGA operation.

Since the variable names in the ClickNP language do not correspond with those in Verilog, online FPGA debugging tools such as SignalTap are not applicable. Therefore, ClickNP requires a customized debugging mechanism. Firstly, in debug mode, ClickNP can record the input and output logs of each pipeline, and transmit them to the host memory via the PCIe I/O pipeline. Secondly, ClickNP allows users to insert printf statements in the component code, and send debug information containing variable values to the host through the pipeline. Thirdly, ClickNP supports users to insert breakpoints at compile time for debugging interactive network protocols or simulating queue blocking leading to deadlock. Breakpoints are compiled into pipeline write operations (notifying the host that the breakpoint has been hit) and blocking read operations (waiting for the host to send the breakpoint continue command). Fourthly, ClickNP allows querying or modifying the value of a variable at any time during operation (including when a breakpoint is hit). When a user queries the value of a variable, a query or modification command is sent through signal, and the value of the variable is returned.

4.5.5 Component Hot Migration and High Availability

Hot migration is a crucial feature that data center network functions need to support. When a virtual machine is hot migrated, the internal state of the corresponding network card on the compute node needs to be hot migrated to the new node, otherwise, it is necessary to reinitialize the network card state on the new node, which brings complexity to the software and migration delay. Similarly, when network and storage

nodes are hot migrated due to upgrades, expansions, etc., the network card state also needs to be hot migrated to the new node. In addition, in order to achieve uninterrupted network function upgrades, the internal state needs to be hot migrated to another node, and then the original node is taken offline for upgrades. To achieve high availability, when adding a backup node, in order to synchronize the internal state of the source node and the newly added backup node, hot migration technology is also needed.

The hot migration process commences with the configuration of the switch to stop sending packets to the old FPGA, instead buffering these packets within the switch. The next step involves halting each component within the FPGA through the breakpoint mechanism, as discussed in section 4.5.4. Following this, all variable values within the components, data in the pipeline, and values in the global memory are exported to the host. The same ClickNP program is then run on the new FPGA, importing the aforementioned internal state of the FPGA through the debugging mechanism, and resuming the operation of each component. Finally, the switch is instructed to modify the routing table, redirecting the address of the old FPGA to the port where the new FPGA is located, and sending the buffered packets in the switch. This concludes the hot migration process.

To ensure high availability of network functions, ClickNP employs the method of state machine replication. Two FPGAs receive the same sequence of packets. Provided there is no randomization or time-related processing logic within the component, and it does not accept control signals from the host, it can be ensured that the internal state of the two FPGAs and the sequence of packets sent out are identical. In the event of a backup node failure, a new backup node is simply initiated, followed by state hot migration. In the event of a primary node failure, a switch to the backup node is necessary. At this point, a small number of input packets may be lost or output packets may be repeated, but these situations can be safely handled by TCP.

4.6 Applications and Performance Evaluation

To assess the adaptability of ClickNP , several standard network functions were developed based on ClickNP , which can operate in the testbed of this study. Table 4.3 encapsulates the number of elements and total lines of code included in each network function, encompassing all element specifications and configuration files. It has been confirmed that the modular architecture of ClickNP significantly enhances code reusability and simplifies the development of new network functions. As depicted in

Table **??**, there are numerous opportunities to reuse an element in many applications, for instance, all network functions in this study utilize L4_Parser. Each network function may take approximately 1 hour for a programmer to develop and debug. The capability to process in conjunction with CPU / FPGA will also significantly aid debugging, as problematic elements can be transferred to the CPU for easy log printing to trace issues.

This chapter evaluates ClickNP in a testbed of 16 Dell R720 servers. For each FPGA board, two Ethernet ports are connected to the Top-of-Rack Dell S6000 switch[197]. All ClickNP network functions operate on Windows Server 2012 R2. This chapter contrasts ClickNP with other cutting-edge software network functions. For those network functions operating on Linux, CentOS 7.2 with kernel version 3.10 is utilized. The test employs the PktGen packet sending tool to generate test traffic at varying rates with different packet sizes (64B packets, maximum throughput is 56.4 Mpps). To measure the processing delay of network functions, a generation timestamp is embedded in each test packet. When the packets traverse the network function, they are looped back to the PktCap packet capture tool, which is located in the same FPGA as PktGen. Then the delay can be determined by subtracting the generation timestamp from the packet reception time. The delay caused by PktGen and PktCap is pre-calibrated by direct loopback (without network function) and removed from the data.

The following sections introduce the network functions based on ClickNP in sequence.

4.6.1 Packet Generator and Packet Capture Tools

The Packet Generator (PktGen) can generate various traffic patterns based on different configuration files. It can produce streams of different sizes and schedule them to start at different times according to a given distribution. The generated streams can be further controlled in terms of flow rate and burstiness through different traffic shapers.

The Packet Capture Tool (PktCap) redirects all received packets to *logger* components, which are typically located in the host. Figure **??** illustrates the component structure of the packet capture tool. Since a single packet capture component cannot fully utilize the capacity of the PCIe I/O channel, PktCap implements a Receive Side Scaling (RSS) component in the FPGA to distribute packets to multiple packet capture components based on the hash value of the flow 5-tuple. Due to the throughput of the PCIe channel being less than the throughput of the 40G network card, an *extractor* component is added, which only extracts important fields of the packet (for example, if there are 5 tuples, DSCP and VLAN tags), and forwards these fields (a total of 16B) and

the timestamp (4B) via PCIe. PktCap is an example of demonstrating the importance of joint CPU / FPGA processing. Compared with FPGA, the CPU has more memory for buffering and can easily access other storage, such as the HDD / SSD drives in the literature[198], so it is more logical to run the logger on the CPU.

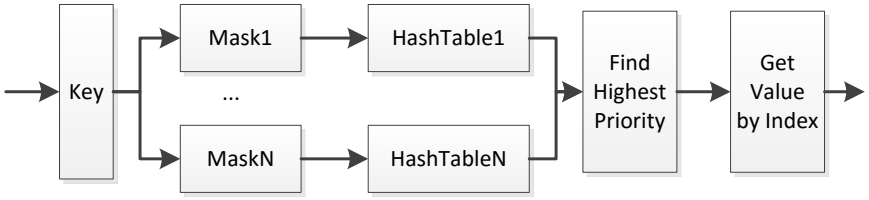


Figure 4.18 HashTCAM.

This experiment compares the OpenFlow firewall with the Linux firewall and Click + DPDK[199]. For Linux, IPSet is used to handle exact match rules, while IPTable is used for wildcard rules. Also included as a reference is the performance of the Dell S6000 switch, which has limited firewall functionality and supports 1.7K wildcard rules. It is worth noting that the original Click + DPDK[199] does not support Receive Side Scaling (RSS). This chapter fixes this problem and finds that when using 4 cores, Click + DPDK has already achieved optimal performance. But for Linux, using as many cores as possible (up to 8 cores due to RSS limitations) can achieve optimal performance.

Figure 4.19 (a) shows the packet processing rate of different firewalls with different numbers of wildcard rules. The packet size is 64B. It can be seen that both ClickNP and S6000 can reach a maximum speed of 56.4 Mpps. Click + DPDK can reach about 18 Mpps. Since Click uses a static classification tree to implement wildcard matching, the processing speed does not change with the number of inserted rules. Linux IPTables has a low processing speed of 2.67 Mpps and the speed decreases as the number of rules increases. This is because IPTables performs linear matching for wildcard rules.

Figure 4.19 (b) shows the processing latency under different loads using small packets (64B) and 8K rules. Since each firewall has significantly different capacities, the load factor is normalized to the maximum processing speed of each system. At all load levels, FPGA (ClickNP) and ASIC (S6000) solutions have μs level latency (ClickNP is 1.23 μs, S6000 is 0.62 μs), with very small variance (ClickNP is 1.26 μs, for S6000 95

Finally, Figure 4.19 (d) illustrates the latency of rule insertion when there are 8K rules. Click's static classification tree requires prior knowledge of all the rules, and generating a tree with 8K rules takes one minute. IPTables rule insertion takes 12 ms, which is proportional to the number of existing rules in the table. Rule insertion in Dell S6000

takes 83.7 μs. For ClickNP , inserting a rule in the HashTCAM table takes 6.3 to 9.5μs for 2 to 3 PCIe round trips, while the SRAM TCAM table takes an average of 44.9 μs to update 13 lookup tables. The data plane throughput of ClickNP does not decrease during rule insertion. The conclusion is that the ClickNP firewall has similar performance to ASIC in packet processing, but has better flexibility and reconfigurability compared to ASIC.

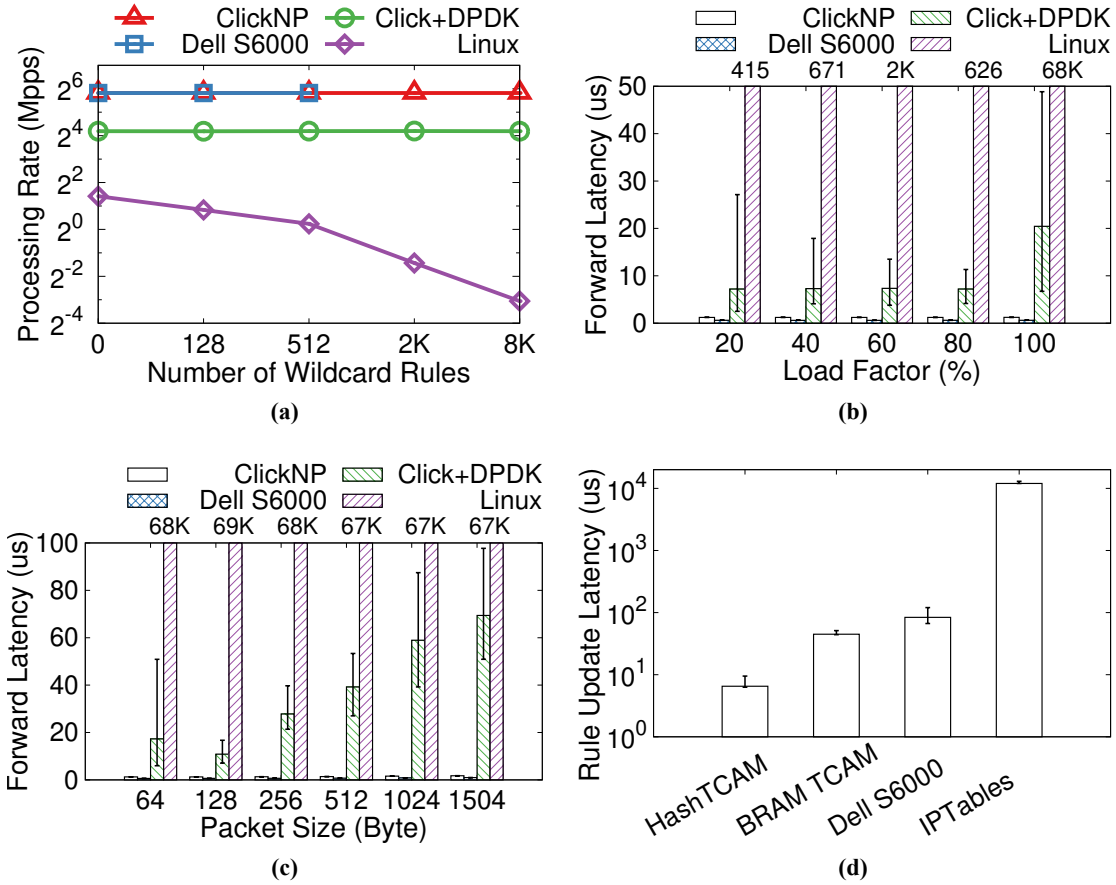


Figure 4.19 Firewall. The error bar represents the latency of the 5% and 95% percentiles. In figures (a) and (b), the packet size is 64 bytes.

4.6.2 IPSec Gateway

One issue with software network functions is that the CPU quickly becomes a bottleneck when packets require some computationally intensive processing, such as IPSec[36]. The IPSec data plane needs to use AES-256-CTR encryption and SHA-1 authentication to process IPSec packets. As shown in §4.5.2, a single AES_CTR component can only achieve a throughput of 27.8 Gbps. Therefore, two AES_CTR components need to run in parallel to achieve line speed. However, SHA-1 is tricky. SHA-1 divides the packet into smaller data blocks (64B). Although the computation within a data block can be pipelined, there is a dependency between consecutive blocks

within an IP packet - the computation of the next block cannot start before the previous block is completed! If these data blocks are processed in sequence, the throughput will be as low as 1.07 Gbps. Fortunately, the parallelism between different packets can be utilized. Although the processing of data blocks in the current packet is still in progress, data blocks from different packets are provided. Since these two data blocks have no dependencies, they can be processed in parallel. To achieve this, we designed a new component called *reservo* (short for reservation station), which can buffer up to 64 packets and schedule independent blocks for the SHA-1 component. After calculating the signature of a packet, the *reservo* component sends it to the next component that attaches the SHA-1 HMAC to the packet.

There is another intricate issue. Although the SHA-1 component has a fixed delay, the total delay of the packets varies, that is, it is proportional to the size of the packet. When scheduling multiple packets in the SHA-1 calculation, these packets may be re-arranged, for instance, smaller packets behind a large packet may be completed earlier. To ensure the output packets maintain the same order as the input, a *reorder buffer* component is further added after the SHA-1 component, which stores out-of-order packets and restores the original order according to the sequence number of the packets. Figure 4.20 illustrates the component structure of the IPSec gateway.

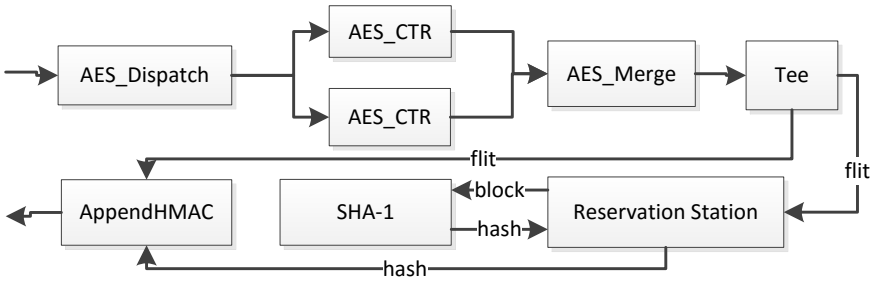


Figure 4.20 Component architecture of the IPSec gateway.

The following compares the IPSec gateway and StrongSwan[200], using the same cipher suite AES-256-CTR and SHA1. In the case of a single IPSec tunnel, Figure 4.21(a) displays the throughput of different packet sizes. For all scales, IPSecGW achieves line rate, that is, 64B packets are 28.8 Gbps, and 1500B packets are 37.8 Gbps. However, StrongSwan can only reach up to 628 Mbps, and as the packet size decreases, the throughput will also decrease. This is because the smaller the size, the more packets need to be processed, so the system needs to calculate more SHA1 signatures. Figure 4.21(b) shows the latency under different load factors. Similarly, the constant delay produced by IPSecGW is 13 μs, but StrongSwan will produce a larger delay and higher variance, up to 5 ms!

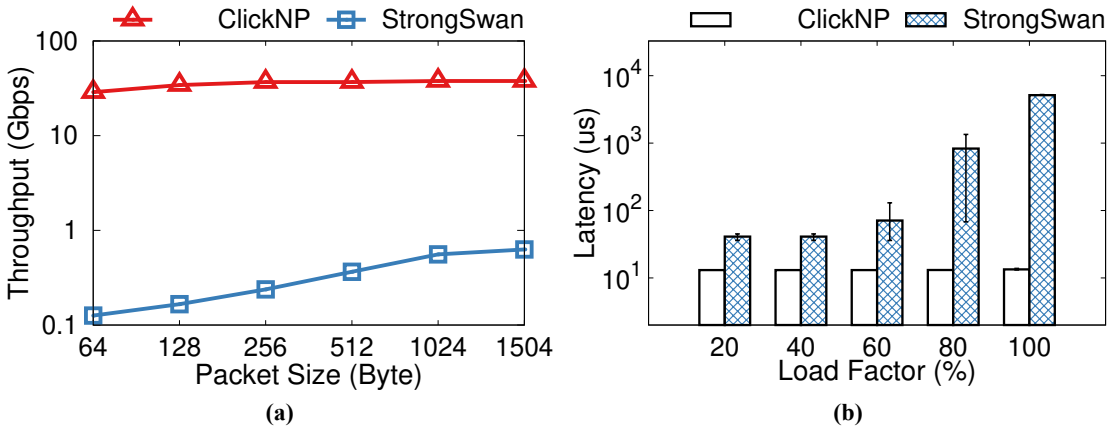


Figure 4.21 IPSec gateway.

4.6.3 L4 Load Balancer

The L4 load balancer is implemented according to the *multiplexer* (MUX) in Ananta[7]. The MUX server essentially examines the packet header and checks whether a *direct address* (DIP) has been allocated for this stream. If so, the packet is forwarded to the server indicated by the DIP through the NVGRE tunnel. Otherwise, the MUX server will call the local controller to allocate a DIP for the stream. The MUX server needs to maintain the state by stream. Since there are failures and the backend server list needs to be updated in real time to avoid black holes, hash-based ECMP cannot be used. In addition, advanced LB may also require load-aware balancing. The flow table is used to record the mapping of the flow to its DIP. To handle the large traffic in the data center, it requires the L4LB to support up to 32 million streams in the flow table. Clearly, such a large flow table cannot fit into the FPGA's BRAM and must be stored in the onboard DDR memory. However, accessing DDR memory is slow. To improve performance, a 4-way associative flow cache with 16K cache lines is created in the BRAM. The least recently used (LRU) algorithm is used to replace entries in the flow cache.

As depicted in Figure 4.22, the incoming packet initially traverses a *parser* component, which extracts the 5-tuple and transmits them to the *flow cache* component. If the flow is not located in the flow cache, the packet's metadata is forwarded to the global flow table, which reads the complete table in the DDR. If there is still no matching entry, then this packet is the first packet of the flow, and the request is dispatched to the *DIPAlloc* component to allocate a DIP for this flow according to the load balancing policy. After the DIP is determined, an entry is inserted into the flow table.

Upon determining the DIP of the packet, the encapsulation component will retrieve

the next hop information, such as the IP address and VNET ID, and generate the packet of the NVGRE encapsulation accordingly. For the remaining packets of the flow, the DIP is extracted from the flow state. If a FIN packet is received or a timeout occurs, the flow entry will be invalidated before receiving any new packets from the flow. After determining the DIP, the next hop metadata is retrieved from the BRAM and the NVGRE header is encapsulated to guide the packet to the allocated DIP.

Except for the *DIPAlloc* component, all components are placed in the FPGA. Since only the first packet of the flow may encounter *DIPAlloc* and the allocation policy may also be complex, it is more suitable to run the *DIPAlloc* on the CPU, which is another example of joint CPU-FPGA processing.

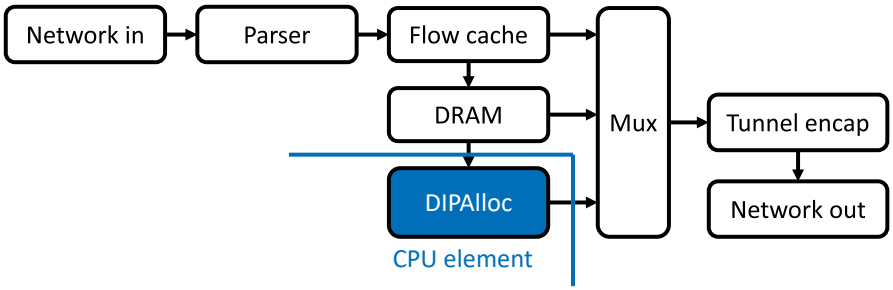


Figure 4.22 Component architecture of the L4 load balancer.

The following compares L4LB with Linux Virtual Server (LVS)[201]. To stress test the system, a large number of concurrent UDP streams are generated using 64B packets, targeting a Virtual IP (VIP). Figure 4.23 (a) shows the processing rate with different numbers of concurrent streams. When the concurrent traffic is less than 8K, L4LB reaches a line rate of 51.2Mpps. However, as the number of concurrent streams increases, the processing rate begins to decline. This is due to cache misses in the flow cache of L4LB. When a flow is missing from the flow cache, L4LB must access the onboard DDR memory, which leads to a performance drop. When the traffic is too high, for example, 32M, cache misses dominate and for most packets, L4LB needs to access DDR memory once. Therefore the processing speed drops to 11Mpps. In any case, the processing rate of LVS is very low. Since LVS associates the VIP with only one CPU core, its processing rate must reach 200Kpps.

The translation of your LaTeX content into English while preserving the original LaTeX markup structure is as follows:

Figure 4.23 (b) illustrates the latency under varying load conditions. In this experiment, the number of concurrent streams is held constant at one million. It is evident that L4LB achieves an impressively low latency of 4 μs. However, LVS incurs a delay of approximately 50 μs. When the throughput load exceeds 100Kpps, the queuing delay

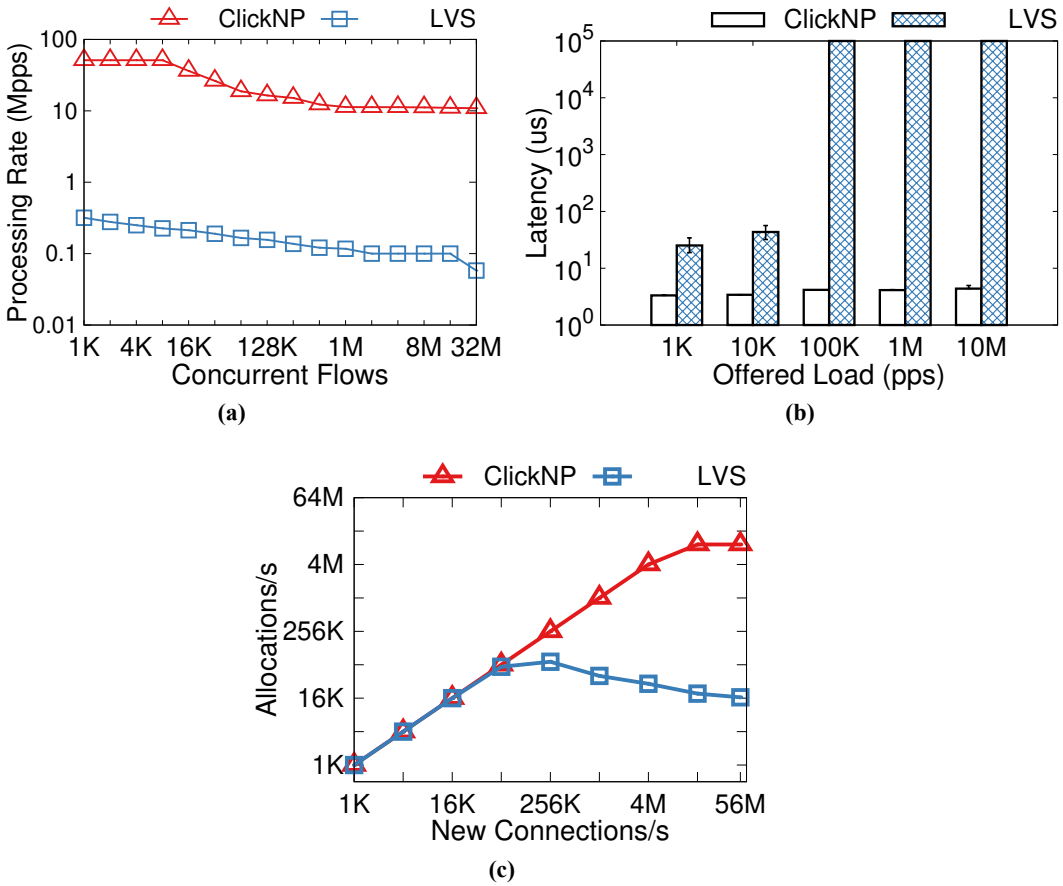


Figure 4.23 Performance evaluation of L4 load balancer.

escalates rapidly, surpassing the processing capacity of LVS.

Lastly, Figure 4.23 (c) contrasts the capacity of L4LB and LVS to accept new flows. This experiment directs PktGen to generate as many single-packet micro-flows as possible. It is observable that L4LB can accommodate up to 10M new connections per second. Given that a single PCIe slot can transmit 16.5M data per second, the bottleneck remains DDR access. For simplicity, the *DIPAlloc* component in this paper allocates DIP in a round-robin fashion. For complex allocation algorithms, the CPU core of *DIPAlloc* will become the bottleneck, and performance can be enhanced by replicating the *DIPAlloc* component on more CPU cores. For LVS, due to its limited packet processing capacity, it can accept a maximum of 75K new connections per second.

4.6.4 pFabric Flow Scheduler

ClickNP is also a valuable tool for network research. Owing to its flexibility and high performance, ClickNP can swiftly create prototypes of the latest research and apply them to real environments.

This section employs ClickNP to implement a recently proposed packet scheduling

rule–pFabric[171]. pFabric scheduling is straightforward. It maintains a shallow buffer (32 packets) and always dequeues the packet with the highest priority. When the buffer is full, the packet with the lowest priority will be discarded. pFabric has achieved near-optimal flow completion time in data centers. In the original paper, the authors proposed using a Binary Comparison Tree (BCT) to select the packet with the highest priority. However, although BCT only requires $O(log_2 N)$ cycles to calculate the packet with the highest priority, there is a dependency between two consecutive selection processes. This is because only after the previous selection is completed can the packet with the highest priority be known, and then the next selection process can be reliably started. This restriction requires a clock frequency of at least 300MHz to achieve a line speed of 40Gbps, which is currently unattainable for the existing FPGA platform.

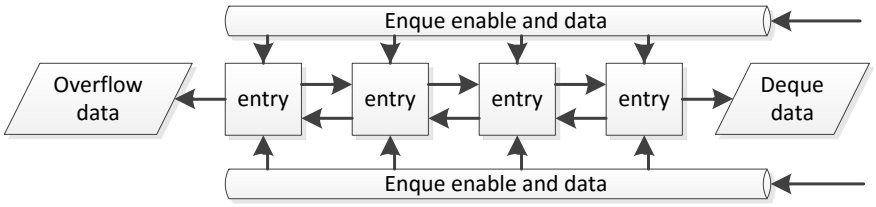


Figure 4.24 Shift register priority queue.

This paper employs a distinct method to implement the pFabric scheduler, which is more amenable to parallelization. The approach is grounded on the *shift register priority queue*[202]. As depicted in Figure 4.24, entries are stored in K registers in a non-increasing priority order. During dequeuing, all entries shift to the right and are popped from the head. This process only necessitates 1 cycle. For the enqueue operation, the metadata of the new packet will be forwarded to all entries. At this point, for each entry, a local comparison can be conducted between the packet in the entry, the new packet, and the packet in the adjacent entry. Given that all local comparisons can be executed in parallel, the enqueue operation can also be accomplished in 1 cycle. Enqueue and dequeue operations can be further parallelized. Hence, a packet can be processed in a single cycle. Figure 4.25 illustrates the component architecture of the pFabric flow scheduler.

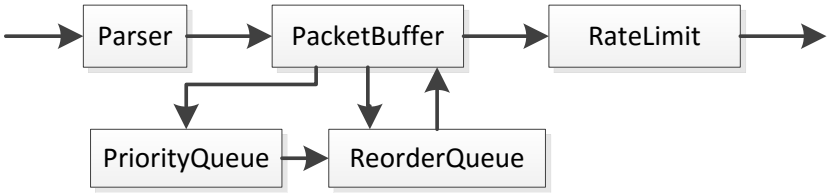


Figure 4.25 Component architecture of pFabric flow scheduler.

In this experiment, a software TCP flow generator[203] was modified to embed the flow priority, that is, the total size of the flow, into the packet payload. The experiment

generated flows in accordance with the data mining workload in[171] and utilized the *RateLimit* element to further set the exit port limit to 10 Gbps. The pFabric application schedules the traffic in the exit buffer based on the flow priority. Figure 4.26 presents the average flow completion time (FCT) and ideal value of pFabric and TCP with a Droptail queue. This experiment validates that pFabric achieves near-optimal FCT in this straightforward scenario.

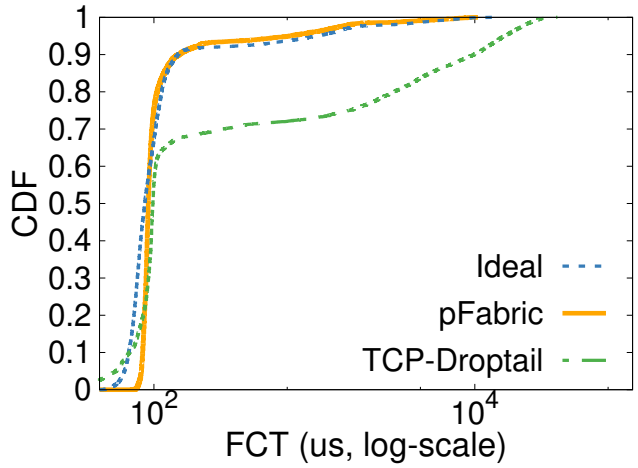


Figure 4.26 Verification of pFabric.

4.6.5 Fault-tolerant EPC SPGW

The data plane of the LTE core network (EPC) employs S-GW and P-GW to process network packets, with its processing flow resembling that of the IPSec gateway, as depicted in Figure 4.27. The EPC SPGW must maintain the state for each bearer (i.e., user), with the bearer's state being modified each time a packet passes through. The EPC SPGW demands not only high throughput and low latency, but also high fault tolerance, meaning that hardware failures should be imperceptible to users and the state should not be lost.

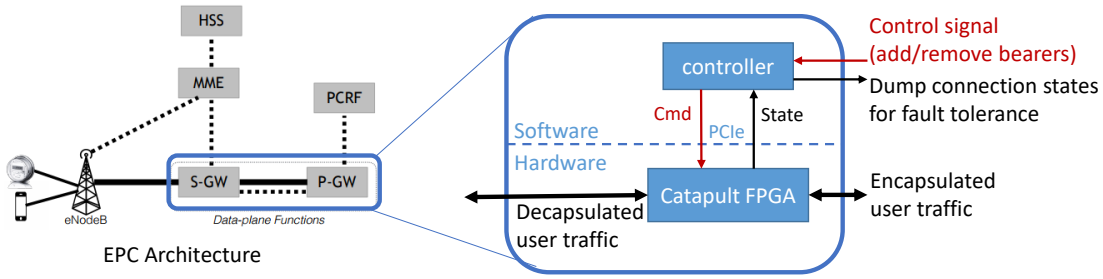


Figure 4.27 Acceleration architecture of LTE core network (EPC).

High fault tolerance is achieved using the state machine replication method outlined in Section 4.5.5, as illustrated in the component structure in Figure 4.28. Each user requires approximately 300 bytes of state, the FPGA's on-chip memory can cache the state of around 4K users, and the on-board BRAM can store the state of 10M users, thus

a single FPGA can support up to 10M concurrent connections. In the typical scenario where user access follows a power law distribution, the FPGA's throughput can reach 40 M packets per second (pps); in the worst-case scenario, due to cache misses, the throughput can reach 20 M pps. Under normal circumstances, the FPGA's 95% latency is 4 μs. The control plane is implemented through the PCIe I/O pipeline, with a 95% control plane latency of 1 μs and a throughput of 1M add or delete user operations per second. When adding a new backup node, it is necessary to back up the user state of the original node to the new node. When the network is idle, this state migration process can fully utilize the 40 Gbps bandwidth, and state replication can be achieved in just 0.8 seconds.

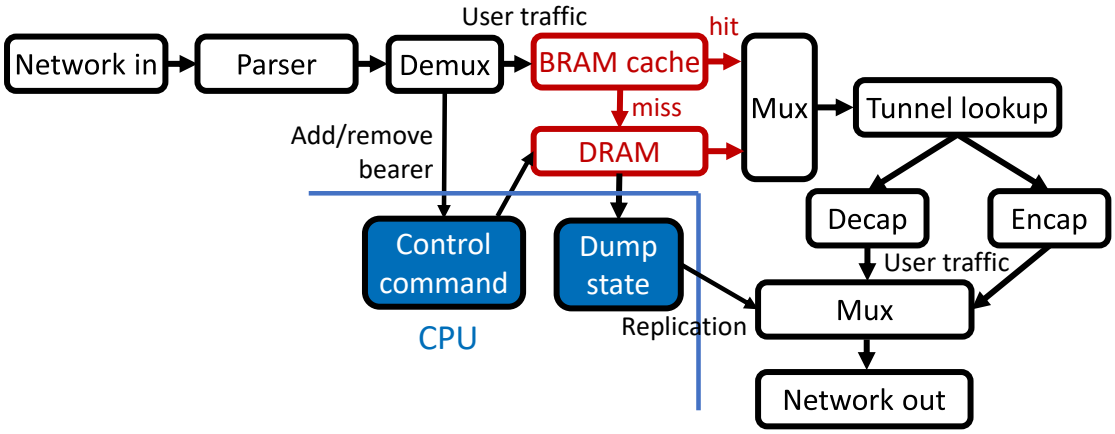


Figure 4.28 Component structure of LTE SPGW.

4.7 Discussion: Resource Utilization

This section will evaluate the resource utilization of the ClickNP network function. The results are summarized in Table 4.3. Except for the IPSec gateway that utilizes most of the BRAM to store the code book, all other network functions only use a moderate amount of resources (5 to 50

Table 4.3 Summary of ClickNP network functions.

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%

† The sum of the number of lines of code for all component description languages and configuration files.

Next, we examine the overhead of ClickNP 's fine-grained modularization. Since each component will generate logic block boundaries and only use FIFO buffers to com-

municate with other blocks, there should be overhead. To measure this overhead, create a simple ”*empty*” component that only passes data from one input port to the output port. The resource utilization of this *empty* component can well reflect the overhead of modularization. Different high-level synthesis tools may use different amounts of resources, but they are all very low, with a minimum of 0.15

Finally, we examine the efficiency of the hardware description language code generated by ClickNP in comparison to manually written hardware description language. For this, we use NetFPGA[138] as a reference. Initially, we extract the key modules in NetFPGA, which have been optimized by experienced Verilog programmers, and implement corresponding components with the same functionality in ClickNP . Subsequently, using different high-level synthesis tools as the backend, we compare the relative area cost between these two implementations. The results are summarized in Table 4.4. As different tools may have varying area costs, we record the maximum and minimum values. It is evident that the automatically generated hardware description language code uses more area compared to manually optimized code. However, the difference is not substantial. For complex modules (as shown at the top of the table), the relative area cost is less than twice. For smaller modules (as shown at the bottom of the table), the relative area cost appears larger, but the absolute resource usage is minimal. This is because existing high-level synthesis tools generate fixed control logic for each component, resulting in area overhead.

Table 4.4 Area overhead compared to NetFPGA.

NetFPGA Function	Logic Lookup Table (LUT) Min / Max	Register Min / Max	Memory (BRAM) Min / Max
Input selector	2.1x / 3.4x	1.8x / 2.8x	0.9x / 1.3x
Output queue	1.4x / 2.0x	2.0x / 3.2x	0.9x / 1.2x
Packet header parser	0.9x / 3.2x	2.1x / 3.2x	N/A
Openflow lookup table	0.9x / 1.6x	1.6x / 2.3x	1.1x / 1.2x
IP checksum calculation	4.3x / 12.1x	9.7x / 32.5x	N/A
Tunnel encapsulation	0.9x / 5.2x	1.1x / 10.3x	N/A

In conclusion, ClickNP can generate efficient hardware description language for FPGA with only a moderate area cost, and can construct practical network functions. Looking ahead, FPGA technology is still rapidly evolving. For instance, the area of Intel’s Arria 10 FPGA and the latest Stratix 10 FPGA are 2.5 times and 10 times that of the chip used in this study (Stratix V), respectively. Therefore, the area cost of high-level synthesis will become less significant in the future.

4.8 Extension: Computation-Intensive Applications

Some network functions are computation-intensive, such as the IPSec gateway in Section 4.6.2 which needs to encrypt and sign the content of each data packet. Due to the large amount of computation for a data packet, the area of fully unfolding the entire computation flow graph into a pipeline will exceed the capacity of the FPGA. Therefore, it is necessary to *exchange time for space*, and split the computation flow graph. Due to the dependencies in the computation process, Section 4.6.2 shows how to use *reservation stations* to divide the computation of data packets into multiple stages, save intermediate states, and fully utilize the parallelism between different connections. This section discusses two applications with larger computations: HTTPS RSA acceleration and neural network inference, to demonstrate the general method of implementing computation-intensive applications.

4.8.1 HTTPS RSA Acceleration

HTTPS is a protocol for secure connections with web services. With increasing user concern for privacy, more and more web services provide access through HTTPS. Since 2010, the proportion of HTTPS traffic has grown by 40

HTTPS provides three mechanisms to ensure security. First, when a connection is established, it uses to verify the identity of the web server and create a shared key for both parties. Second, it encrypts the data transmission between the user and the web server. Third, it checks the integrity of the data. Among these three mechanisms, connection establishment is the most computation-intensive part because it requires asymmetric key operations (such as the RSA algorithm).

Without enabling HTTPS, a single CPU core can process over 7,000 HTTP requests per second. When using HTTPS, the throughput drops to 1/35 due to the TLS handshake in the connection setup. The high computational overhead of the TLS handshake has always been a major obstacle for high-traffic websites to deploy HTTPS to ensure security.

In the TLS handshake, the web server carries out decryption using the RSA private key. As depicted in Figure 4.29, RSA decryption is mathematically a large integer power modulus operation, where the base, exponent, and modulus are all large integers. By binary decomposition of the exponent, the large integer power modulus operation can be implemented by iteratively performing multiple multiplication modulus operations. The multiplication modulus operation can be transformed into 3 multiplication, 1 addition, and 1 subtraction operations by the Montgomery algorithm. For a 2048-bit private key,

decryption requires approximately 8 million 16-bit integer multiplication operations. Although TLS certificate parsing and protocol processing are also complex, the number of computations required is significantly less than decryption. Therefore, we modify the OpenSSL library to offload RSA decryption operations to the FPGA and retain the other parts on the CPU.

Clearly, fully unfolding the entire RSA decryption process into digital logic would occupy too much chip area, so it is necessary to trade time for space and construct a smaller scale multiplication and addition array, repeating the computation multiple times. For a 2048-bit key, even the innermost 1024-bit large integer multiplier has already exceeded the FPGA's DSP quantity limit, so it is necessary to further divide the large integer multiplication. In the Montgomery algorithm, the 3 multiplications can obviously reuse the same multiplier array.① The ModMulti controller in Figure 4.29 moves data between the multiplier, adder, and subtractor; the Binary Exp controller moves data between multiple ModMulti operations. In addition, since there is pipeline delay in the operation components, and there is dependency between the computations of a single RSA decryption, many operation components will be idle if only one RSA decryption is performed. In order to fully utilize the parallelism of the FPGA, multiple RSA decryption tasks need to be processed concurrently. The ModMulti and Binary Exp controllers need to manage the intermediate data and execution status of concurrent tasks, and schedule unrelated subtasks to the computation components.

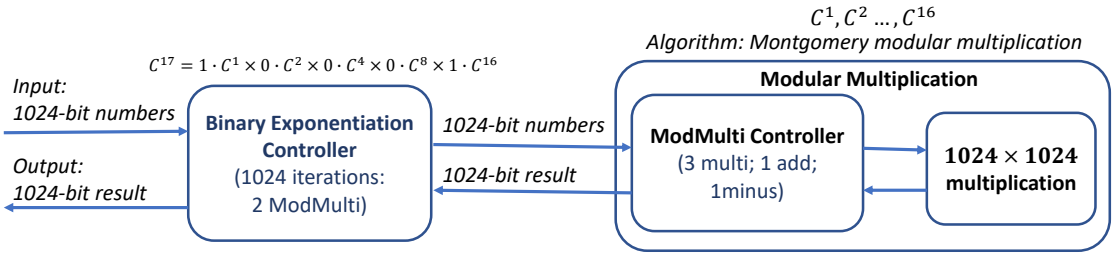


Figure 4.29 Component structure of the HTTPS accelerator (simplified diagram).

Clearly, manually implementing the logic of controller cache management, task division, concurrency control, data movement, etc. is quite laborious. For this reason, we aim to automatically generate controller code from the source code shown in Figure 4.30.

Firstly, ClickNP begins from the innermost loop, attempting to unroll as many loop layers as possible to parallelize the maximum number of computations. In instances of

①If three separate multiplier arrays are used, then under the condition of a fixed total area, the parallelism of each multiplier array will become 1/3, and since there is data dependency between the 3 multiplications, the delay of the RSA decryption operation will increase to about 3 times. Therefore, when the parallelism is adjustable, in most cases it is beneficial to combine multiple elements that cannot be executed in parallel into one element with greater parallelism.

```
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;
}
```

Figure 4.30 Schematic code of ClickNP for large integer power modulus operation used in 2048-bit RSA decryption. The ClickNP template is a syntactic sugar for generating recursive structure code, which will be expanded and eliminate invalid code at compile time. Functions not declared as inline are implemented as independent components, just like the async **primitive.**

multiple parallel loops (such as the multiplication and addition in Figure 4.30), the unroll count for each loop needs to correspond to the number of computations ①, ensuring that the throughput of each computation module is matched. Developers can specify the unroll count for each loop separately; for programs where the loop iteration count is statically known at compile time, a global unroll count can also be specified as the unroll count for the loop with the highest computational intensity, and ClickNP will automatically match the unroll count for other loops.② The upper limit of the loop unroll

①For instance, if two loops have 512 and 1024 iterations respectively, the ratio of the unroll count for the two loops is 1:2.

②For programs with uncertain loop times, developers can manually divide them into several static areas and specify the required throughput or area target for each area.

count is dependent on the FPGA's hardware capacity, and developers can configure it based on the resource estimates reported by the high-level synthesis tool or the results after synthesis and layout routing. Some programs have multi-layer loops whose order can be interchanged, and the data locality of the code generated by splitting different loop levels varies, resulting in different amounts of required data movement. When writing ClickNP code, developers should place the loop with the highest parallelism and strongest data locality in the innermost layer, so that when the compiler unrolls the inner loop, it can minimize the data movement overhead.

Next, ClickNP generates controller code. To generate code for concurrent execution with hidden latency, ClickNP compiles the computation components (i.e., unrolled loops) and the code for data movement between on-chip memory and computation components using high-level synthesis tools, to calculate their latency and throughput. The number of tasks executed concurrently is the product of latency and throughput. The controller stores the current execution status and intermediate data of each task in the on-chip memory and schedules the split sub-computation tasks to the computation components.

A typical inference neural network comprises several sparse and dense layers. Sparse layers primarily read randomly from larger feature arrays, while dense layers mainly perform matrix or vector multiplication. Each sparse and dense layer is represented by a ClickNP component, and the computation flow graph of the neural network is the connection between ClickNP components. If each component is implemented as separate hardware logic, the on-chip resources will be fragmented. For a specific task, only a few computation components are running at the same time, resulting in longer latency for a single task. Therefore, it is necessary to extract common computation components from different components. For network functions, ClickNP has difficulty optimizing this because the types of computations performed by various components of network functions are different and it is difficult to extract common parts.

The method to extract common computation components is to compare the isomorphism of loops in different components, that is, whether the abstract syntax trees of two loops can be made completely the same by replacing the accessed array names and loop variable names. If the loops in several different components are isomorphic, these refactored loops can be extracted into a new `async` component. For example, access to global memory in each sparse layer can be extracted into one component, matrix and vector multiplication in each dense layer can be extracted into one component, and the relu activation function in each dense layer can be extracted into another component.

Next, similar to RSA decryption, ClickNP unrolls the inner loops as much as possible and generates controller code. As shown in Figure 4.5, the higher the parallelism of loop unrolling, the lower the latency of neural network inference.

Table 4.5 Latency of neural network inference at different parallelism levels of dense layers. The neural network used consists of three sparse layers and four dense layers. The features obtained by the three parallel-executing sparse layers are concatenated with other input features and then enter the dense layers. The four dense layers form a pipeline, each consisting of matrix and vector multiplication, activation function, and normalization function.

Dense layer parallelism	Per-sample latency (μs)
8	188.2
16	98.2
32	49.3
64	26.1
128	17.5
256	Insufficient resources

Finally, it should be noted that for compute-intensive tasks, although the high-level synthesis method adopted by ClickNP significantly improves the development efficiency compared to manually writing low-level FPGA code, its area overhead may be considerably higher than manually optimized FPGA code. Fortunately, the performance of network packet processing tasks is primarily limited by network bandwidth, and in most cases, the FPGA area is not a significant consideration.

4.9 Chapter Summary

This chapter introduced ClickNP , an FPGA acceleration platform for highly flexible and high-performance network functions in commercial servers. ClickNP is fully programmable in a high-level language and provides a modular architecture familiar to software programmers in the network domain. ClickNP supports joint CPU / FPGA packet processing and has high performance. Evaluation shows that compared with the most advanced software network functions, ClickNP increases the throughput of network functions by 10 times and reduces latency by 10 times. This chapter presented a specific case, showing that FPGA can accelerate network functions in data centers. This chapter confirmed that high-level language programming on FPGA is actually feasible and practical.

Chapter 5 Acceleration of KV-Direct Data Structures

5.1 Introduction

The theme of this chapter is storage virtualization and data structure processing acceleration, as shown in Figure 5.1. Among them, data structure processing acceleration is the focus of this article, and storage virtualization is only briefly discussed in Section 5.6.3.

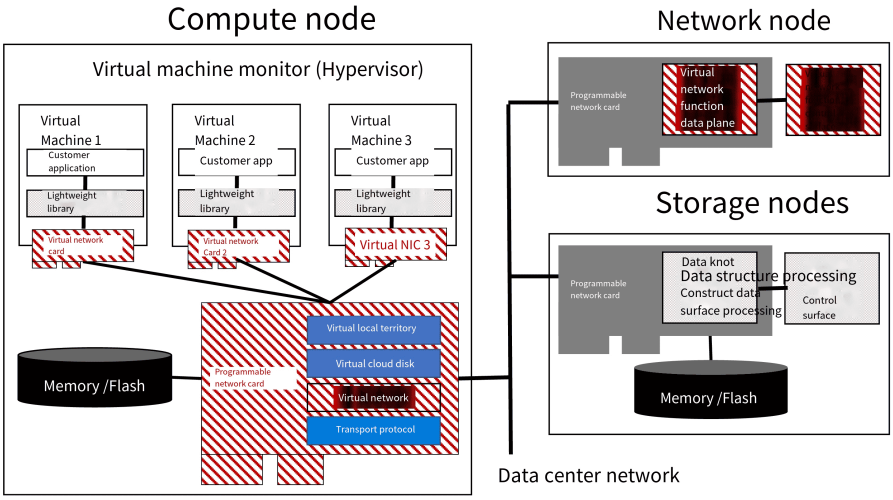


Figure 5.1 The theme of this chapter: storage virtualization and data structure processing acceleration, marked with a bold oblique line background.

In terms of programmable network card programming, this chapter builds on the ClickNP programming framework proposed in the previous chapter, establishes the foundation of the service layer, proposes a framework for stateful processing and data structure processing, and implements memory key-value storage based on this, as shown in Figure 5.2.

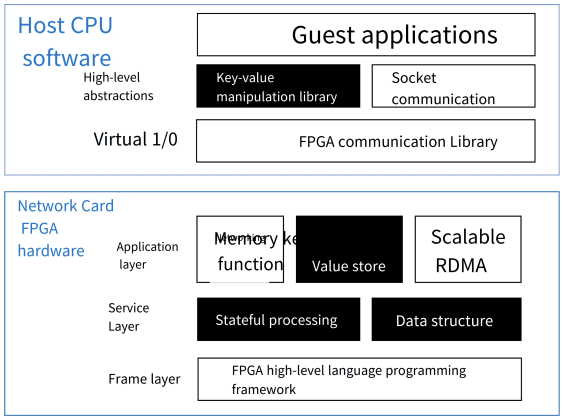


Figure 5.2 The position of this chapter in the programmable network card software and hardware architecture.

In-memory key-value storage is a key component of distributed systems in data

centers. This chapter presents KV-Direct, an in-memory key-value system based on programmable network cards. As the name suggests, the programmable network card of KV-Direct receives and processes key-value operation requests from the network, and applies updates directly in the host memory, bypassing the host CPU. KV-Direct extends RDMA primitives from memory operations (read and write) to key-value operations (GET, PUT, DELETE, and atomic operations). Furthermore, to support vector-based operations and reduce network traffic, KV-Direct also provides new vector primitives UPDATE, REDUCE, and FILTER, allowing users to define active messages[204] and delegate certain computations to the programmable network card.

The design focus of key-value processing within the programmable network card is to optimize PCIe traffic between the network card and the host memory. KV-Direct adopts a series of optimizations to fully utilize PCIe bandwidth and hide latency. Firstly, KV-Direct designs a new hash table and memory allocator to take advantage of FPGA's parallelism and minimize the number of PCIe DMA requests. On average, KV-Direct uses only close to one PCIe DMA operation per GET operation and two PCIe DMA operations per PUT operation. Secondly, to ensure the consistency of key-value storage, KV-Direct designs an out-of-order execution engine to track operation dependencies while maximizing the throughput of independent requests. Thirdly, KV-Direct implements a hardware-based load dispatcher and cache components in FPGA to fully utilize the bandwidth and capacity of onboard DRAM.

Based on the above optimization, a single network card KV-Direct system can achieve up to 180 M key-value operations per second, equivalent to the throughput of 36 CPU cores[31]. Compared with the most advanced CPU key-value storage system, KV-Direct can reduce tail latency to 10 μs, while improving energy efficiency by 3 times. Moreover, KV-Direct can achieve near-linear scalability through multiple network cards. By using 10 programmable network cards in a single commodity server, the performance can reach 1.22 billion key-value operations per second, which is an order of magnitude higher than existing systems.

KV-Direct also supports up to 180 Mops of general atomic operations, significantly better than the performance reported in the most advanced RDMA-based systems: 2.24 Mops[46]. The high performance of atomic operations is mainly attributed to the out-of-order execution engine. The out-of-order execution engine can efficiently track dependencies between key-value operations without blocking the pipeline.

The rest of this chapter is arranged as follows. Section 5.2 introduces the background, and clarifies the design goals and challenges. Section 5.3 describes the design

of KV-Direct. Section 5.5 evaluates the performance of KV-Direct. Sections 5.6 and 5.7 discuss the extensions of this paper. Section 5.8 discusses related work. Section 5.9 concludes.

5.2 Background

5.2.1 The Road to High Performance Key-Value Storage

Building a high performance key-value storage is a non-trivial exercise of optimizing various software and hardware components in a computer system. The rich literature on key-value storage performance optimizations can give us a glimpse of software and hardware evolution in recent years. Early works on distributed in-memory key-value storage such as Memcached[205] uses OS locks for multi-core synchronization and TCP/IP networking stack for communication. Since then, optimizations have been made on multiple fronts to remove bottlenecks in various parts of the system.

1. Synchronization cost

Synchronization is needed in multi-threaded key-value storage implementation since multiple clients might access the same keys concurrently. For example, when two clients make atomic increments to a single key, the value needs to reflect both increments.

To reduce synchronization cost, Masstree[32], MemC3[33] and libcuckoo[34] optimize caching, hashing and memory allocation algorithms, and replace permissive kernel locks with optimistic version-based locks. MICA[30-31] takes a further step to completely avoid synchronization by partitioning the hash table to each core so that each core serves an exclusive portion of the hash table. This approach, however, may introduce core imbalance for long-tail access patterns with a few extremely popular keys[31].

2. Networking overhead

In a key-value storage where computation to communication ratio is low, a significant portion of CPU cycles is spent in the kernel networking stack, including protocol handling, memory copy, system call and multi-core synchronization[12]. Furthermore, the kernel network stack adds hundreds of microseconds latency[29], which greatly impacts response time. and complicates latency-hiding programming of applications that require multiple round-trips to the key-value storage.

Extensive research has been conducted to reduce network communication costs and improve end-to-end latency. One line of work proposes that key-value storage server software communicates directly with network cards by polling while bypass-

ing the kernel[13,206]; packets are processed by a lightweight network stack in user space[16-17]. Chronos[27], RAMCloud[28-29] and MICA[30-31] leverage this approach to achieve high throughput (approximately 5 million key-value operations per second (op/s) per core) and low latency (microsecond-scale) by reducing networking overhead.

The other line of work leverages two-sided RDMA[35] as an RPC mechanism between key-value storage client and server. RDMA is a hardware-based transport that almost completely removes the CPU overhead of networking. Key-value storage systems such as HERD[46-47] achieve per-core throughput and end-to-end latency comparable or superior to the first line of work, but the overall throughput per server largely depends on the processing capacity of RDMA network cards[47].

3. Throughput bottleneck of CPU

When pushed to the limit, in high performance key-value storage systems the throughput bottleneck can be attributed to the computation in key-value operation and the latency in random memory access. Key-value storage needs to spend CPU cycles for key comparison and hash slot computation. Moreover, key-value storage hash table is orders of magnitude larger than the CPU cache, therefore the memory access latency is dominated by cache miss latency for practical access patterns.

By our measurement, a 64-byte random read latency for a contemporary computer (Sec. **??**) is approximately 110 ns. A CPU core can issue several memory access instructions concurrently when they fall in the instruction window, limited by the number of load-store units in a core (measured to be 3 to 4 in our CPU)[207-209]. In our CPU, we measure a max throughput of 29.3M random 64B access per second per core. On the other hand, an operation to access 64-byte key-value pair typically requires approximately 100ns computation or approximately 500 instructions, which is too large to fit in the instruction window (measured to be 100 to 200). When interleaved with computation, the performance a CPU core degrades to only 5.5 MOps. An optimization is to batch memory accesses in a key-value store by clustering the computation for several operations together before issuing the memory access all at once[31,210]. This improves the per-core throughput to 7.9 MOps in our CPU, which is still far less than the throughput of the host DRAM.

One can batch memory accesses in a key-value store, i.e., clustering the computation for several operations together before issuing the memory access all at once[31,210]. Table 5.1 depicts the measured per-core hash table throughput under different key-value sizes and batch sizes, assuming the key-value is inlined in hash table and each key-value operation requires one non-cached memory access. The results fit the following formu-

las within 10% error:

$$\frac{1}{RandAccessThroughput} = \frac{MemLatency}{Parallelism} \tag{5.1}$$

$$\frac{1}{KeyValueOpThroughput} = ComputationTime + \frac{MemLatency}{min(BatchSize, Parallelism)} \tag{5.2}$$

This indicates that when interleaved with computation, the performance of CPU degrades significantly. In the extreme case, even if the instruction window size or memory fetch parallelism goes to infinity, the per-core key-value operation throughput would still be bounded by computation (~10M op/s), 10~20x slower than a single DDR channel.

5.2.2 Domain-Specific Architectures for Key-Value Storage

Ten years ago, processor frequency scaling was over and people turned to multi-core and concurrency[211]. Nowadays, CMOS feature-size reduction is getting more and more difficult, which implies that multi-core scaling is also over. People are turning to domain-specific architectures (DSAs)[212] for better performance. Several such DSAs have been used to improve key-value storage performances.

For computation, DSAs such as GPU, FPGA[213?] and ASIC[214-215] have been quickly accepted by the market. For networking, DSAs are also deployed at scale in datacenters, such as RDMA/RoCE network cards[216], programmable network cards[156,217] and programmable switches[107].

1. One-sided RDMA

Due to high overhead in CPU network processing, DSAs to accelerate networking, such as RDMA/RoCE network cards[216], are deployed at scale in datacenters. High performance key-value storage systems can leverage RDMA capable hardware. One approach is to accelerate RPC with *two-sided* verbs in RDMA/RoCE network cards (section 2, Figure 5.3a). By doing so, the key-value performance is bounded by CPU (section **??**).

A significantly different approach is to leverage *one-sided* RDMA to access remote memory via the network card on the client and bypass the CPU on the server, as shown in Figure 5.3b. In this approach, key-value computation and synchronization are handled by the client CPU, therefore making the key-value server very light-weight and high performance. Despite the high message rate (8M~150M op/s[47]) provided by RDMA network cards, it is challenging to find an efficient match between RDMA primitives

and key-value operations. For a write (PUT or atomic) operation, multiple network round-trips and memory accesses may be required to handle hash conflicts, memory allocation and fetch/save non-inline data. RDMA does not support transactions. Clients must synchronize with each other to ensure consistency by either using RDMA atomics or distributed atomic broadcast[218], both incurring communication overhead and synchronization latency[70,219]. As a consequence, most RDMA-based key-value storage, *e.g.*, Pilaf[219], FaRM[70] and HERD[46] recommend using one-sided RDMA for GET operations only. For PUT operations, they fall back to two-sided RDMA as RPC and let the remote CPU do the actual work. Throughput of write-intensive workload is still bottlenecked by CPU cores.

In addition to the mismatch between RDMA primitives and key-value operations, implementation of commodity RDMA network cards also constrain key-value throughput. For example, RDMA network cards hold a lock for atomic operations when a PCIe DMA to the same memory address is in flight, which bounds RDMA atomics throughput to ~2M op/s[47].

2. Highly parallel architectures

Highly parallel architectures such as many-core processors[220], GPGPU[209].

3. FPGA

FPGA[221-224,224? ? ? -226] have been explored to overcome the limited parallelism of CPU in accessing DRAM. Compared to general-purpose processors, FPGA has more flexible pipeline parallelism and can be specialized for the key-value store application. Compared to RDMA, FPGA can support key-value operation primitives directly, as well as specializing network packet format and PCIe DMA operations to use network and PCIe bandwidth efficiently. Compared to GPGPU, FPGA is more power-efficient and has lower latency.

In recent years, FPGA is becoming cost-effective and is getting deployed at scale in datacenters[213?]. Its programmability has been greatly improved[156]. Most existing work store the entire hash table inside the on-board DRAM, which is often quite limiting (typically on the order of 4~16 GiB), while the host DRAM is often large (on the order of 100~500 GiB). KV-Direct follows this line of work, while leveraging host DRAM for key-value storage, as depicted in Figure 5.3c.

KV-Direct leverages a programmable network card with large-scale deployments in datacenters. The programmable network card is composed of two parts: a traditional RDMA network card plus a field-programmable gate array (FPGA). There has been research on leveraging the reconfigurability of the network card for network processing,

e.g., network virtualization[217,227] and network functions[156]. KV-Direct extends the application of programmable network cards to a novel area: key-value stores.

5.2.3 Concept of Key-Value Storage

As the name suggests, key-value storage stores an unordered collection of key-value pairs. Both the key and value are variable-length arbitrary strings. In a key-value store, the same key can only appear once. The basic operations of key-value storage are GET and PUT. The GET operation inputs a key and outputs the corresponding value. The PUT operation inputs a key-value pair and saves it to the key-value store. If there is the same key, the original key-value pair is deleted and the new key-value pair is saved. To efficiently support read and write operations, key-value storage is usually based on a hash table. Whether it is a GET or PUT operation, it first calculates the hash value of the key and looks it up in the hash table. For the PUT operation, it may need to allocate memory for the new key-value pair and release the memory occupied by the original key-value pair of the same key. In distributed systems, key-value storage is often used as a service, receiving GET and PUT operations from network clients, and sending the processing results back to the clients through the network.

5.2.4 Workload Shift in Key-Value Storage

Historically, key-value stores like Memcached[205] gained popularity as object caching systems for web services. In the era of in-memory computing, key-value stores have transcended caching to become infrastructure services for storing shared data structures in distributed systems. Many data structures can be represented in key-value hash tables, such as data indices in NoSQL databases[65], model parameters in machine learning[81], nodes and edges in graph computing[98,228], and sequence number generators in distributed synchronization[47,229]. In the future, in-memory key-value stores can also provide high-performance temporary storage for serverless computing[100].

The shift in workload from object caching to general data structure storage implies several new design goals for key-value stores.

High throughput for small key-values. In-memory computing often involves large batch access to small key-value pairs, such as sparse parameters in linear regression[34,98] or all neighbor nodes in graph traversal[228], so key-value stores can benefit from batch processing and pipelining operations.

Predictable low latency. For many data-parallel computing tasks, the latency of iterations is determined by the slowest operation[29]. Therefore, controlling the tail

latency of key-value stores is very important. CPU-based designs often need to balance latency and throughput by adjusting batch size[31]. Moreover, due to irregular scheduling by the operating system, unpredictable hardware interrupts, and cache misses, CPU processing time may fluctuate significantly under high load[156].

High efficiency under write-intensive workloads. For caching workloads, the number of reads in key-value stores is usually more than writes[230], but this is no longer the case for distributed computing workloads such as graph computing[69] and parameter servers[81]. For PageRank computation in graphs[69] or gradient descent in parameter servers[81], each iteration cycle reads and writes each node or parameter once. Key-value stores need to provide an equal number of GET (read) and PUT (write) operations. Sequencers[47] require atomic increment operations rather than read-only operations. These workloads require a hash table structure that can efficiently handle read and write operations simultaneously.

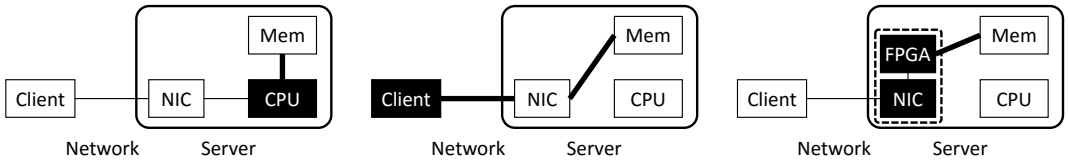
Fast atomic operations. Several very popular applications require atomic operations, such as centralized schedulers[231], sequencers[47,229], counters[232], and temporary values in web applications[230]. This requires high-throughput atomic operations on single keys.

Vector operations. Machine learning and graph computing workloads[81,98,228] often require operations on each element in a vector, such as adding a scalar to each element in a vector, or reducing a vector to the sum of its elements. Key-value stores without vector support require the client to issue a key-value store operation for each element in the vector, or to retrieve the entire vector as a large key-value pair, bring it back to the client, and perform the operation. If key-value stores support vector data types and operations, it can greatly reduce network communication and CPU computation overhead.

5.2.5 Performance Bottlenecks of Existing Key-Value Storage Systems

Building high-performance key-value stores requires global optimization of various software and hardware components in the computer system. Depending on where the data structure is processed, the state-of-the-art high-performance key-value storage systems can be basically divided into three categories: on the CPU of the key-value storage server (Figure 5.3a), on the key-value storage client (Figure 5.3b), or on the hardware accelerator (Figure 5.3c).

When network overhead is reduced to the limit, the throughput bottleneck of high-



(a) Software / Two-sided RDMA. **(b)** One-sided RDMA. **(c)** KV-Direct.

Figure 5.3 Design space of key-value storage data paths and processing devices. Rows represent data paths. A key-value operation (thin line) may require multiple address-based memory accesses (thick line). The black box indicates where key-value processing occurs.

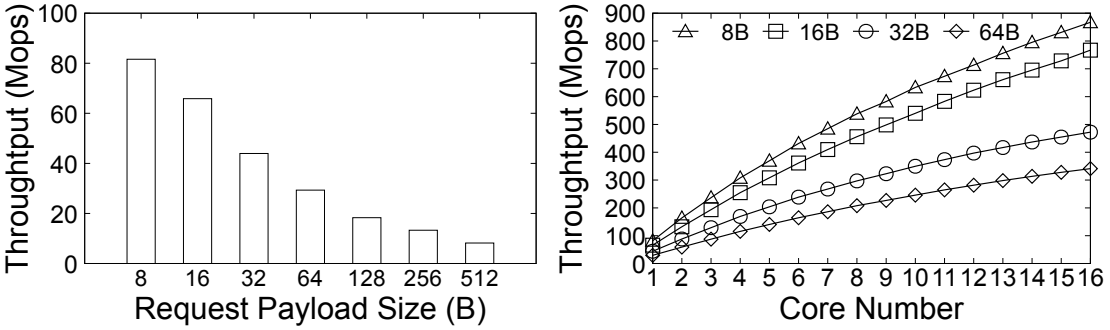
performance key-value storage systems can be attributed to the latency in key-value operations and random memory access. CPU-based key-value storage requires CPU cycles for key comparison and hash slot calculation. In addition, the key-value storage hash table is several orders of magnitude larger than the CPU cache, so the memory access latency is mainly determined by the cache miss latency under the actual access pattern.

Measurements show that the 64-byte random read latency of modern computers is about 100 ns ①. The CPU core can issue multiple memory access instructions at the same time, limited by the number of load-store units in the core (such as 3 to 4)[207-209] ②. As shown in Figure 5.4 and Table 5.1, in the CPU used in this paper's experiment, each core has a maximum throughput of 29.3 M random 64B accesses per second. On the other hand, the operation of accessing a 64-byte key-value pair usually requires about 100 ns of computation or about 500 instructions, which cannot fit into the instruction window ③. When random memory access and computation are interleaved, due to the instruction window not being able to cover the memory access latency, the performance of the CPU core is reduced to 5.5 M key-value operations (Mops) per second. One optimization method is to aggregate the computation of multiple key-value storage operations before issuing a memory access, and perform memory access in batches[31,210]. This optimization can increase the per-core key-value operation throughput of the CPU used in this paper to 7.9 MOps, which is still far lower than the random 64B throughput of the host DRAM.

①This random read latency assumes a normal page size of 4 KiB, taking into account the latency of TLB misses and data cache line misses.

②Although there may be dozens of load-store units in each core of the CPU microarchitecture, 64-byte random memory access will cause multiple TLB misses and data cache line misses, so in the actual measurements of this paper, only 3 to 4 random memory access operations can be completed within one memory access latency.

③The instruction window is the maximum number of instructions that the CPU out-of-order execution engine can rearrange. If there are more than the number of instructions in the instruction window after a memory access instruction, and the memory access latency is greater than the time to execute the number of instructions in the instruction window, then due to the limitation of the instruction window, the pipeline needs to pause (stall) after executing the number of instructions in the instruction window, waiting for the memory access result to return, before it can continue to execute subsequent computation instructions. On the CPU we used, the measured size of the instruction window is about 100 to 200.



(a) Single core. **(b)** Multi-core.

Figure 5.4 CPU random memory access performance.

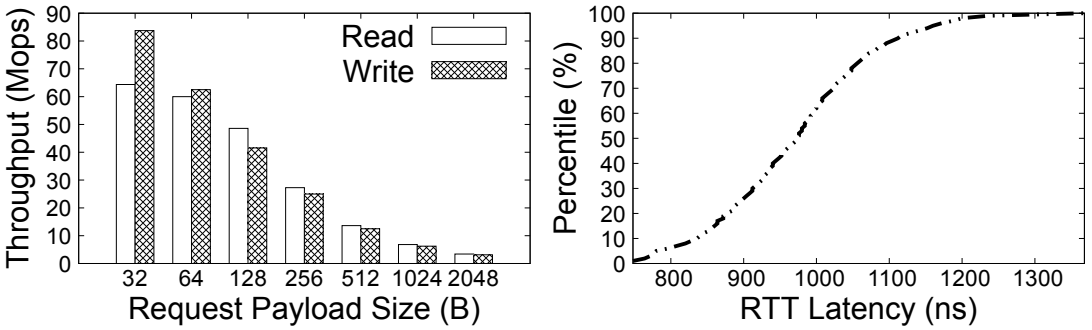
Table 5.1 Throughput (millions of operations per second) under different workloads and memory access granularities.

Size (bytes)	Computation only	Memory access only	Computation and memory access (batch size)			
			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

Observing the limited capacity of the CPU in key-value processing, recent work has used one-sided RDMA to offload key-value processing to the client. One-sided RDMA provides an abstraction of remote access to shared memory. The server-side application registers a block of memory with the local RDMA network card for shared memory. When the client application needs to read and write this shared memory, it sends an RDMA read or write work request to the local RDMA network card. The client RDMA network card converts the work request into a network packet and sends it to the server RDMA network card. The server RDMA network card converts the received packet into a PCIe DMA request, accesses the shared memory, and returns the result to the client RDMA network card. The client RDMA network card sends the read data to the application's memory buffer through PCIe DMA, and then notifies the application through work completion. In this process, the server-side RDMA network card handles read and write requests, completely bypassing the server-side CPU ①.

Although RDMA network cards provide high message throughput (8 to 150 Mops[47]), finding an efficient match between RDMA primitives and key-value operations is a challenge. For write (PUT or atomic) operations, multiple network

① In the modern data center server architecture, this statement is not rigorous, because the PCIe root complex and memory controller are both inside the host CPU, and the network card accesses the host memory through PCIe DMA must go through the CPU. This paper's "bypassing the CPU" follows the customary usage in the system academic community, which is a logical meaning, that is, bypassing the software processing on the CPU core. In the system architecture diagram of this paper, the CPU also refers to software processing. The term "bypassing the CPU" will appear many times in the following text, all with this meaning.



(a) DMA throughput. **(b)** DMA read latency.

Figure 5.5 Random PCIe DMA performance.

round trips and multiple memory accesses may be required to query the hash index, handle hash conflicts, and allocate variable-sized memory. RDMA does not support transactions. To maintain the consistency of data structures, clients must synchronize with each other, using RDMA atomic operations or distributed atomic broadcasts[218]. Both of these schemes generate communication overhead and synchronization latency[70,219]. Therefore, most RDMA-based key-value stores[46,70,219] suggest using one-sided RDMA only for GET (read-only) operations. For write (PUT or atomic) operations, they fall back to using the server CPU for processing. Therefore, the throughput of write-intensive workloads is still limited by the bottleneck of the server CPU.

5.2.6 Challenges Faced by Remote Direct Key-Value Access

KV-Direct moves key-value processing from the CPU to the programmable network card in the server (Figure 5.3c). Like RDMA, the KV-Direct network card accesses host memory via PCIe. PCIe is a packet-switched network with a round-trip latency of about 500 ns and a theoretical bandwidth of 7.87 GB/s per Gen3 x8 endpoint. In terms of latency, because the FPGA hard core used in this paper has an additional processing delay of about 300 ns, the programmable network card reads host memory that has been cached by the CPU via PCIe DMA, with a latency of 800 ns. When randomly DMA reading uncached host memory, there is an additional average delay of 250 ns due to DRAM access, DRAM refresh, and PCIe response reordering in the PCIe DMA engine (Figure 5.5b). In terms of throughput, each DMA read or write operation requires a PCIe transport layer packet (TLP) with a 26-byte header and padding for 64-bit addressing. For a PCIe Gen3 x8 network card that accesses host memory at a granularity of 64 bytes, the theoretical throughput is therefore 5.6 GB/s or 87 Mops.

To saturate the throughput of the PCIe Gen3 x8 interface using 64-byte DMA

requests, considering a latency of 1050 ns, 92 concurrent DMA requests are needed. However, two practical factors further limit the concurrency of DMA requests. First, the number of requests being processed for each type of DMA is limited by PCIe credit-based flow control. The PCIe root complex in the server announces 88 PCIe transport layer packet (TLP) credits for DMA posted operations and 84 TLP credits for DMA non-posted operations. This means that the number of concurrent write operations cannot exceed 88, and the number of concurrent read operations cannot exceed 84. Second, DMA read operations require the allocation of unique PCIe tags to identify and reorder DMA responses. Although the PCIe protocol and many memory controllers support 256 PCIe tags, the DMA engine in the FPGA used in this paper only supports 64 PCIe tags, further limiting the number of concurrent DMA read requests to 64. This results in a PCIe DMA read throughput of only 60 Mops (million operations per second), as shown in Figure 5.5a. On the other hand, for a 40 Gbps network and 64-byte key-value pairs, if the client sends these key-value pairs in batches, the upper limit of network throughput is 78 Mops. This paper aims to saturate network throughput with GET (read) operations. Therefore, the key-value store on the network card must fully utilize the PCIe bandwidth, i.e., the average number of memory accesses per GET operation needs to be close to 1. This boils down to three challenges:

Minimize the number of DMA requests per key-value operation. The hash table and memory allocator are the two main components in the key-value store that require random memory access. Previous work suggests using Cuckoo hash, which keeps the number of memory accesses per GET operation close to 1 even under high load factors. However, Cuckoo hash is optimized for read operations. At load factors above 50

In addition to hash table lookups, dynamic memory allocation is needed to store variable-length key-values that cannot be inlined in the hash table. To match PCIe and network throughput under write-intensive small key-value workloads (i.e., to nearly fully utilize both), the number of memory accesses for hash table lookups and memory allocation should be minimized.

Hide PCIe latency while maintaining consistency. Consistency is a term in distributed transactions that refers to the property of logical isolation between concurrently executing transactions. Different hardware modules inside a programmable network card process in parallel, forming a distributed system. Since a key-value operation requires multiple memory read-write accesses, when multiple key-value operations are processed concurrently, each key-value operation can be considered a distributed trans-

action. This paper implements strict serializability, i.e., the result of concurrently executing multiple key-value operations is the same as if they were executed one after another in the order of network input. Strict serializability is the strongest consistency standard for distributed transactions.

In the FPGA platform of this article, the latency of a PCIe DMA operation is about 1 μs. When the programmable network card processes key-value requests, if it does nothing else while waiting for the DMA read operation to return, then the throughput of the key-value request will only be about 1 Mops, which is obviously unacceptable. Therefore, high-performance key-value storage on programmable network cards must concurrently execute key-value operations and DMA requests to hide PCIe latency. However, key-value operations may have dependencies, and not all key-value operations can be executed concurrently. For example, a GET operation after a PUT operation on the same key needs to return the updated value. Again, two adjacent atomic increment operations need to wait for the first one to complete before executing the second one. This requires tracking the key-value operations being processed, and pausing the pipeline in the event of a data hazard, or better designing an out-of-order executor to solve data dependencies without explicitly pausing the pipeline.

Allocate load between network card DRAM and host memory. An obvious idea is to use the DRAM on the network card as a cache for host memory, but on the network card, the DRAM throughput (12.8 GB/s) is comparable to the achievable throughput of two PCIe Gen3 x8 interfaces (13.2 GB/s). This article expects to allocate memory access between DRAM and host memory to take advantage of their two bandwidths. However, compared with host memory (64 GiB), the on-board DRAM is small (4 GiB), so a mixed cache and load scheduling method is needed.

The following will introduce KV-Direct, a new FPGA-based key-value storage system that meets all the above design goals.

5.3 KV-Direct Operation Primitives

KV-Direct extends the Remote Direct Memory Access (RDMA) primitives to *remote direct key-value access* primitives, as shown in Table 5.2. The client sends *KV-Direct operations* to the key-value storage server, and the programmable network card processes the requests and sends back the results, bypassing the CPU. The programmable network card on the key-value storage server is an FPGA reconfigured as a *key-value processor*.

In addition to the standard key-value storage operations shown at the top of Table 5.2, KV-Direct also supports two types of vector operations: sending a scalar to the network card on the server, the network card will apply the update to each element in the vector; or sending a vector to the server, and the network card updates the original vector element by element. Furthermore, KV-Direct supports including user-defined functions in atomic operations. User-defined functions need to be pre-registered and compiled into hardware logic, indexed at runtime using the function ID. Key-value operations using user-defined functions are similar to *active messages*[204], saving the communication and synchronization costs of retrieving the key-value to the client for processing.

Table 5.2 KV-Direct operations.

get $(k) \rightarrow v$	Get the value of key k.
put $(k, v) \rightarrow$ bool	Insert or replace the pair (k, v).
delete $(k) \rightarrow$ bool	Delete key k.
update_scalar2scalar $(k, \Delta, \lambda(v, \Delta) \rightarrow v) \rightarrow v$	Atomically update key k using function λ, acting on Δ, returning the original value.
update_scalar2vector $(k, \Delta, \lambda(v, \Delta) \rightarrow v) \rightarrow [v]$	Atomically update all elements in key k using function λ and scalar Δ, returning the original vector.
update_vector2vector $(k, [\Delta], \lambda(v, \Delta) \rightarrow v) \rightarrow [v]$	Atomically update all elements in key k using function λ, based on corresponding elements in vector $[\Delta]$, and return the original vector.
reduce $(k, \Sigma, \lambda(v, \Sigma) \rightarrow \Sigma) \rightarrow \Sigma$	Reduce vector k to a scalar using function λ, and return the reduced result Σ.
filter $(k, \lambda(v) \rightarrow$ bool$) \rightarrow [v]$	Filter elements in vector k using function λ, and return the filtered vector.

When performing vector operation updates (update), reductions (reduce), or filters (filter) on a key, its value is considered an array of fixed-width elements. Each function λ operates on an element in the vector, client-specified parameter Δ, and/or initial value Σ for reduction. Based on the KV-Direct development toolchain from Chapter 4, user-defined function λ is replicated multiple times to exploit the parallelism in FPGA and match the computation throughput with the throughput of other components in the key-value processor, then compiled into reconfigurable hardware logic using High-Level Synthesis (HLS) tools[57-58]. Thanks to the design from Chapter 4, the development toolchain automatically extracts data dependencies in the replicated function and generates fully pipelined programmable logic. Before the key-value storage client starts running, the programmable network card on the key-value storage server should load the hardware logic of user-defined function λ.

Using user-defined functions, common stream processing can be implemented in vector operations. For example, network processing applications can interpret this vector as a packet stream for network functions, or a set of states for packet transactions[233]. It is even possible to implement single-object transaction processing entirely within programmable network cards, such as the operation of resetting S_QUANTITY to zero after reaching the threshold in the TPC-C benchmark test[234]. Vector reduction operations can support the calculation of accumulating neighbor node weights in PageRank[69]. Vector filtering operations can also be used to obtain non-zero values in sparse vectors.

5.4 Key-Value Processor

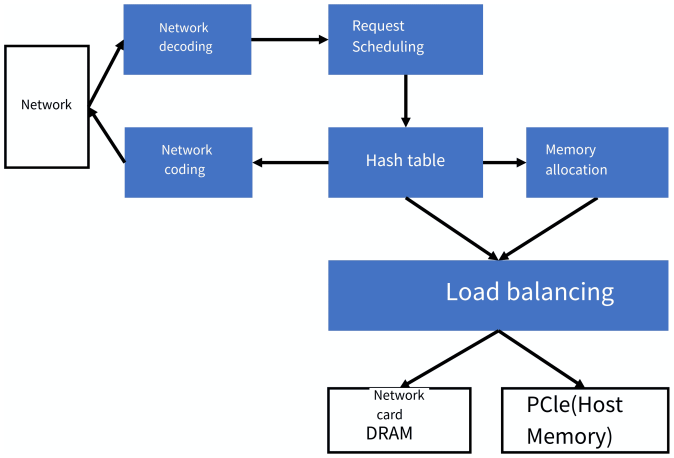


Figure 5.6 Architecture of the key-value processor.

As shown in Figure 5.6, the key-value processor receives packets from the onboard network card, decodes vector operations, and buffers key-value operations in the reservation station ① (Section 5.4.3). Next, the out-of-order execution engine (Section 5.4.3) sends concurrently executable key-value operations from the reservation station to the key-value operation decoder. Depending on the operation type, the key-value processor looks up the hash table (Section 5.4.1) and performs the corresponding operation. To minimize memory access times, smaller key-value pairs are stored inline in the hash table, while other key-value pairs are stored in dynamically allocated memory in the slab memory allocator (Section 5.4.2). Both the hash index and the memory allocated by the slab are managed by a unified memory access engine (Section 5.4.4), which accesses the host memory via PCIe DMA and caches part of the host memory in the onboard DRAM. After the key-value operation is completed, the result is sent

① The reservation station is a concept in computer architecture that stores operations to be executed and schedules appropriate operations for concurrent execution.

back to the out-of-order execution engine (Section 5.4.3), which looks for key-value operations that depend on it in the reservation station and executes them.

As discussed in Section 5.2.6, the scarcity of PCIe throughput requires the key-value processor to save DMA access. For GET operations, at least one memory read is required. For PUT or DELETE operations, for the hash table data structure, at least one read and one write are required [①]. Log-based data structures can achieve less than one write operation per PUT on average, but it sacrifices GET performance. KV-Direct carefully designs the hash table to achieve near-ideal DMA access for each lookup and insertion. KV-Direct also carefully designs the memory allocator so that each dynamic memory allocation averages less than 0.1 DMA operations.

5.4.1 Hash Table

To store variable-sized key-values, key-value storage is divided into two parts. The first part is the hash index (Figure 5.7), which contains a fixed number of *hash buckets*. Each hash bucket contains several *hash slots* and some metadata. The rest of the memory is dynamically allocated and managed by the slab allocator (Section 5.4.2). The *hash index ratio* configured at initialization is the proportion of the size of the hash index to the total memory size of the key-value storage. The choice of hash index ratio will be discussed in Section 5.4.1.

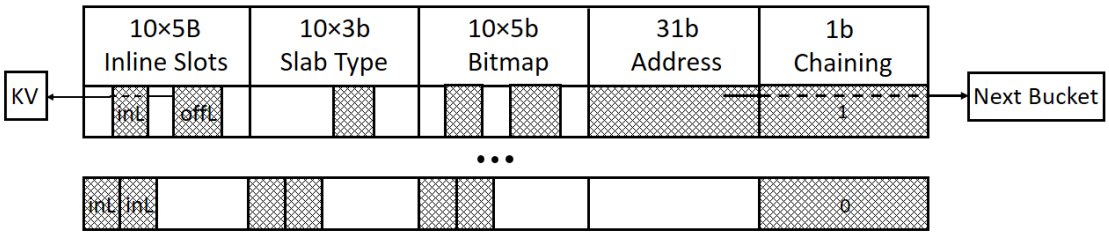


Figure 5.7 Hash index structure. Each row is a hash bucket, containing 10 hash slots, each hash slot includes 3 bits of slab memory type, a bitmap marking the start and end of inline key-value pairs, and a pointer to the next linked bucket in case of hash collision.

Each hash slot includes a pointer to the key-value data in dynamically allocated memory and an *auxiliary hash*. The auxiliary hash is an optimization that uses another hash function independent of the main hash function. Since there are multiple hash slots in each hash bucket, the key-value processor needs to determine which hash slot corresponds to the key to be found. With a 9-bit auxiliary hash, it can be determined with a probability of 511/512 which is the key to be found. However, to ensure correctness, an additional memory access is still needed to fetch the key and compare it byte by

[①] The read operation takes out the key in the hash slot. If the slot is empty or the same as the key to be searched, and no memory space needs to be reallocated, a write operation is required to write back the data

byte. Assuming a 64 GiB key-value storage in the host memory and a 32-byte allocation granularity ①, the pointer requires 31 bits. The size of each hash slot is 5 bytes ②. To determine the size of the hash bucket, a trade-off needs to be made between the number of hash slots in each bucket and the DMA throughput. Figure 5.5a shows that when the granularity is less than 64B, the DMA read throughput is constrained by the PCIe latency and parallelism in the DMA engine. A bucket size less than 64B will increase the possibility of hash collisions. On the other hand, increasing the bucket size to more than 64B will reduce the throughput of hash lookups. Therefore, the bucket size is chosen to be 64 bytes.

The *key-value size* refers to the total size of the key and value. Key-values smaller than the threshold size are stored inline in the hash index to save additional memory accesses to fetch the key-value data. Inline key-values can span multiple hash slots, and their pointer and auxiliary hash fields are reused to store key-value data. Inlining all key-values that can fit into the hash index may not be the best choice. An inline key-value may occupy multiple hash slots, reducing the number of key-values that the hash table can store. If the capacity of the hash table allows, inlining key-values can reduce the average number of memory accesses. For this, KV-Direct selects the *inline threshold* based on the proportion of the hash table that is filled, and inlines key-values smaller than this threshold. Traditionally, the load factor ③ is used to measure the proportion of the hash table that is filled, but this ignores the overhead brought by the metadata and internal fragmentation of the hash table. To compare the choices of different parameters of the hash table more scientifically, this chapter uses *memory utilization* ④. As shown in Figure 5.8, for a certain inline threshold, the average number of memory accesses per key-value operation increases with the increase of memory utilization due to more hash collisions. Under a higher inline threshold, the growth curve of the average number of memory accesses is steeper. Therefore, the optimal inline threshold can be found to minimize the number of memory accesses at a given memory utilization. Like the hash index ratio, the inline threshold can also be configured at initialization.

When all the hash slots in the hash bucket are filled, there are several solutions to solve the hash collision. Cuckoo Hashing[235] and Hopscotch Hashing[236] ensure that

①The 32-byte allocation granularity balances the internal fragmentation and the overhead for memory allocation metadata.

②The design parameters in this paper are configured according to the parameters of the hardware platform used in this paper. For memory of different capacities, parameters such as hash slot size and pointer bit width may change.

③The load factor is the ratio of the number of occupied hash slots to the total number of hash slots.

④Memory utilization is the ratio of the sum of the sizes of all key-values in the key-value storage to the total size of the key-value storage. Due to the existence of metadata and internal fragmentation, memory utilization is always less than 1.

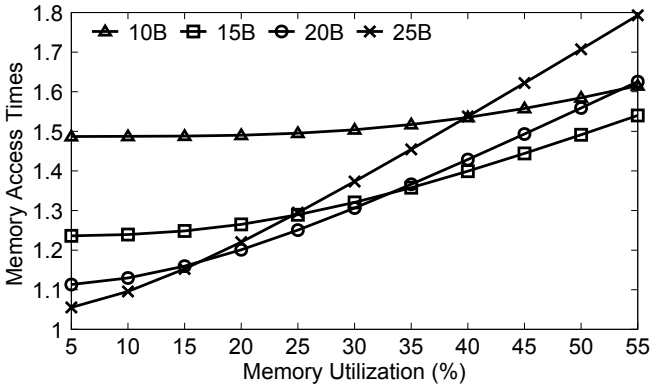
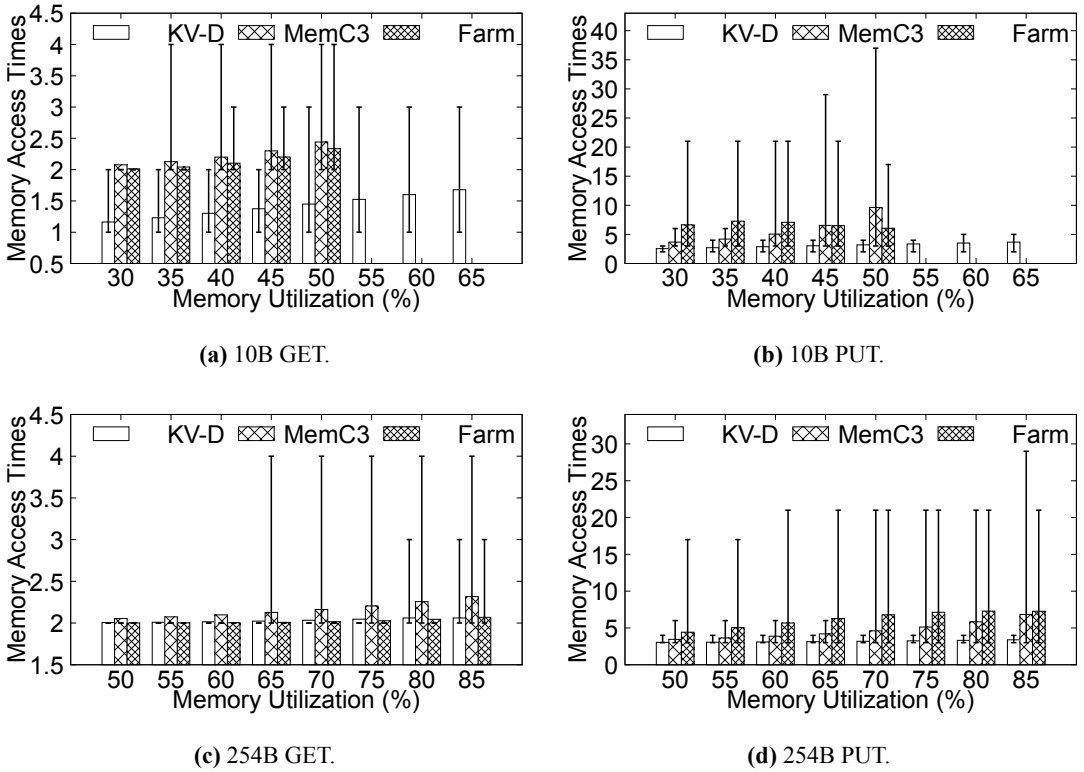


Figure 5.8 Average number of memory accesses and memory utilization under different inline thresholds.

newly inserted key-values are always inserted into the hash bucket by moving occupied hash slots during the insertion process, so that only a constant number of hash slots in the same hash bucket need to be compared during the lookup, achieving constant time lookup. However, in write-intensive workloads, the memory access time under high load rates can fluctuate significantly. In extreme cases, insertion may even fail, requiring hash table expansion. Another solution to hash collisions is *linear probing*, which may be affected by clustering, so its performance is sensitive to the uniformity of the hash function. For this, this paper chooses *chaining* to solve hash collisions, which balances the performance of lookup and insertion, and is more robust to hash clustering.

To compare KV-Direct's chaining, bucketized Cuckoo Hash in MemC3[33], and chain associative Hopscotch Hash in FaRM[70], Figure 5.9 plots the average number of memory accesses per GET and PUT operation in three possible hash table designs. In the KV-Direct experiment, the best choices are made for the inline threshold and hash index ratio for a given key-value size and memory utilization requirement. In the Cuckoo and Hopscotch hash experiments, it is assumed that the key is inlined and can be compared in parallel, and the value is stored in dynamically allocated slab storage. Since the hash tables of MemC3 and FaRM cannot support a memory utilization rate of more than 55

For inline key-values, each GET operation in KV-Direct only requires close to 1 memory access, and each PUT also only requires 2 memory accesses under non-extreme memory utilization. Non-inline key-values have one additional memory access for GET and PUT. Comparing KV-Direct and chained hopscotch hashing under high memory utilization, hopscotch hashing performs better in GET but worse in PUT. Although KV-Direct cannot guarantee the worst-case DMA access times, it will balance between GET and PUT. The GET operation of cuckoo hashing can guarantee access to at most two



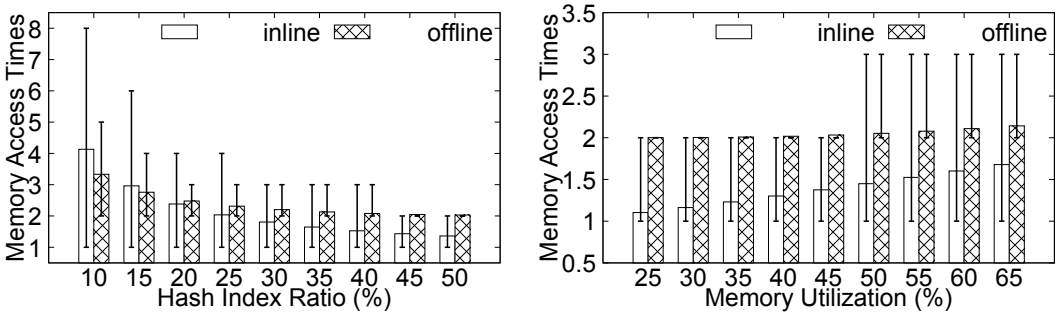
(a) 10B GET. **(b)** 10B PUT.

(c) 254B GET. **(d)** 254B PUT.

Figure 5.9 Number of memory accesses per key-value operation.

hash slots, so under most memory utilizations, KV-Direct has more memory accesses. However, under high memory utilization, cuckoo hashing will cause a large fluctuation in the number of memory accesses for PUT operations.

There are two free parameters in the hash table design: (1) inline threshold, (2) the ratio of hash index in the entire memory space. As shown in Figure 5.10a, as the hash index ratio grows, more key-value pairs can be stored inline, resulting in a lower average number of memory accesses. Figure 5.10b shows the increased number of memory accesses with the use of more memory.



(a) Fixed memory utilization 0.5. **(b)** Fixed hash index ratio 0.5.

Figure 5.10 Memory occupancy at different memory utilization or hash index ratios.

As shown in Figure 5.11, the maximum memory utilization decreases when the

hash index ratio is high, because less memory is available for dynamic allocation. Therefore, to accommodate all key-values to be stored in a given memory space, the hash index ratio has an upper limit. This section chooses this upper limit to obtain the minimum average number of memory accesses, as shown by the dashed line in Figure 5.11, first according to the target memory utilization, to get a large hash index ratio; then according to the hash index ratio, the average number of memory accesses required for the GET operation to find the index can be theoretically obtained.

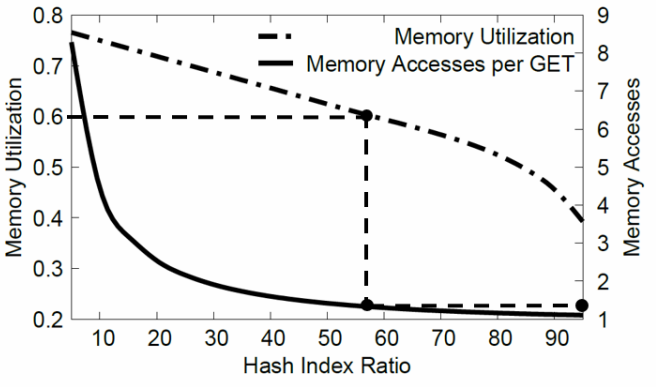


Figure 5.11 How to determine the optimal hash index rate given the memory utilization requirement and key-value size.

5.4.2 Slab Memory Allocator

Chain hash slots and non-inline key-values require dynamic memory allocation. For this, this chapter chooses the slab memory allocator[237] to achieve $O(1)$ average memory access times for each memory allocation and release. The main slab allocator logic runs on the host CPU and communicates with the key-value processor via PCIe. The slab allocator rounds the allocation size to the nearest power of 2, known as *slab size*. It maintains a *free slab pool* for each possible slab size (32, 64, …, 512 bytes) and a global *allocation bitmap* to help merge small free slabs back into larger slabs. Each free slab pool is an array of *slab entries*, consisting of an address field and a slab type field. The slab type field indicates the size of the slab entry.

The available slab pool can be cached on the network card and synchronized with the host memory. Through batch PCIe DMA synchronization operations, each memory allocation or release requires less than 0.07 DMA operations on average. When a free slab pool is almost empty, it is necessary to split larger slabs. Because the slab type is already included in the slab entry, during *slab splitting*, slab entries only need to be copied from the larger pool to the smaller pool, without needing to split into multiple small slab entries.

During reallocation, the slab allocator needs to check whether the released slab

can be merged with its neighbors, requiring at least one read and write to the allocation bitmap. Inspired by garbage collection, *lazy slab merging* on the host merges free slabs in bulk when a slab pool is almost empty and there are not enough slabs in a larger slab pool to split.

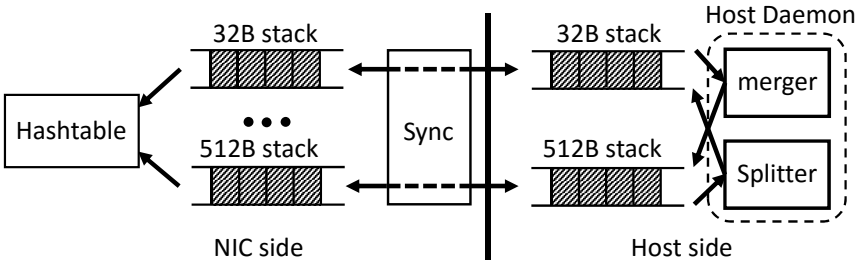


Figure 5.12 Slab memory allocator.

As shown in Figure 5.12, for each slab size, the slab cache on the network card uses two double-end stacks to synchronize with the host DRAM. The left end of the network card's double-end stack (the left side in Figure 5.12) is popped by the allocator and pushed by the deallocator, and the right end synchronizes with the corresponding host end double-end stack via DMA. The network card monitors the size of the network card stack and synchronizes with the host stack according to the high watermark and low watermark. The host daemon periodically checks the size of the host end double-end stack. If it is higher than the high watermark, it triggers slab merging; if it is lower than the low watermark, it triggers slab splitting. Because each end of the double-end stack is exclusively accessed by either the network card or the host, and data is moved before moving the pointer, race conditions will not occur as long as the amount of data in the double-end stack is greater than the protection threshold.

The communication overhead of the slab memory allocator comes from the network card accessing the available slab queue in the host memory. In this chapter, each slab entry is 5 bytes, the DMA granularity is 64 bytes, so the amortized DMA overhead for each slab operation is 5/64 DMA operations. In addition, newly released slab slots on the network card can often be reused by subsequent allocation operations on the network card, so in many cases, no DMA operation is needed at all. To maintain a maximum throughput of 180M operations per second, in the worst case, 180M slab entries need to be transferred, consuming 720 MB/s PCIe throughput, which is 5

The computational overhead of the slab memory allocator comes from slab splitting and merging on the host CPU. Fortunately, they are not often called. For workloads with a stable key-value size distribution, newly released slab slots are reused by subsequent allocations, so they do not trigger splitting and merging.

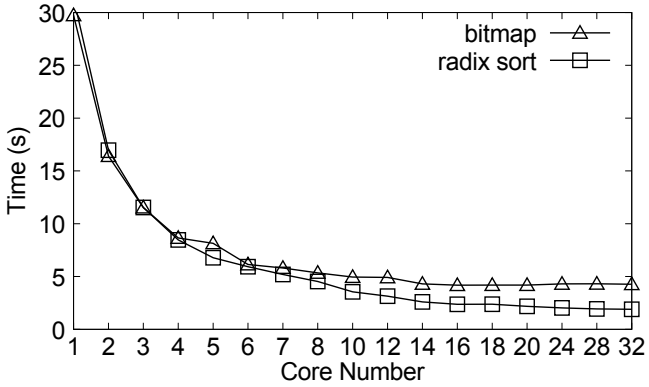


Figure 5.13 Time overhead of merging 4 billion slab slots.

Slab splitting requires moving continuous slab entries from one slab queue to another. When the workload switches from large key-values to small key-values, in the worst case, the CPU needs to move 90M slab entries per second, which only takes up 10

Merging available slab entries into larger slab entries is a fairly time-consuming task, as this garbage collection process needs to fill the allocation bitmap with the addresses of slab entries, thus requiring random memory access. To sort the addresses of available slab entries and merge continuous slabs, radix sort[238] has better multi-core scalability than a simple bitmap. As shown in Figure 5.13, merging all 4 billion free slab slots in a 16 GiB vector takes 30 seconds on a single CPU core, but only 1.8 seconds on 32 cores using radix sort[238]. Although garbage collection of free slab slots takes a few seconds, it runs in the background without stopping the slab allocator, and in fact only triggers when the workload switches from small key-values to large key-values.

5.4.3 Out-of-Order Execution Engine

In the key-value processor, the dependency between two key-value operations with the same key can lead to data hazards and pipeline stalls. This problem is more pronounced in single-key atomics, where all operations are dependent and must be processed one by one, limiting the throughput of atomic operations. This section borrows the concept of *dynamic scheduling* from the field of computer architecture and implements a *reservation station* to track all ongoing key-value operations and their *execution contexts*.

To fully utilize PCIe, DRAM bandwidth, and processing pipelines, up to 256 concurrent key-value operations are needed. However, parallel comparison of 256 16-byte keys would occupy 40% of the FPGA's logical resources. To avoid parallel comparison, this section stores key-value operations in a small hash table in on-chip BRAM,

indexed by the hash of the key. Key-value operations with the same hash value are considered to have dependencies. Different keys may have the same hash value, so there may be false dependencies, but it will never miss dependencies. Key-value operations with the same hash are organized into a linked list structure, processed sequentially by the key-value processor. Hash collisions increase false dependencies and reduce key-value processing efficiency, so the reservation station includes 1024 hash slots, keeping the possibility of hash collisions below 25%.

The reservation station not only saves operations temporarily suspended due to dependencies but also caches recently accessed key-values for *data forwarding*. When the main processing pipeline completes a key-value operation, its result is returned to the client, and the latest value is forwarded to the reservation station. The reservation station checks pending operations in the same hash slot one by one, immediately executes operations with matching keys, and removes them from the reservation station. For atomic operations, calculations are performed in a dedicated execution engine. For write operations, the cached value is updated. The execution result is returned directly to the client. After scanning the dependency linked list, if the value cached in the reservation station has been updated, a PUT operation is issued to the main processing pipeline to write the cache back to the main memory. This data forwarding and fast execution path allows single-key atomic operations to process an operation per clock cycle①, eliminating head-of-line blocking for frequently accessed keys. The reservation station ensures data consistency, as no two operations on the same key can proceed simultaneously in the main processing pipeline. Figure 5.14 describes the structure of the out-of-order execution engine.

The following evaluates the effectiveness of out-of-order execution. The workloads used include single-key atomics and long-tail distribution. The comparison method is a simple method that stalls the pipeline when a key conflict is encountered. The throughput of one-sided RDMA and two-sided RDMA[47] is used as a baseline.

Without the out-of-order execution engine, atomic operations have to wait for PCIe latency and processing latency in the network card, during which subsequent atomic operations on the same key cannot be executed. As shown in Figure 5.15a, the throughput of single-key atomic operations with the pipeline stalling method is 0.94 Mops, close to the 2.24 Mops measured using a commercial RDMA network card[47]. The higher throughput of the commercial RDMA network card can be attributed to its higher clock

①The clock frequency of the FPGA key-value processor in this chapter is 180 MHz, so the throughput can reach 180 M op/s.

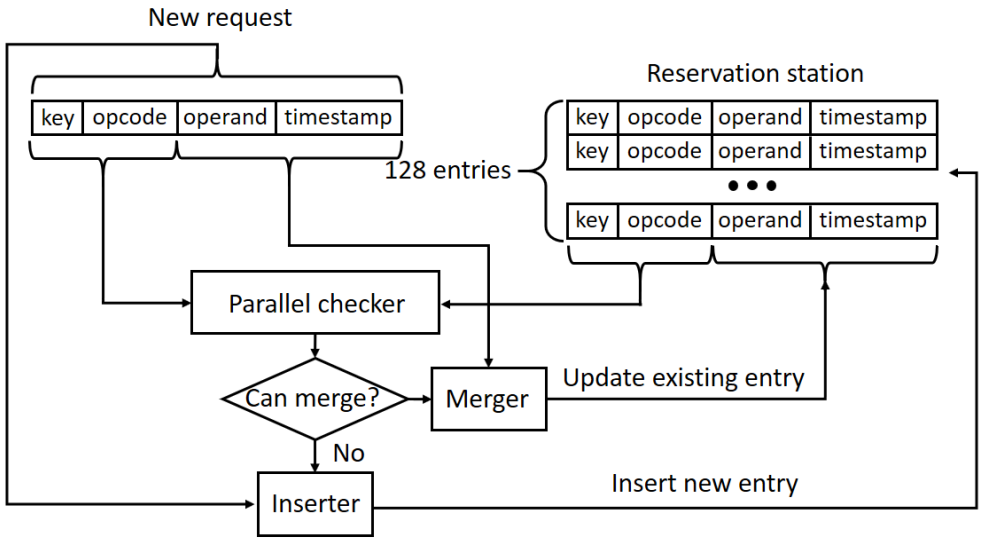
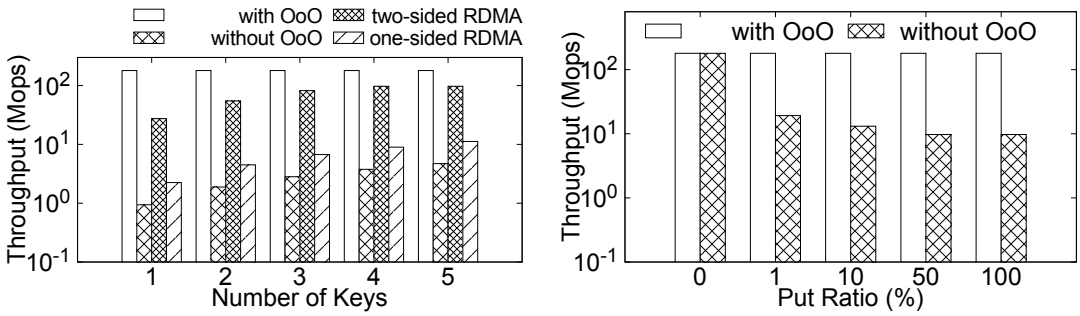


Figure 5.14 Out-of-order execution engine.



(a) Atomic operations. **(b)** Long-tail load.

Figure 5.15 Efficiency of the out-of-order execution engine.

frequency and lower processing latency. With out-of-order execution, KV-Direct's single-key atomic operations can reach peak throughput, processing one key-value operation per clock cycle. In MICA[30], the throughput of single-key atomics is limited by the processing capability of a single CPU core and cannot scale with multiple cores. In fact, the performance of atomic increment operations can scale with multiple cores[47], but it relies on the commutativity between atomic operations, so it is not applicable to non-commutative atomic operations, such as compare-and-swap.

By utilizing out-of-order execution, the single-key atomic throughput has increased by 191 times, reaching the clock frequency limit of 180 Mops. When atomic operations are evenly distributed among multiple keys, the throughput of single-sided RDMA, double-sided RDMA, and KV-Direct without out-of-order execution grows linearly with the number of keys, but it still has a significant gap compared to the best throughput of KV-Direct using out-of-order execution.

Figure 5.15b shows the throughput under long-tail workloads. When a PUT oper-

ation finds any ongoing operation with the same key in the pipeline, the pipeline will pause. Long-tail workloads have multiple keys that are accessed very frequently, so two operations with the same key are likely to arrive almost simultaneously. When the proportion of PUT operations in all operations is higher, it is more likely that at least one of the two operations with the same key is a PUT operation, which triggers the pipeline to pause.

5.4.4 DRAM Load Balancer

To further alleviate the burden of PCIe, this section schedules memory access between PCIe and the onboard DRAM of the network card. The network card DRAM has a capacity of 4 GiB and a throughput of 12.8 GB/s, which is an order of magnitude smaller than the key-value storage on the host DRAM (64 GiB) and slightly slower than the PCIe link (14 GB/s). One method is to put a fixed part of the key-value storage into the network card DRAM. However, the network card DRAM is too small to hold only a small part of the entire key-value storage. Another method is to use the network card DRAM as a cache for the host memory, but the throughput may even decrease due to the limited throughput of the network card DRAM (even lower than the PCIe throughput).

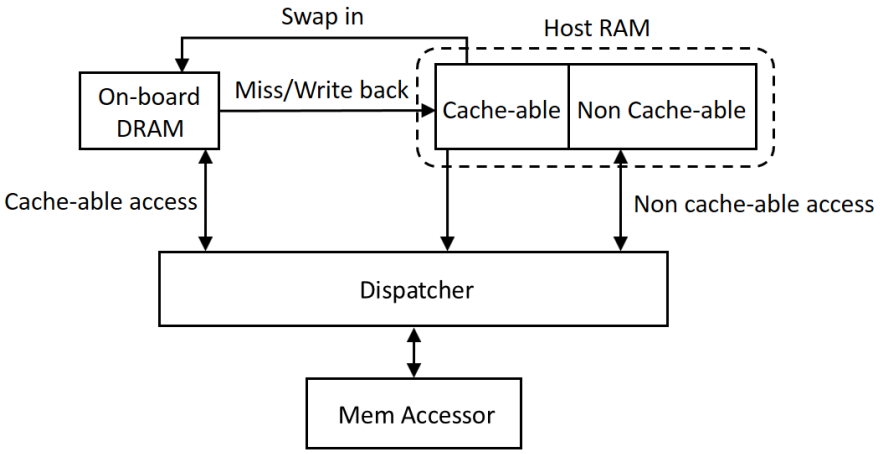


Figure 5.16 DRAM load balancer.

This section adopts a hybrid solution, using DRAM as a cache for a fixed part of the key-value storage in the host memory, as shown in Figure 5.16. The cacheable part is determined by the hash of the memory address, with a granularity of 64 bytes (DRAM memory access granularity). Choose a hash function so that the hash index and the address in the dynamically allocated memory have the same cache probability. The part of the cacheable memory in the entire key-value storage memory is called the *load distribution ratio* (l). If the load distribution ratio l increases, a larger proportion of the load will be allocated to the onboard DRAM, and the cache hit rate $h(l)$ will increase.

To balance the load on PCIe and onboard DRAM, the load scheduling ratio l should be optimized so that:

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

Specifically, under uniform load, let k be the ratio of the size of the onboard DRAM to the size of the host key-value storage, then the cache hit rate $h(l) = \frac{\text{cache size}}{\text{cache-able memory size}} = \frac{k}{l}$. When $k \leq l$, the cache under uniform load is not efficient. Under long-tail load (Zipf distribution), let n be the total number of key-values, then roughly $h(l) = \frac{\log(\text{cache size})}{\log(\text{cache-able part size})} = \frac{\log(kn)}{\log(ln)}$, when $k \leq l$. Under long-tail workloads, the cache hit probability of 1M cache in 1G key-value storage is as high as 0.7. The optimal l can be obtained numerically, which will be discussed in section 5.7.1.

A technical challenge is to store metadata in the DRAM cache. For each cache line of 64 bytes, 4 address bits and a dirty flag bit of metadata are required. Because all key-value storage is accessed by the network card, no cache valid bit is needed. To store the 5 metadata bits of each cache line, if the cache line is expanded to 65 bytes, the DRAM performance will be reduced due to unaligned access; if the metadata is stored elsewhere, the number of memory accesses will double. Instead, this paper uses the spare bits in ECC (Error Correction Code) DRAM for metadata storage. ECC DRAM usually has 8 ECC bits for every 64 bits of data. In fact, to correct a one-bit error in 64 bits of data, only 7 additional check bits are needed. The 8th ECC bit is a parity bit used to detect double-bit errors. When accessing DRAM with a granularity of 64 bytes and in an aligned manner, each 64B data has 8 parity check bits. This paper increases the check granularity of parity from 64 data bits to 256 data bits, so double-bit errors can still be detected. This saves 6 additional bits that can be used to save address bits and dirty flag metadata.

Figure 5.17 shows the improvement in DRAM load scheduling throughput compared to using only PCIe. Under uniform workloads, the caching effect of DRAM can be ignored because its size is only 6

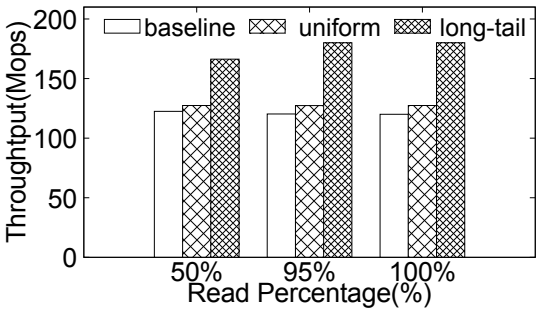


Figure 5.17 DMA throughput under load distribution (fixed load distribution ratio at 0.5).

5.4.5 Vector Operation Decoder

The entire key-value processor design views batch processing as a general principle. This includes batch fetching multiple hash slots in a bucket, batch synchronization of idle slab queues with host memory, lazy slab splitting and merging, and reservation stations batch processing dependent key-value operations along the linked list. Batch processing improves performance by spreading control plane overhead across multiple effective data plane payloads.

Compared to PCIe, the network is a more scarce resource, with lower bandwidth (5 GB / s) and higher latency (2 μs). Ethernet RDMA write packets have 88 bytes of header and padding overhead, while PCIe TLP packets only have 26 bytes of overhead. This is why previous FPGA-based key-value stores[226,239] did not saturate PCIe bandwidth, despite their hash table design being less efficient than KV-Direct. To fully utilize network bandwidth, *client batching* is required in two aspects: batch processing multiple key-value operations in a single packet, and supporting vector operations for more compact representation. For this, a decoder is implemented in the key-value engine to decompress multiple key-value operations from a single RDMA packet. Observing that many key-values have the same size or repeated values, the key-value format includes two flag bits to allow copying key and value sizes, or values of previous key-values in the packet. Fortunately, many important workloads (*such as graph traversal, parameter servers*) can be batched for key-value operations. Looking forward, if higher bandwidth networks can be used, batching will not be necessary.

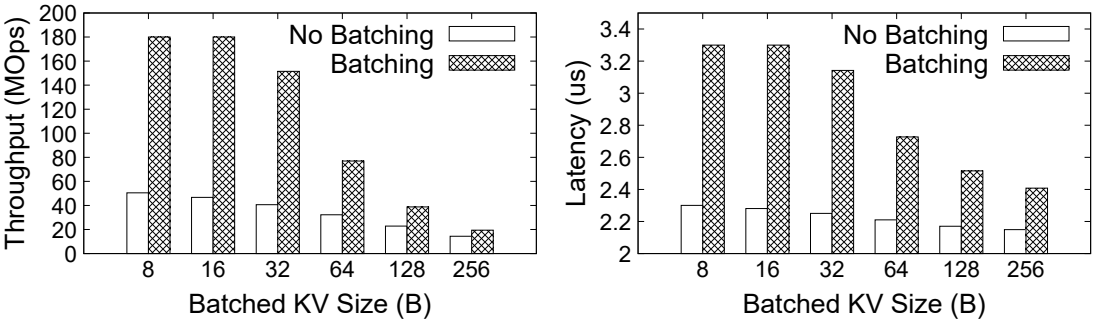
To evaluate the efficiency of vector operations in KV-Direct, Table 5.3 compares the throughput of atomic vector increments with two alternatives: (1) If each element is stored as a different key, the bottleneck is the network transmitting key-value operations. (2) If the entire vector is stored as a large opaque value, retrieved and processed by the client, the overhead of sending the vector over the network is also high. In addition, the two alternatives in Table 5.3 cannot ensure consistency within the vector when accessed by multiple clients simultaneously. Adding synchronization between clients would incur further overhead.

Table 5.3 Throughput of vector operations (GB/s).

Vector size (bytes)	64	128	256	512	1024
Vector update (with return)	11.52	11.52	11.52	11.52	11.52
Vector update (no return)	4.37	4.53	4.62	4.66	4.68
Each element a key	2.09	2.09	2.09	2.09	2.09
Retrieve for client processing	0.03	0.06	0.12	0.24	0.46

KV-Direct clients package key-value operations in network packets to reduce

packet header overhead. Figure 5.18 shows that network batching can increase network throughput by 4 times, while keeping network latency below 3.5 μs.



(a) Throughput. **(b)** Latency.

Figure 5.18 Efficiency of network batching.

5.5 System Performance Evaluation

5.5.1 System Implementation

To improve development efficiency, Intel FPGA SDK for OpenCL[57] is used to synthesize the hardware logic of OpenCL. The key-value processor is implemented with 11,000 lines of OpenCL code, all kernels are fully pipelined, that is, the throughput is one operation per clock cycle. With a clock frequency of 180 MHz, key-value operations can be processed at 180 M op/s, if the network, DRAM, or PCIe is not a bottleneck.

5.5.2 Testbed and Evaluation Method

This section evaluates KV-Direct on a testbed of 8 servers and 1 Arista DCS-7060CX-32S switch. Each server is equipped with two 8-core Xeon E5-2650 v2 CPUs with hyperthreading disabled, forming two NUMA nodes connected by QPI Link. Each NUMA node is equipped with 8 DIMM 8 GiB Samsung DDR3-1333 ECC RAM, with a total of 128 GiB of host memory on each server. The programmable network card[213] is connected to the PCIe root complex of CPU 0, and its 40 Gbps Ethernet port is connected to the switch. The programmable network card has two PCIe Gen3 x8 links in the bifurcated Gen3 x16 physical connector. The tested server is equipped with a Super-Micro X9DRG-QF motherboard and a 120 GB SATA SSD running Archlinux (kernel version 4.11.9-1).

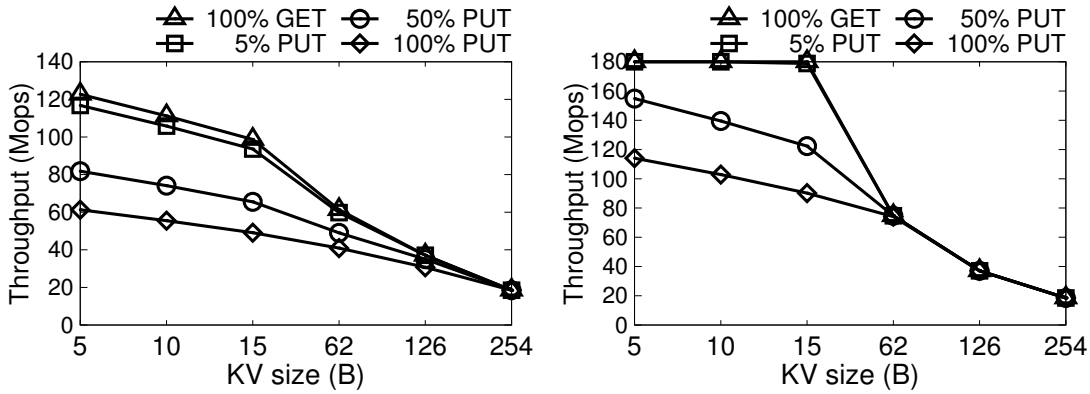
For system benchmarking, the YCSB workload[240] is used. For skewed Zipf workloads, this paper selects a skewness of 0.99 and refers to it as a *long-tail* workload.

Before each benchmark, the hash index ratio, inline threshold, and load distribution ratio are adjusted according to the key-value size, access pattern, and target memory utilization. Then, random key-value pairs of a given size are generated. The key size for a given inline key-value size is irrelevant to the performance of KV-Direct, as the keys are padded to the longest inline key-value size during processing. For testing inline cases, the key-value size is used as a multiple of the slot size (when the size is ≤ 50, i.e., 10 slots). For testing non-inline cases, the key-value size used is a power of 2 minus 2 bytes (for metadata). As the final step of preparation, PUT operations are issued to insert the key-value pairs into the free key-value storage until 50% memory utilization. Performance under other memory utilizations can be obtained from Figure 5.9.

During the benchmark, a FPGA-based packet generator[156] is used in the same ToR to generate batches of key-value operations, send them to the key-value server, receive completions, and measure sustainable throughput and latency. The processing latency of the packet generator is pre-calibrated by direct loopback and removed from the latency measurement. The error lines represent the 5^{th} and 95^{th} percentiles.

5.5.3 Throughput

Figure 5.19 shows the throughput of KV-Direct under YCSB uniform and long-tail (skewed Zipf) workloads. Three factors may be bottlenecks for KV-Direct: clock frequency, network, and PCIe/DRAM. For 5B to 15B key-values inlined in the hash index, most GETs require one PCIe/DRAM access, while PUTs require two PCIe/DRAM accesses. These small key-values are common in many systems. In PageRank, the key-value size of edges is 8B. In sparse logistic regression, the key-value size is typically 8B-16B. For sequencer programs and locks in distributed systems, the key-value size is 8B.



(a) Uniform distribution. **(b)** Long-tail distribution.

Figure 5.19 Throughput of KV-Direct under YCSB workloads.

At the same memory utilization, larger inline key-values have lower throughput due to a higher probability of hash collisions. Key-values of 62B and larger are not inlined, so they require additional memory accesses. The long-tail workload has higher throughput than the uniform workload and can reach the clock frequency range of 180 Mops under read-intensive workloads, or reach the network throughput of ≥ 62B key-value sizes. Under the long-tail workload, the out-of-order execution engine merges about 15% of operations on the most popular keys, and the on-board DRAM has about a 60% cache hit rate at a 60% load distribution ratio, which can result in up to 2x the throughput as a uniform workload. As shown in Table 5.4, the throughput of the KV-Direct network card is comparable to that of state-of-the-art key-value storage servers with dozens of CPU cores.

5.5.4 Power Efficiency

Inserting the KV-Direct NIC can add 10.6 W of power to an idle server. When the KV-Direct server is at peak throughput, the system power is 121.4 watts (measured at the wall). Compared with the most advanced key-value storage systems in Table 5.4, the power efficiency of KV-Direct is three times that of other systems, and it is the first to reach one million key-value operations per watt on a commercial server.

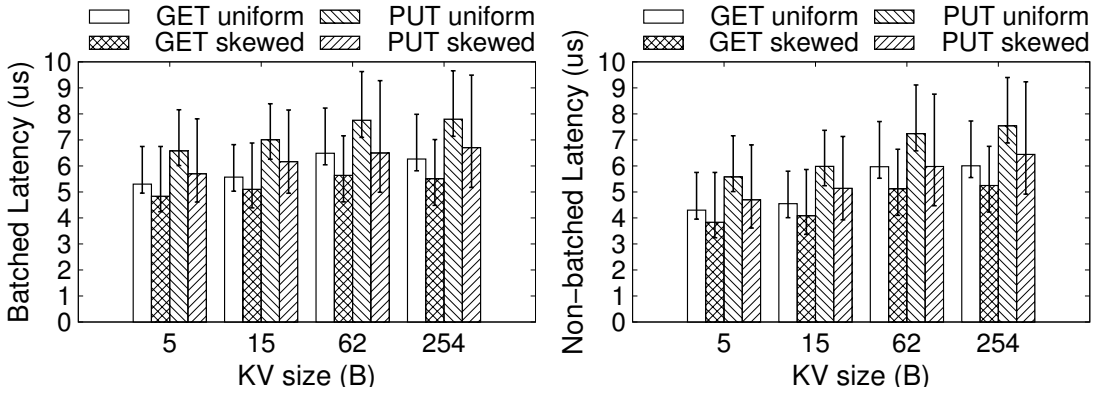
When the KV-Direct NIC is unplugged, the power consumption of the idle server is 87.0 watts, so the total power consumption of the programmable NIC, PCIe, host memory, and daemon on the CPU is only 34 watts. The measured power difference is reasonable because the CPU is almost idle, and the server can run other workloads while KV-Direct is running (using the same standard for one-sided RDMA, as shown in parentheses in Table 5.4). In this respect, the power efficiency of KV-Direct is ten times that of CPU-based systems.

5.5.5 Latency

Figure 5.20 shows the latency of KV-Direct under peak YCSB workload throughput. Without network batching, the tail latency ranges from 3 to 9 μs, depending on the key-value size, operation type, and key distribution. Due to the additional memory access, PUT has a higher latency than GET. Skewed workloads have lower latency than uniform ones because they are more likely to be cached in on-board DRAM. Larger key-values have higher latency due to the additional network and PCIe transfer latency. Network batching adds less than 1 μs of latency compared to non-batching operations, but significantly improves throughput, as evaluated in Figure 5.18.

Table 5.4 Comparison of KV-Direct with other key-value storage systems under long-tail (skewed Zipf) load and 10-byte small keys. For performance numbers not reported in related work, this paper simulates these systems using similar hardware and reports rough measurement results. For CPU bypass systems, the numbers in parentheses report the difference in power consumption under peak load and idle conditions.

Key-value storage Table ??	Note	Performance bottleneck	Throughput (Mops)		Power efficiency (Kops/W)		Average laten
			GET	PUT	GET	PUT	GET
Memcached[205]	Traditional	Inter-core CPU synchronization	1.5	1.5	~5	~5	~50
MemC3[33]	Traditional	Operating system network protocol stack	4.3	4.3	~14	~14	~50
RAMCloud[29]	Kernel bypass	Dispatch thread	6	1	~20	~3.3	5
MICA[30]	Kernel bypass, 24 cores, 12 NIC ports	CPU key-value processing	137	135	342	337	81
FaRM[70]	One-sided RDMA GET	RDMA NIC	6	3	~30 (261)	~15	4.5
DrTM-KV[72]	One-sided RDMA and HTM	RDMA NIC	115.2	14.3	~500 (3972)	~60	3.4
HERD'16[47]	Two-sided RDMA, 12 cores	PCIe	98.3	~60	~490	~300	5
Xilinx'13[239]	FPGA	Network	13	13	106	106	3.5
Mega-KV[209]	GPU (4 GiB on-board RAM)	GPU key-value processing	166	80	~330	~160	280
KV-Direct (1 NIC)	Programmable NIC, two Gen3 x8	PCIe & DRAM	180	114	1487 (5454)	942 (3454)	4.3
KV-Direct (10 NICs)	Programmable NIC, one Gen3 x8 per card	PCIe & DRAM	1220	610	3417 (4518)	1708 (2259)	4.3



(a) With batching. **(b)** Without batching.

Figure 5.20 Latency of KV-Direct under peak YCSB workload throughput.

5.5.6 Impact on CPU Performance

KV-Direct aims to bypass the server CPU, using only a portion of host memory for key-value storage. Therefore, the CPU is still available to run other applications. When a single NIC KV-Direct is at peak load, the measured impact on other workloads on the server is minimal. Table 5.5 quantifies the impact of KV-Direct peak throughput. Except for the sequential throughput of CPU 0 to access its own NUMA memory (rows marked in bold), the latency and throughput of CPU memory access are mostly unaffected. This is because the 8 host memory channels can provide higher random access throughput than all CPU cores can consume, while the CPU can indeed stress the sequential throughput of the DRAM channels. The impact of the host daemon process is minimal when the distribution of key-value sizes is relatively stable, as the garbage collector is only invoked when the number of available slots for different board sizes is unbalanced.

Table 5.5 Impact on CPU memory access performance when KV-Direct is at peak throughput. Measured using Intel Performance Counter Monitor (Intel PCM) V2.11.

KV-Direct Status →		Idle	Busy
Random Access Latency	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
Sequential Access Throughput	**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
Random Access Throughput	32B Read	10.53 GB/s	10.46 GB/s
	64B Read	14.41 GB/s	14.42 GB/s
	32B Write	9.01 GB/s	9.04 GB/s
	64B Write	12.96 GB/s	12.94 GB/s

5.6 Extensions

5.6.1 CPU-based Scatter-Gather DMA

For 64B DMA operations, PCIe has a 29% TLP header and padding overhead (§5.2.6), and the DMA engine may not have enough parallelism to saturate the PCIe Bandwidth-Delay Product (BDP) with small TLPs. The PCIe root complex in the system supports larger DMA operations, up to 256 bytes of TLP payload. In this case, the TLP header and padding overhead is only 9%, and the DMA engine has enough parallelism (64) to saturate the PCIe link with 27 ongoing DMA reads. To batch DMA operations on the PCIe link, the CPU can be utilized to perform scatter-gather (Figure 5.21). First, the NIC DMA sends addresses to a request queue in host memory. The host CPU polls the request queue, performs random memory accesses, places data into a response queue, and writes an MMIO doorbell to the NIC. Then, the NIC extracts data from the response queue via DMA.

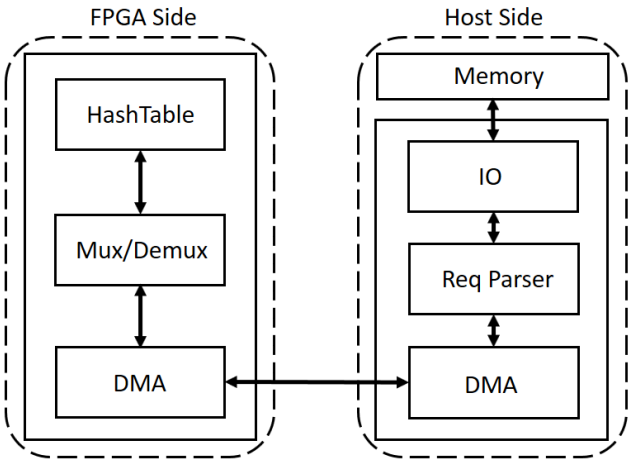
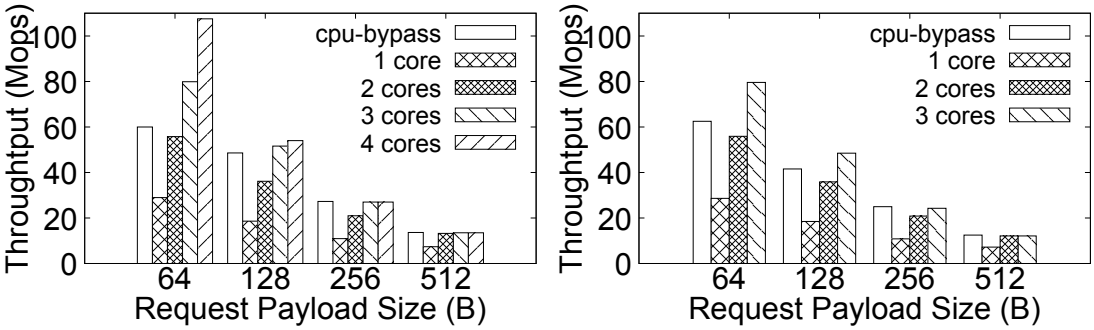


Figure 5.21 Scatter-gather architecture.

Figure 5.22 shows that, compared to the CPU bypass method, the throughput of CPU-based scatter-gather DMA is improved by 79%. Besides the CPU overhead, the main disadvantage of CPU-based scatter-gather is the additional latency. To batch 256 DMA operations per doorbell from the CPU to the NIC, it takes 10 μs to complete. The total latency for the NIC to access host memory using CPU-based scatter-gather is about 20 μs, nearly 20 times higher than direct DMA.

5.6.2 Single-Host Multi-NIC

The primary use case of KV-Direct is to enable remote direct key-value access without CPU overhead on the server. In some cases, it may be necessary to build a dedicated key-value store with maximum throughput per server. Through simulation,[31]



(a) Read operations. **(b)** Write operations.

Figure 5.22 Scatter-gather performance.

showed the possibility of achieving one billion key-value operations on a single server with four (currently unavailable) 60-core CPUs. As shown in Table 5.4, with 10 KV-Direct NICs on the server, it is easy to achieve one billion key-value op/s performance using a commercial server.

As shown in Figure 5.23, the server consumes 357 watts of power (measured at the wall) to achieve 1.22 Gop/s GET or 0.61 Gop/s PUT performance.

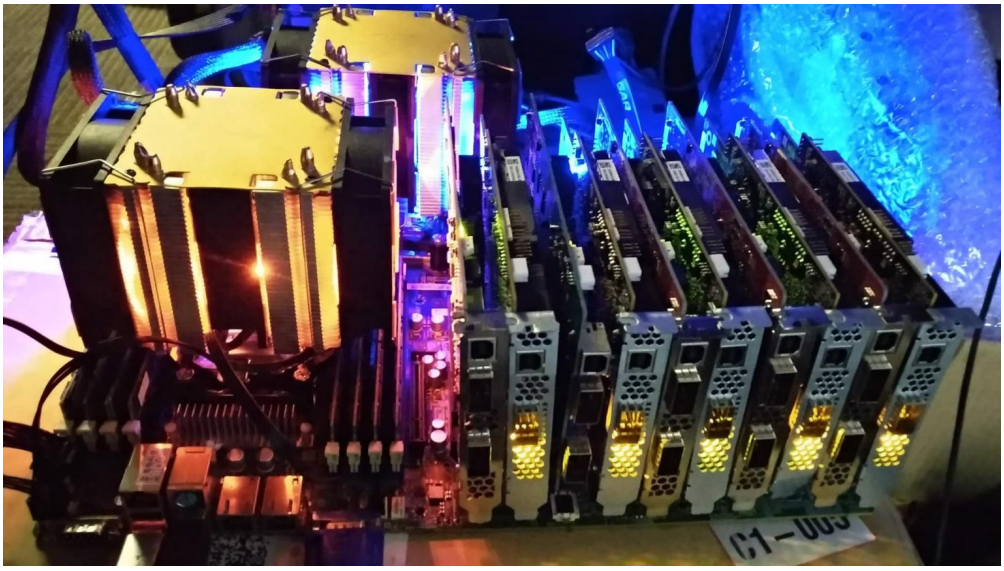


Figure 5.23 10-card KV-Direct system achieving 1.22 billion key-value operations per second at 357 watts power consumption.

To saturate the 80 PCIe Gen3 lanes of two Xeon E5 CPUs, the motherboard of the benchmark server was replaced with a SuperMicro X9DRX+-F motherboard with 10 PCIe Gen3 x8 slots, the PCIe topology is shown in Figure 5.24.

Each of the 10 programmable network cards on each slot is connected using a PCIe x16 to x8 converter, with only one PCIe Gen3 x8 link enabled on each network card, so the throughput of each network card is lower than that shown in Figure 5.19. Each network card has an exclusive memory area in the host memory and provides disjoint

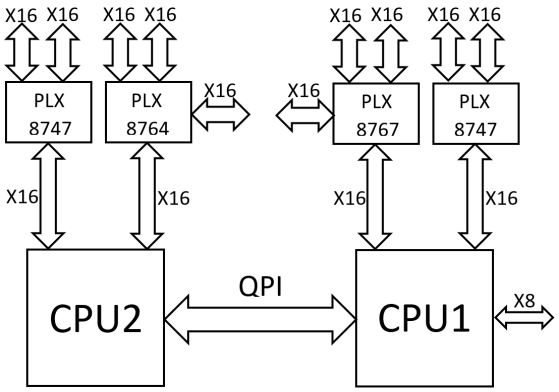


Figure 5.24 PCIe topology of the 10-card KV-Direct system.

key partitions. Multiple network cards encounter the same load imbalance problem as multi-core key-value storage implementations. Fortunately, for a small number of partitions (such as 10), load imbalance is not important[30-31]. Under the YCSB long-tail workload, the average load of the network card with the highest load is 1.5 times, and the load increase of very popular keys is provided by the out-of-order execution engine (§5.4.3). In contrast, to achieve performance matching with 240 CPU cores, the load of the hottest CPU core will be 10 times the average. Figure 5.25 shows that the throughput of KV-Direct is almost linearly related to the number of network cards on the server.

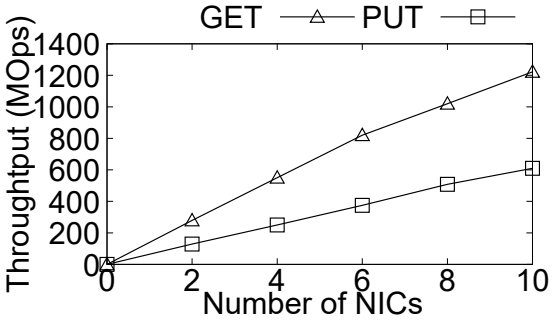


Figure 5.25 Performance scalability of multiple network cards on a single machine.

5.6.3 SSD-based Durable Storage

Data will be lost after power failure in memory-based data structure storage. For persistence, this section implements persistent key-value storage using SATA SSD. Since the number of SSDs on the server is limited, and the operating system and applications also run on the SSD, the key-value storage needs to share the SSD hardware with the operating system and applications. For this, the SSD provides two access interfaces: block storage and key-value storage. Similar to the dedicated memory space used by memory key-value storage, key-value storage is also located in a dedicated block storage space. The CPU needs two ways to access block storage: one is to access

the block device through the operating system's storage protocol stack, and the other is to bypass the operating system and directly access it through the user-mode fast interface. Since the operating system itself and many software run on block storage, it is necessary to maintain the compatibility of the first traditional access method. Storage performance-sensitive applications use the runtime library provided in this paper to access block storage through the second method.

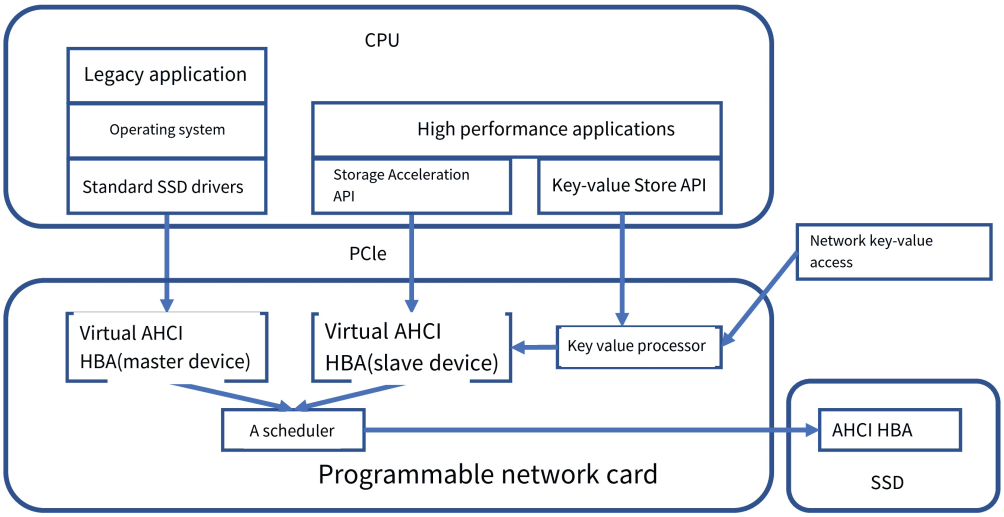


Figure 5.26 SSD persistent storage architecture.

As shown in Figure 5.26, the programmable network card virtualizes the SSD into two virtual AHCI HBA devices. The scheduler inside the programmable network card virtualizes the data plane of the storage hardware (such as the 32 request slots of SATA and the request queue of NVMe) into two logical storage devices. The control registers of the storage hardware (such as PCIe configuration registers) are transparently passed to the main logical storage device and managed by the original operating system. The secondary logical storage device has no control plane, only a data plane, and can only perform data read and write operations, and cannot perform management operations. The above storage virtualization architecture does not require the programmable network card to manage the control plane, simplifying the design of the scheduler; and it maintains compatibility with the original storage device driver and operating system. As shown in Figure 5.27, the sequential access throughput of the original operating system and software after storage virtualization through the main logical storage device has not changed significantly. The delay has increased from about 30 μs to about 60 μs, which is the overhead of the programmable network card forwarding. Due to the increase in latency, the throughput of single-threaded random access (4K block size) has also decreased. When 64 threads are randomly accessed, because SATA only has 32 request slots, that is, only 32 read and write requests can be performed in parallel,

the throughput is limited by the average delay of the request.

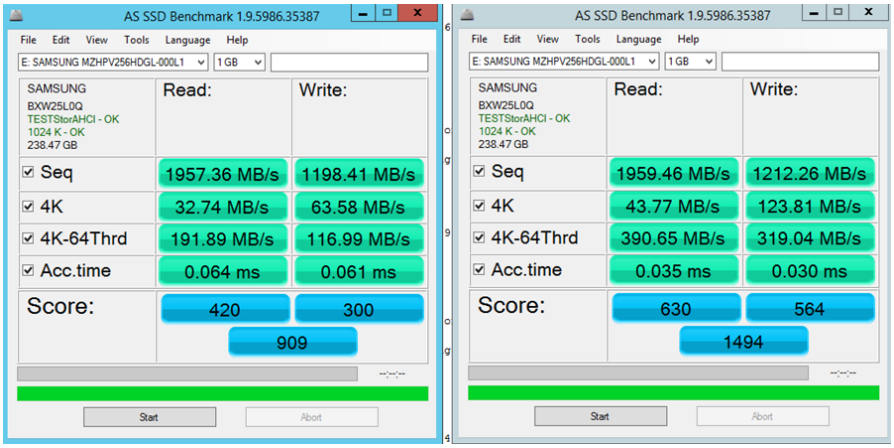


Figure 5.27 Performance evaluation of SSD virtualization. The left figure is the result of the original operating system and SSD performance test program after storage virtualization. The right figure is the performance test result of this SSD without using storage virtualization.

In order to provide an efficient block device access interface, this section uses the PCIe I/O pipeline of Chapter 4 to allow applications to directly access the programmable network card through the storage acceleration API, bypassing the operating system and driver. The storage acceleration API can access any storage block on the SSD, and it is the responsibility of the application to avoid conflicts with the file system of the operating system. Usually, the application creates a large file to reserve storage space. Experiments show that with only a single CPU thread, the storage acceleration API can fully utilize the sequential read and write throughput of about 2 GB/s on the SSD and the random read and write throughput of about 50 K times per second with a 4K block size. The traditional operating system storage protocol stack requires 8 CPU threads to fully utilize the throughput of random read and write with a 4K block size. In order to provide persistent key-value storage, the key-value processor is connected to the slave device of virtual storage, and treats virtual storage as a large piece of memory for reading and writing. This paper does not optimize for the read and write characteristics of SSD, which will be future work.

5.6.4 Distributed Key-Value Storage

In distributed key-value storage, we assume that each host is both a client using key-value storage and can allocate some memory resources as a key-value server. Each key-value pair needs to be replicated on multiple server nodes to provide high availability and improve the performance of key-value access. Traditional key-value storage services simply select servers based on the hash value of the key. However, not every host accesses each key with the same probability. For example, in graph computing,

if a host is responsible for processing a vertex, then the probability of accessing the key-value corresponding to the vertex will be higher than other hosts. At this time, the key-value pair of the vertex is best stored on the host to take advantage of locality to accelerate access.

Distributed storage systems need to decide on which hosts each key-value pair is replicated. Read operations only need to read the nearest replica. Write operations need to be synchronized to all replicas through the master node. If there are too many replicas, the master node synchronizing write operations to each replica will become a bottleneck. To balance the load between the master node and each replica node, the number of replicas depends on the read-write ratio. Assuming the ratio of read and write operations is R, and R is much greater than 1 ①. It can be derived that when the load of the master node is equal to that of the replica node, the number of replicas is approximately $\sqrt{R}$. Compared with a single replica, the load of read operations is reduced by about $\sqrt{R}$ times; compared with replication to each host, the load of write operations is reduced by about $\sqrt{R}$ times.

After deciding on the number of replicas, the next question is to select the most frequently read hosts for replication. This paper maintains an approximate counter for read and write times for each key in the programmable network card of the master node ②. When write operations are synchronized to each replica, the master node summarizes the read times of all replicas. The optimal number of replicas can be calculated based on the ratio of read and write times, and the number of replicas can be increased or decreased when it differs significantly from the current number of replicas, so that the replicas are stored on several nodes with the most read times. To prevent the master node from not being able to respond to a large number of read requests in a timely manner due to scarce write operations, the replica node can also actively report to the master node after receiving a large number of read requests.

Another problem is that the access frequency of some keys may be so high that the throughput of a single machine may become a bottleneck. For example, the serial number generator in distributed transactions, the access counter of popular network resources, and the lock of shared resources require high-throughput single-key atomic operations. Recent works such as NetCache[241], NetChain[242] use programmable switches as caches to achieve higher single-key read and write performance than a single

① In write-intensive workloads, if there is only one master node, not making any replicas is obviously the most performance optimal. For high availability, it may be necessary to limit the minimum number of replicas.

② The approximate counter is to save storage space. For example, the approximate count is represented by an 11-bit significand and a 5-bit exponent floating point number. Each time you access, the significand is incremented with a probability of one in the power of 2.

host, but they are still limited by switch performance.

This paper proposes that with multi-master replication, single-key read and write performance can scale with the number of hosts and ensure strong consistency. The mechanism to ensure consistency is the token ring, at any time there is only one node holding the token processing write operations. The token is passed in order on the ring formed by each master node and carries the current latest value. The key to the token ring's ability to improve single-key performance is that read and write operations and many atomic operations are associative. For example, multiple write operations can be reduced to the last write operation; multiple atomic addition and subtraction operations can be reduced to one atomic addition and subtraction operation; compare and swap atomic operations are more complex, but multiple operations can also be reduced to a lookup table of size not exceeding the number of atomic operations. Therefore, when a node does not hold a token, it records the received operations in the buffer and reduces these operations. When the token arrives, the node can quickly execute the reduced operation based on the latest value, and send the updated value with the token to the next node. After that, the node replays the operations in the buffer and returns the result of each operation to the client. Theoretical analysis shows that when requests arrive evenly at each master node, the system's throughput and average delay increase linearly with the number of master nodes. The worst delay is the time it takes for the token ring to turn around. Since the operations in the buffer are reduced, the processing delay of each node is bounded, and the time for the token ring to turn around is also bounded. The size of the request buffer required for each master node is equal to the product of the worst delay and network bandwidth. The overhead of the token ring is a packet that is constantly passed in the network, even if there are no read and write requests, the token has to be constantly passed in the network.

The actual key-value storage system has more than one frequently accessed key. The network overhead of setting a token for each key is too high, so this paper uses one token to serve all keys. However, waiting for all keys to be processed and then sending all updated key-value pairs in a concentrated manner will bring higher latency. Assuming that the keys accessed over a period of time are random, that is, most key-value operations involve non-repetitive keys. At this time, the size of the key-value update carried by the token is proportional to the number of key-value operations in the buffer, and the time to transmit these key-value updates is no longer bounded, so the request delay will tend to infinity as the load factor approaches 1. For this reason, this paper pipelines key-value processing and transmission between adjacent master nodes.

The master node organizes the key-value operations in the buffer into a priority queue sorted by key in ascending order, and the key-value updates are also sent in ascending order of keys, and the token indicates the end of the key-value update sequence. When the master node receives the update of key K, the key-value operations in the buffer that do not exceed K can be processed and sent to the next master node along the token ring. In this way, multiple master nodes on the token ring can concurrently process different keys, that is, the master nodes closer to the direction of token passing process smaller keys. Theoretical analysis shows that when the keys of the requests follow a uniform distribution and the requests arrive evenly at each master node, the request delay is still bounded, regardless of the load factor, and is only slightly higher than the delay in the single-key situation.

5.7 Discussion

5.7.1 Network Interface Card Hardware of Different Capacities

The goal of KV-Direct is to offload significant workloads (key-value access) using existing data center hardware, rather than designing special hardware to achieve maximum key-value storage performance. Programmable network cards usually contain a limited amount of DRAM for buffering and connection state tracking. Large DRAM is expensive in terms of chip size and power consumption.

Even if future network cards have faster or larger onboard memory, under long-tail workloads, the load distribution design of this paper (§5.4.4) still shows higher performance than simple partition design. Keys are unified according to the capacity of the network card and host memory. Table 5.6 shows the optimal load distribution ratio for long-tail workloads with 1 billion keys, different network card DRAM and PCIe throughput, and different network card and host memory size ratios. If the network card has faster DRAM, more load is dispatched to the network card. A load distribution ratio of 1 indicates that the behavior of the network card memory is exactly the same as the cache of the host memory. If the network card has larger DRAM, slightly less load is dispatched to the network card. As shown in Table 5.7, even if the size of the network card DRAM is only a small part of the host memory, the throughput gain is significant.

The out-of-order execution engine (§5.4.3) can be applied to various applications that need to hide latency, and this paper hopes that future RDMA network cards can support more powerful atomic operations.

In a 40 Gbps network, network bandwidth limits the key-value throughput of non-

Table 5.6 Optimal load distribution ratio for long-tail workloads under different network card DRAM / PCIe throughput (vertical) and network card / host memory size ratio (horizontal).

	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

Table 5.7 Relative throughput of load dispatch compared to simple partitioning. Row and column titles are the same as Table 5.6.

	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

batch transfers, so this paper uses client batching. With higher network bandwidth, the batch size can be reduced, thereby reducing latency. In a 200 Gbps network, the KV-Direct network card can reach 180 Mop/s without batch transfer.

KV-Direct utilizes widely deployed programmable network cards and FPGA implementation[48,213]. FlexNIC[127,243] is another promising architecture of programmable network cards with reconfigurable match-action tables (RMT)[107]. NetCache[241] implements key-value caching in programmable switches based on RMT, showing the potential to build KV-Direct in network cards based on RMT.

5.7.2 Performance Impact on Real-world Applications

When KV-Direct is applied to end-to-end applications, backend computation shows potential performance improvement. In PageRank[69], each edge traversal can be achieved through a single key-value operation, thus KV-Direct supports 1.22G TEPS on a server with 10 programmable network cards. In contrast, GRAM[97] supports 250M TEPS per server, constrained by interleaved computation and random memory access.

KV-Direct supports user-defined functions and vector operations (Table 5.2), which can further optimize PageRank by offloading client computation to hardware. Similar parameters apply to the parameter server[81]. This paper hopes that future work can leverage hardware-accelerated key-value storage to improve the performance of distributed applications.

5.7.3 Stateful Processing in Programmable Network Cards

The ClickNP architecture of Chapter 4 is more suitable for stateless or simple state pipeline data packet processing, but it is inadequate for processing based on connection state or application layer requests. For example, the scheduler and hash table in the Layer 4 load balancer application are tightly coupled with the application logic, making it difficult to scale to a large number of concurrent connections, and the code maintainability is not strong. In fact, a lot of time was spent in the development process to solve deadlock problems.

KV-Direct proposes a new architecture for stateful processing. The fundamental difference between KV-Direct and traditional general-purpose processor architectures lies in the separation of control, computation, and memory access. In traditional processors, computation and memory access share the same instruction stream, so for serial programs alternating between computation and memory access, the computation and memory access logic often need to wait for each other, resulting in neither the computation nor memory access throughput being fully utilized. KV-Direct uses separate control instruction streams, computation instruction streams, and memory access instruction streams, and exploits the parallelism between KV operations with an out-of-order execution engine, allowing computation and memory access to be fully pipelined. Developers need to divide the request processing process into several stages alternating between computation and memory access, and abstract the request processing process into a state machine. The control instruction stream manages the state of the request and dispatches the next stage task to the computation or memory access components. After a stage of the task is completed by the computation or memory access components, it returns to the controller.

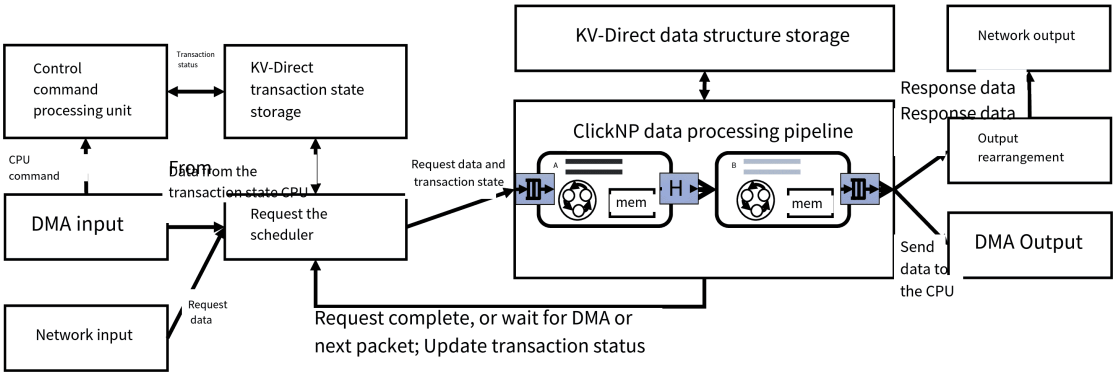


Figure 5.28 Application layer architecture based on KV-Direct for programmable network cards.

The stateful processing architecture based on KV-Direct is shown in Figure 5.28. Transactions represent dependencies, such as a connection in stateful network process-

ing, an application layer HTTP request split across multiple packets, or operations on the same key in key-value storage. Different requests within the same transaction need to be processed in sequence, while requests in different transactions can be processed concurrently. To hide latency and maximize concurrent processing capability, the request scheduler looks up the transaction number corresponding to the request from the transaction state key-value storage based on KV-Direct, and queues the requests of the transactions being processed. The data processing pipeline based on ClickNP processes according to the request data and transaction state, and may query other data structures (such as memory allocation tables, host virtual address mapping tables, firewall rule tables, routing tables, etc.) during the processing. If the request processing is completed, the response data enters the output rearrangement module, rearranges the order of responses to meet the consistency requirements of transaction processing (for example, requests from different transactions also need to respond in the order of arrival), and finally outputs to the network. If the processing of the request still depends on the next packet or data DMA from the host memory, in order not to block the data processing pipeline, the request will return to the scheduler, waiting for the dependent operation to complete before proceeding to the next stage of processing.

The KV-Direct architecture can serve as the basis for many programmable network card applications, such as the stateful network functions in Chapter 4 (such as Layer 4 load balancers) and the scalable RDMA in Chapter 6.

5.7.4 Extending from Key-Value to Other Data Structures

KV-Direct implements a hash table data structure for key-value mapping. Key-value storage systems represented by Redis[78] in data centers also support secondary indexes, ordered key-value sets, message queues, and other data structures. The essential features of these more complex data structures are stateful processing within programmable network cards (Section 5.7.3).

For message queues, the advantage of FPGA processing is fast centralized coordination. In the producer-consumer model, the message queue needs to distribute the producer's messages to each consumer, which is a centralized FIFO abstraction. Due to the limited single-core processing capability of the CPU, parallel or distributed message queues often need to sacrifice some consistency ① to improve performance scalability. However, the clock frequency of FPGA hardware logic is sufficient to handle requests at 100 Gbps line speed, without sacrificing consistency.

① Consistency refers to the first-come, first-served order feature

In the message queue based on KV-Direct, messages are stored in a circular linked list composed of fixed-size buffers. The use of a circular linked list facilitates the allocation of new fixed-size buffers when space is insufficient, and also facilitates the recycling of buffers that have been idle for a long time. For the producer-consumer model, a head pointer is maintained for all producers, and a tail pointer is maintained for all consumers. For the publisher-subscriber model, a head pointer is maintained for all publishers, and a tail pointer is maintained for each subscriber, allowing different subscribers to receive messages at different speeds.

In the secondary index, requests are queued in the request scheduler according to the primary index, and the metadata matching the primary index of the secondary index is obtained using the standard KV-Direct method. The metadata is stored in the transaction status storage as a cache. Then a new request is generated for the secondary index query, which is queued in the request scheduler according to the merged primary and secondary index. When processing this request, the corresponding metadata is taken from the transaction status storage and a second DMA query is initiated. In addition, another complexity of the secondary index compared to the single-level index is that the secondary index may often need to change the size of the hash table when adding or deleting elements, thus requiring rehashing, during which the operations corresponding to the primary index need to be suspended and wait. The advantage of using FPGA to handle secondary indexes is the fine-grained memory access latency hiding, such as other secondary indexes can operate during the rehashing of a certain secondary index.

5.8 Related Work

Driven by performance, the research and development of distributed key-value storage systems, an important infrastructure, has received considerable attention. A large number of distributed key-value storages are based on CPU. To reduce computational costs, Masstree[32], MemC3[33], and libcuckoo[34] optimize locks, caches, hashes, and memory allocation algorithms, while KV-Direct comes with a new hash table and memory management mechanism specifically designed for FPGA to minimize PCIe traffic. MICA[30] partitions the hash table to each core, thus completely avoiding synchronization. However, this method introduces core imbalance for skewed workloads.

To get rid of the overhead of the operating system kernel, Netmap[13] and DPDK[206] directly poll network packets from network cards, while mTCP[17] and

SandStorm[16] use user-mode lightweight network stacks to handle these packets. Key-value storage systems[27-31] benefit from such high-performance optimization. As another step in this direction, recent works[35,46,46-47,47] leverage the hardware-based network stack of RDMA network cards, using two-sided RDMA as the RPC mechanism between key-value storage clients and servers to further increase per-core throughput and reduce latency. Nevertheless, these systems are still CPU-bound (§5.2.5).

A different approach is to utilize one-sided RDMA. Pilaf[219] and FaRM[70] adopt one-sided RDMA reads for GET operations, with FaRM achieving network-saturating throughput. Nessie[218], DrTM[72], DrTM+R[73], and FaSST[20] utilize distributed transactions to implement one-sided RDMA GET and PUT. However, the performance of PUT operations is inevitably affected by the synchronization overhead of consistency guarantees, and is limited by RDMA primitives[47]. In addition, the client CPU is involved in key-value processing, limiting the throughput per core to about 10 Mops on the client side. In contrast, KV-Direct extends RDMA primitives to key-value operations, ensuring server-side consistency, making key-value storage clients completely transparent, and achieving high throughput and low latency, even for PUT operations.

As a flexible and customizable hardware, FPGA is now widely deployed at data center scale[48,213], and significant improvements have been made for programmability[54,156]. Some early works have explored building key-value storage systems on FPGA. However, some of them only use on-chip data storage (about a few MB of memory)[244] or on-board DRAM (e.g., 8 GB of memory)[221-222,224], thus the storage capacity is limited. The work[226] focuses on increasing system capacity rather than throughput, and uses SSD as a secondary storage for on-board DRAM. The work[222,244] can only store fixed-size key-value pairs, such key-value storage systems can only be used for some specific applications, and are not general enough. The work[223,239] uses host DRAM to store hash tables, and the work[245] uses network card DRAM as a cache for host DRAM, but they do not optimize for network and PCIe DMA bandwidth, resulting in poor performance. KV-Direct makes full use of network card DRAM and host DRAM, making FPGA-based key-value storage systems general and capable of large-scale deployment. In addition, careful hardware and software co-design, as well as optimization for PCIe and network, push the performance of this paper to the physical limit.

The use of programmable switches supporting P4[106] to accelerate key-value storage systems has been a hot research topic in recent years[111]. SwitchKV[246] uses content-based routing to route requests to backend nodes based on cached keys,

while NetCache[241] further caches frequently accessed key-values in the switch. NetChain[242] implements a highly consistent, fault-tolerant key-value store in network switches.

Secondary indexing in data storage systems, which allows data retrieval using keys other than the primary key, is an important feature[247-248]. SLIK[248] supports multiple secondary keys in key-value storage systems using the B+ tree algorithm. Exploring how to support secondary indexing to help KV-Direct move towards a general data storage system would be interesting. SwitchKV[246] uses content-based routing to route requests to backend nodes based on cached keys, while NetCache[241] further caches key-values in the switch. This load balancing and caching will also benefit the system. Eris[229] uses a network sequence generator to implement efficient distributed transactions, which can bring new life to the one-sided RDMA methods for client synchronization.

5.9 Chapter Summary

This chapter describes the design and evaluation of KV-Direct, a high-performance in-memory key-value store. In the long history of computer system design, KV-Direct is another exercise in leveraging reconfigurable hardware to accelerate important workloads. KV-Direct is able to achieve exceptional performance by carefully designing hardware and software to eliminate bottlenecks in the system and achieve performance close to the physical limits of the underlying hardware.

Chapter 6 Acceleration of SocksDirect Communication Primitives

6.1 Introduction

The theme of this chapter is the acceleration of operating system communication primitives, as shown in Figure 6.1.

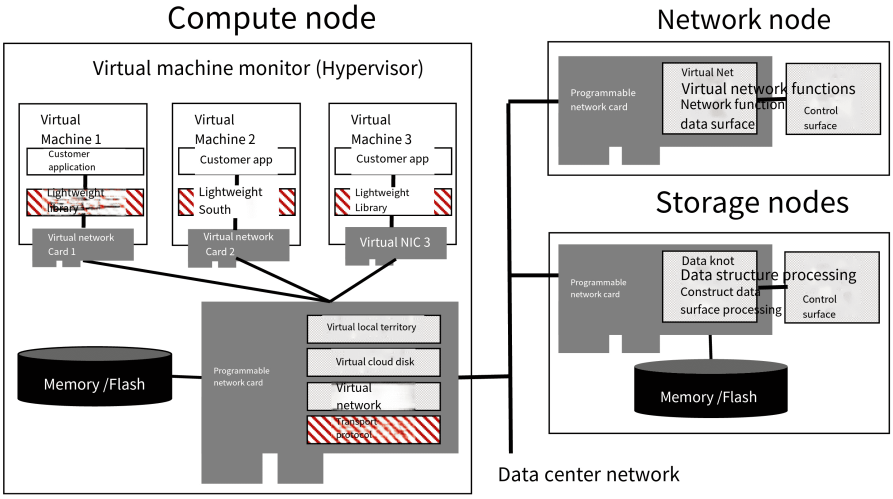


Figure 6.1 The theme of this chapter: acceleration of operating system communication primitives, marked with a bold italic line background.

As the last research work introduced in this paper, we have implemented a user-space socket communication library that is compatible with existing applications, using shared memory and RDMA as the communication methods between processes on the same machine and between different hosts, respectively. As a supplement, based on the ClickNP programming framework and KV-Direct data structure processing service proposed in previous chapters, we have implemented a scalable RDMA connection number in the programmable network card.

The Socket API is the most widely used communication primitive in modern applications, typically used for communication between processes, containers, and hosts. Linux sockets can only achieve latency and throughput that are one to two orders of magnitude worse than bare hardware (such as shared memory and RDMA). In recent years, a lot of work has been aimed at improving socket performance. Existing methods either optimize the kernel network protocol stack[249-251], move the TCP/IP protocol stack to user space[16-19,22], or offload the transport layer to RDMA network cards[43-44]. However, all these solutions have limitations in terms of compatibility and performance. Most of them are not fully compatible with Linux sockets in aspects such as process

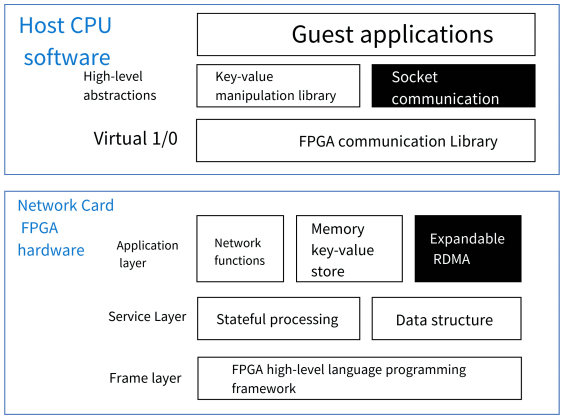


Figure 6.2 The position of this chapter in the programmable network card software and hardware architecture.

fork, event polling, multi-application socket sharing, and intra-host communication. Some of them[17] have isolation issues, not allowing multiple applications to share a network card. Despite these efforts to improve performance, there is still a lot of room for performance improvement. Existing work cannot achieve performance close to bare RDMA and shared memory because they cannot eliminate important overheads such as multi-thread synchronization, buffer management, and memory copying. For example, sockets are shared among multiple threads within a process, so many systems use locks to avoid race conditions.

Recognizing these limitations, this chapter designs SocksDirect, a user-space socket system that achieves compatibility, isolation, and high performance simultaneously.

- Compatibility. Applications can use SocksDirect as a substitute for Linux sockets without any modifications. SocksDirect supports both intra-host and inter-host communication, and behaves correctly during process forks and thread creation. If the remote endpoint does not support SocksDirect, the system will transparently fall back to standard TCP.
- Isolation. Firstly, SocksDirect maintains isolation between applications and containers, i.e., no application can listen to or interfere with connections between other applications, and a malicious program cannot cause erroneous behavior at its connection endpoint. Secondly, SocksDirect can enforce access control policies to prevent unauthorized connections.
- High performance. SocksDirect provides high throughput and low latency, comparable to raw RDMA and shared memory, and can achieve performance scaling across multiple CPU cores.

To achieve high performance, SocksDirect fully utilizes the capabilities of modern

hardware. It uses RDMA for inter-host communication and *shared memory* for intra-host communication. However, converting socket operations into RDMA and shared memory operations is not straightforward. Simple solutions may violate compatibility or leave a lot of performance on the table. For example, after a socket send() returns, the application may overwrite the buffer. However, RDMA send operations require write-protecting the buffer. Existing work[43] either provides a zero-copy API that is incompatible with unmodified applications, or requires the protocol stack to manage internal buffers and copy data from the buffer.

To achieve all three goals simultaneously, it is first necessary to understand how Linux sockets provide compatibility and isolation. Linux sockets provide a Virtual File System (VFS) abstraction to applications. Through this abstraction, application developers can communicate like operating files without delving into the details of network protocols. This abstraction also provides good isolation between applications sharing address and port spaces. However, the VFS abstraction is very complex, and many APIs are inherently unscalable[17,124-125].

Despite the universality and complexity of VFS, many commonly used socket operations are actually quite simple. Therefore, the design principle of this chapter is to optimize for common cases while maintaining compatibility.

To accelerate data transmission while maintaining isolation in connection management, SocksDirect separates the control and data planes[12]. In each host, a *monitor* daemon is introduced as the *control plane* to enforce access control policies, manage address and port resources, dispatch new connections, and establish transport channels between communication endpoints. The *data plane* is handled by a dynamically loaded user-space library libsd, which intercepts function calls to the standard Linux C library. libsd implements the socket API in user space and forwards non-socket related APIs to the kernel. Applications can take advantage of libsd by loading the library using the *LD_PRELOAD* environment variable in Linux.

In SocksDirect, data transmission and event polling are handled directly between peer processes, while connection establishment is delegated to the monitor. This chapter utilizes various techniques to efficiently use hardware and improve system efficiency. Typically, socket connections are shared between threads and processes created by fork. To avoid race conditions when accessing socket metadata and buffers, synchronization is needed. By using a token-based sharing method instead of locking for each operation, SocksDirect eliminates synchronization overhead in common cases. When sending and receiving data from the network card, existing systems allocate buffers for each packet.

To eliminate buffer management overhead, this chapter designs a ring buffer exclusive to each connection, with a copy at both the sender and receiver, and then synchronizes from the sender's ring buffer to the receiver's using RDMA and shared memory. To achieve zero-copy for larger messages, SocksDirect remaps pages using the virtual memory mechanism.

The design of SocksDirect presents numerous challenges. (1) How can a socket be shared among threads and forked processes without locking? (2) How can it scale to accommodate many concurrent connections? (3) How can shared memory and RDMA be utilized efficiently for intra- and inter-host communication?

In both multi-thread and multi-process scenarios, a connection may be shared by multiple senders and receivers. Existing approaches require locking to protect shared queue and metadata. To avoid the overhead of locking, we treat each thread as a separate process. libsd uses thread-specific storage and creates peer-to-peer queues between each pair of communicating threads. To preserve FIFO semantics, we optimize for the common case while preparing for the worst case, and take special care on fork and thread creation.

To efficiently handle many concurrent connections, we need to save memory footprint and improve spatial locality. For each pair of threads, SocksDirect multiplexes socket connections through one message queue. Instead of maintaining a separate buffer for each connection and an event notification queue, we receive events and data from the message queue directly. Observing the event-driven behavior of applications, in normal case the data in queue is fetched by the application in send order. We design carefully to enable fetching from the middle of queue and solve the head-of-line blocking problem.

We leverage different transports to push performance to the limits of underlying hardware. For inter-process and inter-container sockets within a same host, we use shared memory in user space. For sockets among hosts in an RDMA enabled data center, SocksDirect can transparently determine whether the remote endpoint supports SocksDirect. we fall back to kernel TCP socket. We design different queue structures for shared memory and RDMA. We use batched one-sided RDMA write and amortize polling overhead with shared CQ. To remove memory copy for large messages, we use *page remapping* to achieve transparent zero copy. To share a CPU core efficiently among multiple active threads, SocksDirect uses *cooperative multitasking* to remove thread wakeup overhead.

SocksDirect achieves latency and throughput close to the performance of underlying shared memory queues and bare RDMA. In terms of latency, SocksDirect achieves

0.3 microseconds RTT for intra-host sockets, which is 1/35 of Linux, and only 0.05 microseconds higher than bare-metal shared memory queues. For inter-host sockets, SocksDirect achieves 1.7 microseconds RTT between RDMA hosts, almost the same as bare RDMA writes, and 1/17 of Linux. In terms of throughput, a single thread can send 23 M intra-host messages per second (20 times of Linux) or 18 M inter-host (15 times of Linux, 1.4 times of bare RDMA writes). For large messages, through zero-copy, a single connection can saturate the bandwidth of a 100 Gbps network card. The above performance can be linearly scaled with the number of cores. SocksDirect provides significant acceleration for actual applications. For example, the HTTP request latency of Nginx[252] is reduced to 1/5.5, and the latency of the standard RPC library can also be reduced by 50

In summary, this chapter makes the following contributions:

- Analyzes the overhead of Linux sockets.
- Designs and implements SocksDirect, a high-performance user-space socket system that is compatible with Linux and can maintain isolation between applications.
- Supports technologies such as fork, lock-free connection sharing, ring buffer, and zero-copy, which may be useful in many scenarios beyond sockets.
- Evaluations show that SocksDirect can achieve performance comparable to RDMA and shared memory queues.

We evaluate the end-to-end performance of SocksDirect using two categories of applications: *network functions* and *web services*. For a multi-core pipelined network function (NF) chain, a socket application achieves comparable performance with a state-of-the-art NF framework[253]. We also evaluate SocksDirect on a standard web application composed of a load balancer, a web service, and a key-value store. For an HTTP request that involves multi-round-trip key-value store accesses, SocksDirect reduces end-to-end latency by two-thirds.

6.2 Background

6.2.1 Introduction to Linux Sockets

Sockets are the standard communication primitives among applications, containers, and hosts. Figure 6.3 illustrates the pseudocode of a typical server application using socket primitives. First, the server creates a socket file descriptor lfd for listening to ports, receiving new connections, and sets it to non-blocking for asynchronous process-

ing. Then it creates an event file descriptor efd for receiving new connection events and events of data transmission on each connection.

Next, it enters the event loop. For each received event, if it is a new connection, it calls accept to accept it and adds it to event monitoring. If data arrives on an existing connection, it receives all data on that connection (because of the size limit of the receive buffer, a single recv may not receive all data). If the peer is ready to receive (i.e., the receive buffer has free space), it sends out the data to be sent.

The translation of your document is as follows:

```
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);
    }
  }
}
```

Figure 6.3 Pseudocode of a typical socket server application, illustrating the most crucial socket operations. Socket connections are identified by the integer *FD* (file descriptor), a FIFO byte stream channel. Linux employs a readiness-driven I/O multiplexing model, where the operating system informs the application which file descriptors are ready to receive or send, and then the application can prepare the buffer and execute socket operations.

TCP sockets in contemporary operating systems typically serve three functions: (1) addressing, locating, and connecting to another application; (2) providing a reliable and ordered communication channel, identified by the integer *file descriptor*; (3) polling events from multiple channels, such as poll and epoll. Most Linux applications utilize

a readiness-driven I/O multiplexing model, that is, the operating system informs the application which file descriptors are ready to receive or send, and then the application can prepare the buffer and initiate receive or send operations.

6.2.2 Overhead in Linux Sockets

Modern data center networks have microsecond latency and tens of Gbps throughput. However, traditional Linux sockets are implemented in the operating system kernel space with shared data structures, making sockets a well-known bottleneck for communication-intensive applications running on multiple hosts[6]. In addition to inter-host communication, microservices and containers on the same host often communicate with each other, making intra-host socket communication increasingly important in the cloud era. Under stress testing, applications such as Nginx [254], Memcached [205] and Redis [255] consume 50% to 90% of CPU time in the kernel, mainly for handling TCP socket operations[17].

Conceptually, the Linux network protocol stack consists of three layers. First, the VFS layer provides the socket API (such as *connect*, *send* and *epoll*) to applications. Socket connections are bidirectional, reliable, and ordered pipelines, identified by the integer *file descriptor*. Second, the traditional TCP/IP transport layer provides I/O multiplexing, congestion control, packet loss recovery, routing, and Quality of Service (QoS) functions. Third, the network card layer communicates with network card hardware (or virtual loopback interface for intra-host sockets) to send and receive packets. It is well known that the VFS layer contributes a large part of the cost in the network protocol stack[124-125]. This can be verified by a simple experiment: the latency and throughput of a Linux TCP socket between two processes in a host are only slightly worse than those of a pipe, FIFO, and Unix domain socket. (In Table 6.2, the Linux TCP latency is 11 μs, throughput is 0.9 M op/s, and the latency of pipe, FIFO, and Unix domain socket is 8 9 μs, throughput is 0.9 1.2 M op/s.) Pipes, FIFOs, and Unix domain sockets bypass the transport and network card layers, but their performance is still unsatisfactory.

The seminal work of Clark et al.[124] categorized socket overhead into per-packet and per-byte costs. In contemporary protocol stacks, due to the substantial cost of connection establishment[17,249], we propose a new type of cost: per-connection cost. Given that each socket operation incurs a certain cost at the VFS layer, irrespective of the number of packets it manages (some operations, such as `dup2`, do not manage packets at all), we introduce another new type of cost: per-operation cost. Subsequently, we

Table 6.1 Overhead of Linux sockets.

Type	Overhead	Solution in this chapter
Per operation	Kernel crossing (system call)	User-space library (§6.3)
Per operation	Socket file descriptor lock for concurrent threads and processes	Token-based socket sharing (§6.4.1)
Per packet	Transport layer protocol (TCP/IP)	Using RDMA or shared memory (§6.4.2)
Per packet	Buffer management	New ring buffer design (§6.4.2)
Per packet	I/O multiplexing	Using RDMA or shared memory (§6.4.2)
Per packet	Interrupt handling	Event notification (§6.4.4)
Per packet	Process wakeup	Event notification (§6.4.4)
Per byte	Data copying	Page remapping (§6.4.3)
Per connection	Kernel file descriptor allocation	File descriptor remapping table (§6.4.5)
Per connection	TCB lock management	Dispatch to libsd (§6.4.5)
Per connection	Dispatch new connection	Daemon process (§6.4.5)

will classify socket overhead into four types: per operation, per packet, per byte, and per connection.

1. Overhead per operation

Kernel crossing. Traditionally, the socket API is implemented in the kernel, so a kernel crossing (i.e., system call) is required for each socket operation. Worse, to prevent Meltdown[256] attacks, the Kernel Page Table Isolation (KPTI) patch[257] makes kernel crossing 4 times more expensive, as shown in Table 6.2 (before the KPTI patch, kernel crossing required 50 ns, and after KPTI, it required 200 ns). The goal of this chapter is to bypass the kernel without compromising security (§6.3).

Socket file descriptor lock. Many applications are multithreaded for two reasons. First, unlike FreeBSD, the asynchronous interface for reading and writing disk files in Linux cannot take advantage of operating system cache and buffer, so applications continue to use multithreading and synchronous interfaces[258]. Second, many web application frameworks prefer to handle each user request with a synchronous programming model because it is easier to write and debug[6]. Multiple threads in a process share socket connections. In addition, after a process forks, the parent process and child process share existing sockets. Sockets can also be passed to another process through Unix domain sockets. To protect concurrent operations, the Linux kernel acquires a lock for each socket operation[125,249-250]. Table 6.2 shows that even without multicore contention, the latency of a shared memory queue protected by atomic operations is up to 4 times that of a lock-free queue, and the throughput is only 22The goal of this chapter is to minimize synchronization overhead as much as possible by optimizing common cases

and removing synchronization operations from common socket operations (§6.4.1).

Intra-host communication. Most existing approaches for intra-host socket either use kernel network stack or network card loopback. The kernel network stack has evolved to become quite complicated over the years[251], which is an overkill for intra-host communication.

Arrakis uses the network card to forward packets from one application to another. As shown in Table 6.2, the hairpin latency from CPU to network card is still 25x higher than inter-core cache migration delay (~30 ns). The throughput is also limited by Memory-Mapped I/O (MMIO) doorbell latency and PCIe bandwidth[135,259].

We aim to leverage user-space shared memory for intra-host socket communication.

The main challenge for leveraging RDMA for inter-host socket communication is to bridge the semantics of socket and RDMA[70]. For example, RDMA preserves messages boundaries while TCP does not. For I/O multiplexing, RDMA provides a completion notification model while event polling in Linux socket requires a readiness model[250]. Further, one-sided and two-sided RDMA verbs have different efficiency and overheads[46,260].

We aim to use RDMA efficiently for inter-host socket communication, while falling back to TCP transparently in case of non-RDMA peers.

Many concurrent connections. Internet facing applications often need to serve millions of concurrent connections efficiently[11,17,249]. Moreover, it is also common for two backend applications to create a large number of connections between them, where each connection handles a concurrent task[170,261-262]. In Linux, a socket connection has dedicated send and receive buffers, each is at least one page (4 KB) in size[263]. With millions of concurrent connections, the socket buffers can consume gigabytes of memory, most of which is empty. Random accesses to a large number of buffers also cause CPU cache misses and TLB misses. Similar issue exists in RDMA network cards with limited on-chip memory for caching connection states[260,264].

The translation of your text is as follows:

We aim to minimize memory cache misses per data transmission by multiplexing socket connections.

2. Overhead per packet

Transport Protocol (TCP / IP). Traditionally, TCP/IP has been the de facto standard for data center transport protocols. The processing of TCP/IP protocol, congestion con-

Table 6.2 Round-trip latency and single-core throughput operations (test platform settings in §6.5.1). Unless otherwise specified, the message size is 8 bytes.

Operation	Latency (μs)	Throughput (M operations per second)
Inter-core cache migration	0.03	50
Polling 32 empty queues	0.04	24
System call (before KPTI)	0.05	21
Spinlock (no contention)	0.10	10
Allocation and release of buffer	0.13	7.7
Spinlock (with contention)	0.20	5
Lock-free shared memory queue	0.25	27
Intra-host SocksDirect	0.30	22
System call (after KPTI)	0.20	5.0
Copying 1 memory page (4 KiB)	0.40	5.0
Cooperative context switch	0.52	2.0
Mapping a memory page (4 KiB)	0.78	1.3
Intra-host communication via network card	0.95	1.0
Atomic shared memory queue	1.0	6.1
Mapping 32 memory pages (128 KiB)	1.2	0.8
Opening a socket file descriptor	1.6	0.6
Unilateral RDMA write operation	1.6	13
Bilateral RDMA send / receive operation	1.6	8
Inter-host SocksDirect	1.7	8
Process awakening	2.8∼5.5	0.2∼0.4
Linux pipe / FIFO	8	1.2
Unix domain sockets in Linux	9	0.9
Inter-host Linux TCP sockets	11	0.9
Copying 32 memory pages (128 KiB)	13	0.08
Inter-host Linux TCP sockets	30	0.3

trol, and packet loss recovery consume CPU on every sent and received packet. Furthermore, packet loss detection, rate-based congestion control, and the TCP state machine employ timers, making it challenging to achieve microsecond granularity and low overhead[17]. Fortunately, in recent years, we have seen large-scale deployment of RDMA in many data centers[41,265-266]. RDMA offloads the transport protocol to the RDMA network card, providing a hardware-based transport layer equivalent to TCP/IP. For inter-host sockets, the aim of this chapter is to leverage the high throughput, low latency, and near-zero CPU overhead of the RDMA hardware transport layer (§1). For intra-host sockets, the aim of this chapter is to completely bypass the transport layer.

Buffer Management. Traditionally, the CPU sends and receives packets from the network card through a *ring buffer*. The ring buffer comprises a fixed number of fixed-length metadata entries. Each entry points to a buffer that stores the packet payload. To send or receive a packet, it is necessary to allocate and release buffers. Table 6.2 shows the cost of the ring buffer. Additionally, to ensure that packets of MTU size can be received, each receive buffer should be at least the size of an MTU. However,

many packets are smaller than the MTU [267], so internal fragmentation reduces memory utilization. Although modern network cards support LSO and LRO [268] to batch process multiple packets, the aim of this chapter is to completely eliminate the overhead of buffer management (§6.4.2).

I/O Multiplexing. For traditional network cards, packets from different connections are usually mixed in a ring buffer, so the network protocol stack needs to classify the packets into the corresponding socket buffers. Modern network cards support packet steering[112], which can map specific connections to dedicated ring buffers used by high-performance socket systems[17,22,249]. This chapter utilizes similar functionality in RDMA network cards to demultiplex received packets into ring buffers dedicated to each connection.

Interrupt Handling. The Linux network protocol stack is divided into system call and interrupt contexts because it handles events from both applications and hardware devices. For instance, when an application calls send, the network protocol stack sends the packet in the process context (if the window allows). When the network card receives the packet, it sends an interrupt to the CPU, and then the network protocol stack processes the received packet in the interrupt context. The ACK clocking mechanism in TCP congestion control[264] requires timely handling of interrupts and timers. The interrupt context is not necessarily on the same core as the application, leading to a decrease in CPU core locality. However, RDMA network card hardware implements packet processing that requires precise timing, so the host CPU no longer needs to handle most of the data plane interrupts.

Process Awakening. When a process calls a remote procedure call (RPC) and waits for a reply, should the CPU switch to other ready-to-run processes? The answer in Linux is yes, and this process switching wake-sleep process takes 3 to 5 μs, as shown in Table 6.2. Within the round-trip time of RPC within the host, two process awakenings contribute more than half of the latency. For host-to-host RPC via RDMA, the round-trip latency of small messages smaller than the MTU size on the network is even lower than the process awakening latency. For this reason, many distributed systems and user-space protocol stacks use polling to avoid awakening overhead. However, simple polling methods consume a CPU core for each thread and cannot scale to a large number of threads. To hide microsecond-level RPC latency[6], cooperative context switching through sched_yield is much faster than process awakening. The goal of this chapter is to efficiently share cores among multiple threads (§6.4.4).

Container Networking. Many container deployments use isolated network namespaces, and these containers communicate through a virtual overlay network. In Linux, a virtual switch[172] forwards packets between the host network card and the virtual network card in the container. This architecture incurs overhead from multiple context switches and memory copies on each packet, and the virtual switch becomes a bottleneck[269]. Slim [270] reduces three kernel round trips to one. Several recent works[25,121,271-272] delegate all operations to a virtual switch running as a daemon, thus increasing latency and CPU cost on the data path. The solution in this chapter is a centralized control plane and a distributed data plane (§6.4.5).

3. Per-byte Overhead

Payload Copying. In most socket systems, the semantics of send and recv result in memory copying between the application and the network protocol stack. For non-blocking send, the system needs to copy data out of the buffer because the application may overwrite the buffer immediately after send returns. Simply eliminating the copy may violate the correctness of the application. Zero-copy recv is even more challenging than send. Linux provides a readiness-based event model, i.e., the application knows about incoming data (e.g., through epoll) and then calls recv, so data received by the network card but not delivered to the application must be stored in the system buffer. Because recv allows the application to provide any buffer as the data target, the system needs to copy data from the system to the application buffer. The goal of this chapter is to implement zero-copy for larger payload transfers in standard socket applications (§6.4.3).

4. Per-connection Overhead

Kernel File Descriptor Allocation. In Linux, each socket connection is a file in VFS, thus requiring the allocation of integer file descriptors and *inode*. The challenge with user-space sockets is that many APIs (such as open, close, and epoll) support both socket and non-socket file descriptors (such as files and devices), so socket file descriptors must be distinguished from other file descriptors. Linux-compatible sockets in user space[22,43] typically open a file in the kernel to obtain a virtual file descriptor for each socket, so they still require kernel file descriptor allocation. LOS [25] divides the file descriptor space into user and kernel parts, but violates the Linux property of allocating the smallest available file descriptor. However, many applications, such as Redis [78] and Memcached [26], rely on this property. The goal of this chapter is to bypass kernel socket file descriptor allocation while maintaining compatibility (§6.4.5).

Optimization of Kernel Network Protocol Stack: The first type of work is the optimization of the kernel TCP/IP protocol stack. FastSocket[249], Affinity-Accept[273], FlexSC[274], and zero-copy sockets[275-277] achieve good compatibility and isolation.

MegaPipe[250] and StackMap[251] propose new APIs to implement zero-copy and improved I/O multiplexing, at the cost of requiring modifications to the application. However, a significant amount of kernel overhead still exists. The challenge of supporting zero-copy is socket semantics.

User-space TCP/IP Protocol Stack: The second type of work completely bypasses the kernel TCP/IP protocol stack and implements TCP/IP in user-space. In this category, IX[11] and Arrakis[12] are new operating system architectures that use virtualization to ensure security and isolation. IX uses LwIP[278] to implement TCP/IP in user-space, while using the kernel to forward each packet to achieve performance isolation and QoS. In contrast, Arrakis offloads QoS to the network card, thus bypassing the kernel on the data plane. These works use the network card to forward packets between applications on the same host. As shown in Table 6.2, the round-trip (hairpin) latency from the CPU to the network card is much higher than the cache migration latency between cores. The throughput is also limited by the doorbell latency of memory-mapped I/O (MMIO) and PCIe bandwidth[135,259].

In addition to these new operating system architectures, many user-space sockets on Linux use high-performance packet I/O frameworks, such as Netmap[13], Intel DPDK[14], and PF_RING[15], to directly access network card queues in user-space. SandStorm[16], mTCP[17], Seastar[18], and F-Stack[19] propose new APIs, thus requiring modifications to the application. Most API changes aim to support zero-copy, while the standard API still copies data. FaSST[20] and eRPC[21] provide an RPC API instead of sockets. LibVMA[22], OpenOnload[23], DBL[24], LOS[25], and TAS[279] comply with the standard socket API.

The user-space TCP/IP protocol stack offers superior performance compared to Linux, but it still falls short when compared to RDMA and shared memory. A significant reason for this is that existing works do not support the sharing of sockets between threads and processes, which results in compatibility issues in fork and container hot migration, as well as overhead from multi-threaded locks.

Firstly, in LibVMA and RSocket, after a process forks, for sockets created by the parent process before the fork, the child process either takes ownership of all existing sockets or cannot access any sockets (i.e., these sockets still belong to the parent process). There is no way to independently control the ownership of each socket. However,

many web services[252,280-283] and key-value stores[26] have a master process to accept new connections from the listening socket, then it may fork a child process to handle requests, and the child process needs to access the socket of the new connection. At the same time, the parent process still needs to accept new connections through the listening socket. This makes such web services unable to work properly. A more tricky situation is that the parent process and the child process can simultaneously write to the log server through the existing socket.

Secondly, multi-threading is common in applications. Applications bear the risk of race conditions in socket operations, or must adopt socket file descriptor locks for each operation. The latter method ensures correctness, but even without contention between locks, locks can impair performance.

Offloading the Transport Layer to the Network Card: To reduce the overhead of operating system communication primitives, a series of works offload part of the socket system to the network card hardware. The TCP Offload Engine (TOE)[39] offloads part or all of the TCP/IP protocol stack to the network card, but due to the rapid growth of general processor performance according to Moore's Law, these dedicated hardware have limited performance advantages and only achieve success in dedicated fields, such as iSCSI HBA storage cards[284] and stateless offloads (such as checksum, Receive Side Scaling (RSS), Large Send Offload (LSO), Large Receive Offload (LRO)[268]). In recent years, due to hardware trends and application demands in data centers, the story of *stateful* offloads has begun to revive[40]. Therefore, RDMA[35] is widely used in production data centers[41]. RDMA provides two types of abstractions: one-sided primitives for reading and writing remote shared memory, and two-sided primitives with socket-like send-receive semantics. Compared with the software-based TCP/IP network protocol stack, RDMA uses hardware offloading to provide ultra-low latency and near-zero CPU overhead. To enable socket applications to use RDMA, RSocket[43], SDP[44], and UNH EXS[45] convert socket operations into two-sided RDMA primitives. They have similar designs, with RSocket being the most actively developed and the de facto standard for socket-to-RDMA conversion. FreeFlow[272] uses RDMA network cards to provide container overlay networks, using shared memory for intra-host communication and RDMA for inter-host communication. To implement RDMA virtualization, FreeFlow is essentially a microkernel architecture, with control plane and data plane operations handled by a user-space virtual switch. FreeFlow uses RSocket to convert sockets to RDMA.

However, due to the mismatch between RDMA and socket abstractions, these

works have limitations. In terms of compatibility, first, they lack support for several important APIs, such as *epoll*, thus they are incompatible with many applications, including Nginx, Memcached, Redis, etc. This is because RDMA only provides transport functionality, while *epoll* is a file abstraction integrated with OS event notification. Second, RDMA QP does not support *fork* and container hot migration[272], hence RSocket has the same problem. Third, since RSocket uses RDMA as the network packet format, it cannot connect to regular TCP/IP peers. This is a deployment challenge, as all hosts and applications in a distributed system must switch to RSocket simultaneously. The goal of this chapter is to transparently detect whether the remote end supports Rsocket, and if not, fall back to TCP/IP. In terms of performance, they cannot eliminate payload copy, socket file descriptor lock, buffer management, process wake-up, and per-connection overhead. For example, RSocket allocates buffers and copies payloads at both the sender and receiver. Similar to Arrakis, RSocket uses the network card for intra-host communication, leading to performance bottlenecks.

6.3 Architecture Overview

To simplify deployment and development[149], and to eliminate kernel crossing overhead, this chapter implements SocksDirect in user space rather than in the kernel. To use SocksDirect, the application loads the user-space library libsd by setting the LD_PRELOAD environment variable. libsd intercepts all Linux APIs related to file descriptor operations in the standard C library (glibc), implementing socket functionality in user space. From a security perspective, because libsd resides in the application address space, its behavior is untrusted. For example, a malicious program might directly write arbitrary messages into the RDMA QP, bypassing the security checks in the libsd library. In addition, as shown in Table 6.3, although most socket operations can be implemented between the caller's local or both ends of the connection, many socket operations require centralized coordination. For example, the TCP port number is a global resource that needs to be centrally allocated[249,272]. Therefore, a trusted component outside of libsd is needed to enforce access control and manage global resources.

For this purpose, a *monitor* daemon (hereinafter referred to as *monitor*) is designed on each host to coordinate control plane operations, such as connection creation. To ensure isolation, we treat all applications and monitors as a distributed system that does not share data structures, using message communication as the only communication

Initialization		Connection Management	
API	Category	API	Category
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
Data Transfer		**Process Management**	
API	Category	API	Category
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

Table 6.3 Main Linux APIs related to sockets and intercepted by libsd. Categories include Local, Peer-to-Peer (P2P), and Non-partitionable (NoPart). *Italic* APIs indicate other uses of the operating system besides sockets. Bold APIs are called more frequently than other APIs, and therefore are more worth optimizing.

mechanism. On each host, all applications loaded with libsd must establish a shared memory (shared memory) queue with the host's monitor, thus forming a control plane. On the data plane, applications build end-to-end (peer-to-peer) queues for direct communication, thereby alleviating the burden of the monitor daemon. The monitor is a single-threaded user-mode program that polls messages from the end-to-end queues of all applications. Figure 6.4 shows the architecture of SocksDirect.

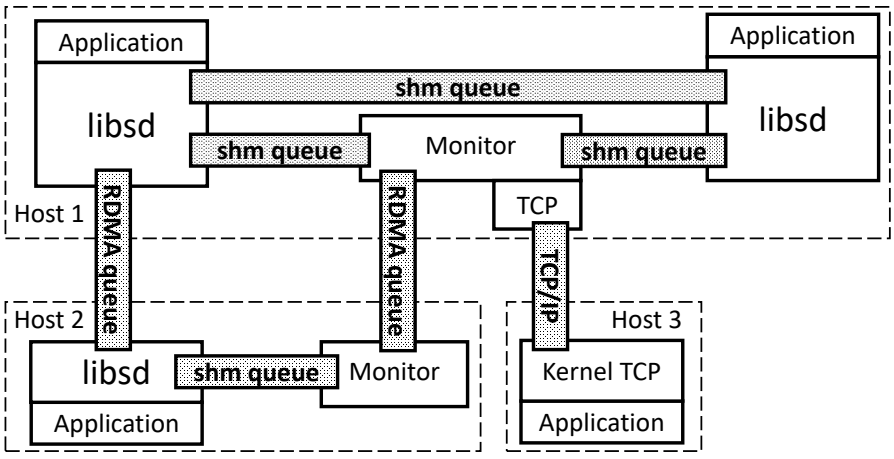


Figure 6.4 The architecture of SocksDirect. Hosts 1 and 2 have RDMA capabilities, while Host 3 does not.

To achieve low latency and high throughput, SocksDirect uses shared memory

for intra-host communication and RDMA for inter-host communication. Each socket connection is mapped to a shared memory queue or RDMA QP. The shared memory or RDMA QP is marked by a unique token, so other non-privileged processes cannot access it. Socket send operations are converted into shared memory or RDMA write operations on the socket buffer at the remote endpoint.

For *intra-host* communication, the initiator first sends a request to the local monitor, and then the monitor establishes a shared memory queue between the two applications (possibly in different containers). After this, the two applications can communicate directly.

For *inter-host* communication, the monitors of both hosts are involved. When an application connects to a remote host, its local monitor establishes a regular TCP connection and detects whether the remote host supports SocksDirect and RDMA. If both are supported, an RDMA queue is established between the two monitors, so that future socket connections between the two hosts can be created faster. The monitor at the remote end schedules the connection with the target and helps the two applications establish an RDMA queue, as between Hosts 1 and 2 in Figure 6.4. If the remote host does not support SocksDirect, it will continue to use the TCP connection, as between Hosts 1 and 3 in Figure 6.4. The detailed connection management protocol is introduced in Section 6.4.5.

To ensure thread safety and avoid locks, as well as support fork and container hot migration, this chapter optimizes for the common case where only one pair of send and receive threads are active, while ensuring correctness in all cases (Section 6.4.1). To eliminate buffer management overhead, a ring buffer is designed, where each inter-host message only requires one RDMA write operation, and each intra-host message requires one cache migration (Section 6.4.2). This chapter further designs a zero-copy mechanism that can eliminate data copying of larger messages at the sending and receiving ends (Section 6.4.3). Finally, Section 6.4.4 provides an event notification mechanism.

As shown in Figure 6.5, the libsd runtime library consists of four layers: API encapsulation, Virtual File System (VFS), queue, and transmission. The API encapsulation layer uses a *file descriptor remapping table* to distinguish between socket file descriptors and kernel file descriptors (such as files and devices), implements socket functions in user space, and forwards other system calls to the kernel through the standard C library (glibc). The Virtual File System layer implements functions such as connection creation and closure, event polling and notification, multi-process shared sockets, fork, container migration, etc., and is the most complex layer. Next to the Virtual File System

is the signal layer, which is responsible for receiving events from the operating system and communicating with the monitor and the other end. The next layer is the lock-free queue based on the ring buffer. The bottom layer is the transmission layer, implemented with Shared Memory (SHM) or RDMA.

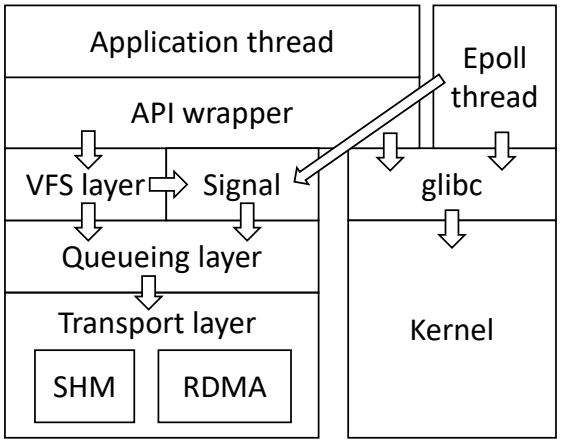


Figure 6.5 The architecture of libsd runtime library.

6.4 System Design

6.4.1 Token-based Socket Sharing

Most socket systems maintain a lock for each file descriptor to enable socket sharing among threads and processes. Previous work[125,285] has shown that many socket operations are not commutative and synchronization cannot always be avoided. This chapter takes advantage of the fact that shared memory message passing is much cheaper than locks[271], and uses message passing as the only synchronization mechanism.

Logically, a socket consists of two FIFO *queues* in opposite transmission directions, each with multiple concurrent senders and receivers. The system design goal is to maximize general case performance while maintaining FIFO semantics. This paper observes the following two characteristics of applications: first, high-performance applications rarely fork and create threads due to high costs. Second, it is not common for several processes to concurrently send or receive from a shared socket, as the byte stream semantics of sockets make it difficult to avoid receiving partial messages. Applications that need to send or receive simultaneously usually use a message broker[77,286-287] instead of directly sharing sockets. The common case of inter-process socket sharing is that applications implicitly migrate sockets from one process to another, such as offloading transactions from the main process to a worker process.

The solution in this chapter is that each *socket queue* (a transmission direction of a socket) has a *send token* and a *receive token*. Each token is held by an *active thread*,

which has the authority to send or receive. Therefore, at any point in time, there is only one active sender thread and one active receiver thread. The socket queue is shared between threads and processes, allowing concurrent lock-free access from a sender and a receiver (details will be discussed in section 6.4.2). When another thread wants to send or receive, it should request to *take over* the token.

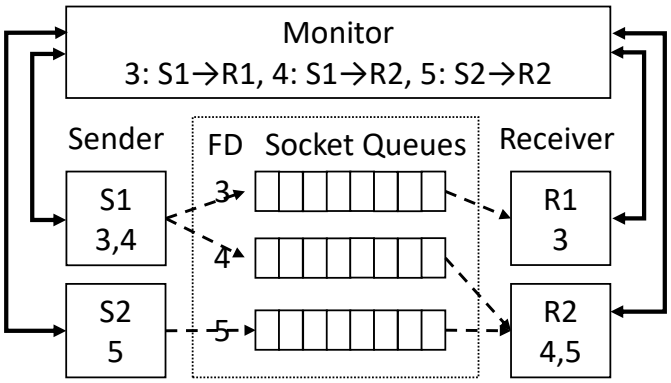


Figure 6.6 Two sender threads and two receiver threads share a token-based socket. The dashed arrows represent the active sender and receiver for each socket. Each thread tracks its active sockets and communicates with the monitor through an exclusive queue.

Detailed information for each type of operation is as follows: a) data transmission (send and recv), b) adding new senders and receivers (fork and thread creation), c) container hot migration, and d) connection closure.

1. Send/Receive Operations

When a thread does not have a send token but wants to send through a socket, the non-active thread needs to *take over* the token. If a direct communication channel is created between the non-active thread and the active thread, it requires point-to-point queues, the number of which is the square of the number of threads, or a shared queue with locks. To avoid these two overheads, a monitor is used as a proxy during the *takeover* process. Since takeover is an infrequent operation, the monitor generally does not become a bottleneck. This message passing design also has the following advantages: the sender process can be located on different hosts, which is very useful in container hot migration.

The takeover process is as follows: the non-active sender sends a *takeover* command to the monitor through the shared memory queue. The monitor polls messages from various shared memory queues, adds the sender to the waiting list, and proxies the command to the current active sender. When the active sender receives the request, it sends the send token to the monitor. The monitor grants the token to the first non-active sender in the waiting list and updates the active sender. The non-active sender can send after receiving the token. This mechanism is deadlock-free because at least one sender

or monitor holds the send token. It is also starvation-free because each sender can appear in the waiting list at most once and is served in FIFO order. The takeover process on the receiver side is similar.

The takeover process requires 0.6 microseconds, so if multiple processes send concurrently through the same socket, the total throughput may drop to 1.6 M operations per second. However, if we simply use locks, the throughput in the usual case will drop to 5 M operations per second, far lower than the 27 M operations per second throughput that can be achieved by token-based socket sharing.

2. Fork, Exec and Thread Creation

Socket data sharing. The main challenge is to share socket metadata, buffers, and the underlying transport layer after fork and exec. After fork, the memory space becomes copy-on-write and is erased after exec, but the socket file descriptor still needs to be available. SocksDirect uses shared memory to store socket metadata and buffers, so the data is still shared after fork. To attach shared memory after exec, libsd connects to the monitor to get the shared memory key of its parent process. After fork, because the parent process cannot see the sockets created by the child process, the child process creates a new shared memory to store the metadata and buffers of the new socket.

Next, consider the underlying transport layer mechanism. The shared memory-based transport layer does not require special handling, as the shared memory created before fork / exec is still shared after fork / exec. However, there are problems with RDMA after fork / exec, because the DMA memory area is not in shared memory. They become copy-on-write after fork, and the network card still DMAs from the original physical page, so the child process cannot use the existing RDMA resources. After exec, the entire RDMA context will be cleared. The solution in this chapter is to let the child process reinitialize the RDMA resources (PD, MR, etc.) after fork / exec. When the child process uses a socket created before fork, it asks the monitor to re-establish the RDMA QP with the remote endpoint. Therefore, the peer process may see two or more QPs of a socket, but they link to the only copy of the socket metadata and buffer. In the next section (§6.4.2), we will see that we only use the RDMA one-sided write primitive, so using any QP is equivalent. Figure 6.7 shows an example of fork.

File descriptor space sharing. Unlike socket data, the file descriptor space becomes copy-on-write after fork: file descriptors created before fork are shared, but new file descriptors are exclusive to the creating process. Therefore, just keep the file descriptor remapping table in heap memory, taking advantage of the operating system's copy-on-write mechanism after fork. To restore the file descriptor remapping table af-

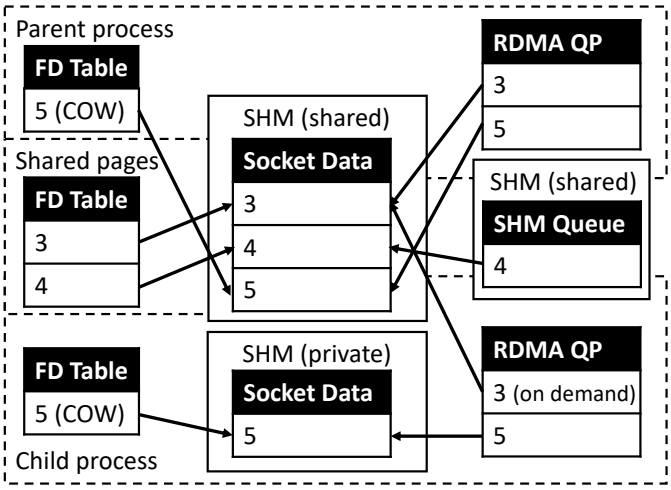


Figure 6.7 Memory layout after fork. File descriptors 3 and 4 were created before fork and are therefore shared. After fork, the parent process and the child process each create a new file descriptor 5, which is copied on write in the file descriptor table. The socket metadata and buffers of file descriptors 3 and 4 are in shared memory and are therefore shared. The child process creates a new shared memory to store the socket metadata and buffers of file descriptor 5, which will be shared with the child process when it forks again. RDMA QP is in private memory, while the shared memory queue is shared.

ter exec, it is copied to shared memory before exec and copied back to the new process during libsd initialization.

Security. Security is an issue because a malicious process may masquerade as a child process of a privileged parent process. To identify parent-child relationships in the monitor, when an application calls fork, clone, or pthread_create, libsd first generates a key for pairing and sends it to the monitor, then calls the original function in *libc*. After fork, the child process creates a new shared memory queue for the monitor and sends the key (the child process inherits the parent memory space, thus knowing the key). Therefore, the monitor can pair the child process with the parent process.

Monitor operations. During fork, exec, or thread creation, the monitor needs to add the new process to the sender and receiver lists of each existing socket to manage takeover operations.

3. Container Live Migration

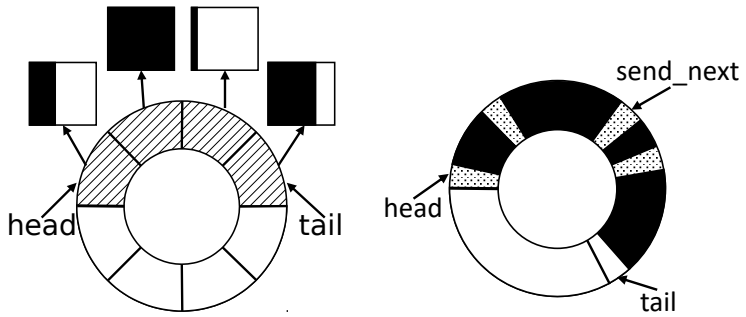
Migrating Remaining Data in Socket Queue. Since libsd runs in the same process as the application, its memory state will be migrated to the new host along with the application. The memory state includes the socket queue, so data in transit (sent but not received) will not be lost. Sockets can only be shared within a container, and all processes in the container are migrated together, so the memory on the original host can be deallocated after migration.

Migration of Monitor State. The monitor tracks information about listening sockets, active threads, and the waiting list for each connection as well as shared memory

keys. During migration, the old monitor dumps the state of the migrated container and sends them to the new monitor.

Establishing New Communication Channels. After migration, all communication channels are obsolete, because shared memory is local on the host, and RDMA does not support live migration[270,272]. First, the migrated container on the new host needs to establish a connection with the local monitor. The local monitor instructs the following process. A host-internal connection between two containers may become host-external, so libsd creates an RDMA connection in this case. A host-external connection between two containers may become host-internal, libsd creates a shared memory connection. Finally, libsd re-establishes the remaining host-external RDMA and host-internal shared memory connections.

6.4.2 Ring Buffer Based on RDMA and Shared Memory



(a) Traditional ring buffer. **(b)** Ring buffer of SocksDirect.

Figure 6.8 Ring buffer data structure. The shaded part is metadata, and the black part is the valid payload.

Traditionally, the network protocol stack uses a ring buffer to send and receive packets from the network card. As shown in Figure 6.8a, the traditional ring buffer consists of a set of fixed-length metadata, each of which points to a fixed-length memory page to store the valid payload. This design leads to memory allocation overhead and internal fragmentation. The reason why traditional network cards use this design is that the number of ring buffers is limited, and multiple connections need to reuse a ring buffer. This design can move the metadata of the valid payload to the metadata queue of each connection without copying the content of the valid payload. Fortunately, the one-sided RDMA write operation (write verb) opens up new design possibilities. The innovation of this chapter is that *each socket connection has its own dedicated ring buffer* and stores packets back-to-back, as shown in Figure 6.8b. The sender determines the offset in the ring buffer (i.e., the *tail* pointer), and then uses a one-sided RDMA write operation to write the packet into the position pointed to by the *tail* pointer in

the remote memory. During the transmission process, the receiving end CPU does not need to do anything. When the receiving end application calls the recv operation, the data is dequeued from the position of the ring buffer pointed to by the *head* pointer. When the *head* pointer moves to coincide with the *tail*, the metadata pointed to by the pointer is empty, so the receiving end's libsd library can detect that the queue is empty. It should be noted that the *head* and *tail* pointers are maintained locally by the receiver and sender, respectively, so there is no need to synchronize these two pointers. The process of transmitting data through shared memory is similar, because both shared memory and RDMA support write operations.

In order to judge whether the ring buffer is full, the sender will keep a *queue credit* count, indicating the number of free bytes in the ring buffer. When the sender enqueues a packet, it will consume credits equal to the size of the enqueued packet. When the credits are insufficient, the sender will block and wait. When the receiver dequeues the packet, it will increase the counter locally, and when the counter exceeds half of the size of the ring buffer, it will write a *credit return flag* into the memory of the sender. The sender regains the queue credit when it detects this flag. Please note that the queue credit mechanism is unrelated to congestion control; the latter is handled by the network card hardware[265].

Both the sender and receiver have a copy of the ring buffer. The above mechanism still requires the sender to allocate memory, because the sender needs a buffer to construct the RDMA message. Secondly, the above mechanism does not support container hot migration, because the remaining data in the RDMA queue that has not been received in time is difficult to migrate. Third, the goal of this chapter is to batch process small messages to improve throughput. For this purpose, this chapter keeps a copy of the ring buffer at both the sender and receiver. The sender writes to its local ring buffer and calls a one-sided RDMA write operation to synchronize the sender's ring buffer with the receiver's. In order to minimize latency on idle links and maximize throughput on busy links, this paper designs an adaptive batching mechanism. libsd creates an RDMA reliable connection (RC) queue pair (QP) for each ring buffer and maintains a counter of RDMA messages. If the counter does not exceed the threshold, an RDMA message is sent for each socket send operation. Otherwise, the message is not sent temporarily, but the first unsent message is marked with *send_next*. After completing the RDMA write operation, libsd sends a message containing all unsent changes (from *send_next* to *tail*) in Figure 6.8b. For shared memory, since cache coherence hardware can automatically perform inter-core synchronization, only one ring buffer shared by

two processes is needed ①.

Consistency between payload and metadata. For shared memory, Intel and AMD's x86 processors provide total store ordering[288-289], which means that each CPU core observes the write order of other CPU cores to be the same. The 8-byte MOV instruction is atomic, so writing the packet header is atomic. Since the sender writes the packet header after the payload, the message read by the receiver is consistent, and no memory barrier instruction is required.

Because RDMA cannot guarantee the write order of different parts of the message[35], it is indeed necessary to ensure that the message is fully arrived before processing the message. Although the write of the message is always sequential in RDMA network cards using go-back-0 or go-back-N packet recovery[70], this is not the case for more advanced network cards with selective retransmission[264**?**]. In libsd, the sender uses the RDMA *write with immediate* operation to generate a completion event at the receiver. The receiver polls the RDMA *completion queue* rather than the ring buffer. RDMA can ensure the cache consistency of the receiver and guarantee that the completion event is later than the data written to the libsd ring buffer.

Amortizing polling overhead. When sockets are not frequently used, polling the ring buffer wastes the receiver's CPU cycles. This chapter uses two techniques to amortize the polling overhead. Firstly, for RDMA queues, the RDMA network card multiplexes event notifications into a single queue. Each thread uses a *shared completion queue* for all RDMA QPs, so it only needs to poll one queue instead of all socket queues.

Secondly, each queue can switch between *polling* and *interrupt* modes. The queue of the monitor is always in polling mode. The receiver of each queue maintains a counter for continuous empty polling. When it exceeds a threshold, the receiver sends a message to the sender, notifying that the queue is entering interrupt mode, and stops polling after a short time. When the sender writes to the queue in interrupt mode, it also notifies the monitor, and the monitor will notify the receiver to resume polling.

6.4.3 Zero-copy

The main challenge of zero-copy is to maintain the semantics of the socket API. Fortunately, virtual memory provides an indirect layer, and many related works have taken advantage of this *page remapping* technique, which can remap physical pages from the sender's virtual address to the receiver's without copying. Linux zero-copy

① Shared memory is a physical page mapped to the user address space of two processes respectively.

sockets[277] only support the sender, and they work by setting the data page to copy-on-write. However, many applications often overwrite the send buffer after calling send, so the copy-on-write mechanism only delays the copy from the time of calling send to the time of the first overwrite. To achieve zero-copy reception, 20 years ago, BSD[275] and Solaris[276] remapped the virtual pages of the application buffer to the physical pages of the operating system buffer. However, as shown in Table 6.2, on modern CPUs, the cost of remapping a page is even higher than copying it due to kernel crossing and TLB refresh overhead. Recently, many high-performance TCP/IP stacks[250-251] and socket-to-RDMA libraries[43-44] provide standard socket APIs and alternative zero-copy APIs, but they have not implemented zero-copy for the standard API. In addition, no existing work supports zero-copy for intra-host sockets.

To enable zero-copy, the network card driver needs to be modified to expose several kernel functions related to page remapping. To amortize the cost of page remapping, libsd only uses zero-copy for send or recv if the payload is at least 16 KiB. Smaller messages are copied directly.

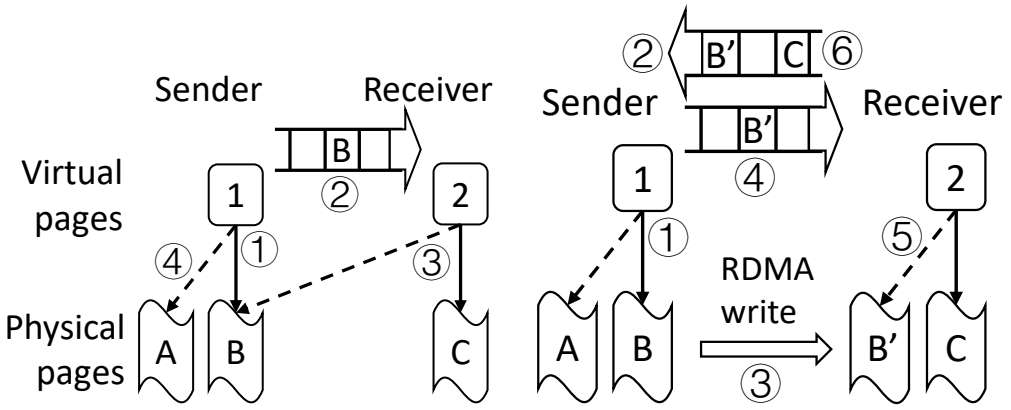
Memory alignment. Page remapping is only effective when the send and receive addresses are page-aligned and the transfer includes entire pages. libsd intercepts malloc and realloc functions and allocates 4 KiB aligned addresses for memory allocation operations larger than 16 KiB, so most buffers will be aligned with page boundaries, and smaller memory allocations are allocated as before to avoid internal fragmentation. If the size of the sent message is not a multiple of 4 KiB, the last piece of data that is not a whole page will be copied during send and recv.

Sometimes, applications need to receive or send data not from the starting address of the allocated buffer. For example, the data is part of an HTTP request, the memory of the HTTP request is aligned, but the data is not aligned due to the presence of the HTTP header. For non-aligned cases, if the application sends directly after receiving without reading and writing the data itself, SocksDirect can also achieve zero-copy message transmission. The method of SocksDirect is that for non-aligned receive buffers, no memory copy is performed by default, but the mapping and offset relationship of the page is recorded. When the application accesses for the first time, the data copy is triggered by the page exception; if the application does not access the data, no copy is needed.

Reduce copy-on-write. When the sender overwrites the buffer after send, the existing design uses copy-on-write. Copying is necessary because the sender may read the unwritten part of the page. Since applications almost always reuse the buffer for

subsequent send operations, copy-on-write is called in most cases, making zero-copy basically useless for the sender. This paper observes that most applications do not write to the send buffer byte by byte. Instead, they overwrite the entire page of the send buffer through recv or memcpy, so there is no need to copy the original data of the page. For memcpy, libsd calls the kernel to remap new pages and disable copy-on-write, and then performs the actual copy. For recv, the old page mapping is replaced by the received page.

Page allocation overhead. The page remapping mechanism requires the kernel to allocate and release memory pages for each zero-copy send and recv. Page allocation in the kernel uses a global lock, which is inefficient. Therefore, libsd manages the available page pool in each process locally. libsd also tracks the source of the received zero-copy pages. When a page is unmapped, if it comes from another process, libsd returns the page to the owner through a message.



(a) Shared memory within the server. 1) Obtain physical pages and set write-on-copy; 2) Send page address via shared memory; 3) Map received pages; 4) (Optional) Remap during write / memcpy / recv on the sender side.

(b) RDMA between servers. 1) Obtain physical pages and set write-on-copy; 2) Get available pages from the page pool; 3) Send data via RDMA; 4) Send page address via RDMA; 5) Map received pages; 6) Return unmapped pages to the page pool.

Figure 6.9 Process of sending zero-copy pages.

Securely sending page addresses via shared memory. For intra-host sockets, libsd sends physical page addresses in the user-space queue messages, as shown in step 2 of Figure 6.9a. For security, SocksDirect must prevent arbitrary page mapping without the sender's permission. To this end, libsd calls the modified network card driver to get the encrypted physical page address of the send buffer and sends the encrypted address to the receiver via the shared memory queue. On the receiver side, libsd calls the kernel to remap the encrypted physical page address to the virtual address of the receive buffer provided by the application.

Zero-copy under RDMA. libsd initializes a fixed page pool on the receiver and

sends the physical addresses of the pages to the sender. The page pool is managed by the sender. On the sender side, libsd pins the virtual to physical page mapping of the send buffer, then allocates pages from the remote receiver's page pool to determine the remote address for RDMA write, as shown in step 2 of Figure 6.9b. On the receiver side, when recv is called, libsd calls the network card driver to map the pages in the pool to the virtual address of the application buffer. After the remapped pages are released (for example, they are overwritten by another recv), libsd returns them to the page pool manager on the sender side (step 6).

6.4.4 Event Notification

Challenge 1: Multiplexing events between the kernel and libsd. Applications poll for events from sockets and other *kernel file descriptors* handled by the Linux kernel. A simple way to poll for kernel events is to call a system call (such as epoll_wait) each time, which incurs high overhead because event polling is a frequent operation that is almost called every time send and receive are called. LOS[25] periodically calls the non-blocking epoll_wait system call with kernel file descriptors, leading to a trade-off between latency and CPU overhead: if called too frequently, the CPU overhead is high; if not called frequently enough, the average latency of notifying kernel events to the application is high. In contrast, libsd creates an *epoll thread* in each process, which calls the epoll_wait system call to poll for kernel events. Whenever the epoll thread receives a kernel event, the application thread will report this event along with user-space socket events.

Challenge 2: Interrupting busy processes. The socket takeover mechanism (Section 1) requires processes to respond to monitor requests. However, processes may execute application code for a long time without calling libsd, and monitor requests cannot be responded to. To solve this problem, this chapter designs a *signal* mechanism analogous to interrupts in the operating system. After the monitor sends a request, it first polls the receive queue for a while, and if there is no reply after a timeout, it sends a Linux *signal* to wake up the corresponding process.

The signal handler registered by libsd first determines whether the process is executing the application or the libsd code. libsd sets and clears flags at the entrance and exit of the library. If the signal handler finds that the process is in libsd, it does nothing, and libsd will handle the event before returning control to the application. Otherwise, the signal handler will immediately process the monitor's message. Since libsd is designed to be fast and non-blocking (all system calls that may cause blocking are called

in the epoll thread), the monitor will receive a response quickly after sending the signal.

Challenge 3: Let multiple threads share the core in time. For blocking socket operations (such as, blocking recv, connect and epoll_wait), libsd first polls the ring buffer once. If the operation is not completed, libsd calls *sched_yield* to give up the CPU and switch to other processes on the same core. As described in Section 6.2.2, context switching in cooperative multitasking only requires 0.4 μs. However, some applications may need to wait a long time to receive a socket message, which leads to frequent wasteful awakenings. For this, libsd counts the consecutive awakenings that do not process any messages, and puts the process to sleep when it reaches a threshold. If the number of libsd idling times reaches a certain threshold, it will put itself to sleep. Before going to sleep, it sends a message to the monitor and all processes (peers) that communicate directly with it, so that it can be awakened later by a signal.

6.4.5 Connection Management

1. File Descriptor Remapping Table

Socket file descriptors and other file descriptors (such as disk files) share a namespace, and Linux always allocates the smallest available file descriptor. To preserve this semantics without allocating virtual file descriptors in the kernel, libsd intercepts all Linux APIs related to file descriptors and maintains a *file descriptor remapping table* to map each application file descriptor to a user-space socket object or kernel file descriptor. When a file descriptor is closed, libsd puts it into a *file descriptor recycling pool*. When allocating a file descriptor, libsd first tries to get a file descriptor from the pool. If the pool is empty, it allocates a new file descriptor by incrementing the *file descriptor allocation counter*. The file descriptor recycling pool and allocation counter are shared among all threads in a process.

2. Connection Establishment

Figure 6.10 shows the connection establishment process.

Binding Address (bind). After creating a socket, the application calls `bind` to allocate an address and port. Since the address and port are globally protected resources, allocation is coordinated by the monitor. As shown in Figure 6.10, libsd sends the request to the monitor. To hide the latency of communicating with the monitor, as an optimization, if the bind request cannot fail (for example, when no port is specified for the client socket), libsd immediately returns to the application.

Listening Port (listen). When the server application is ready to accept connections from clients, it calls `listen` and notifies the monitor. The monitor maintains a list of

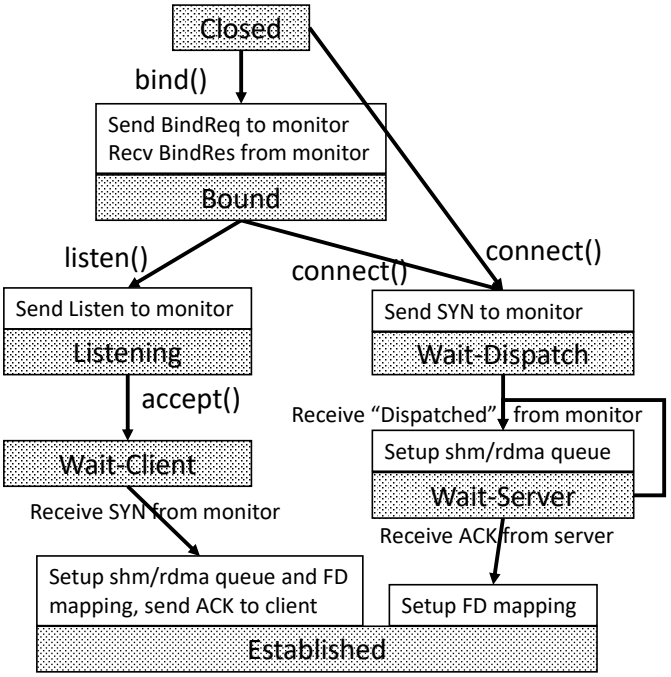


Figure 6.10 State machine of the connection establishment process in libsd.

listening processes on each address and port to dispatch new connections.

Initiating Connection (connect). The client application calls connect and sends a SYN command to listen via the shared memory queue. Now, the monitor needs to dispatch the new connection to the listening application. In Linux, new connection requests are queued in the kernel's *backlog*. Each time the server application calls accept, it accesses the kernel to dequeue from the backlog, which requires synchronization and increases latency. To solve this problem, SocksDirect maintains a backlog for each listener thread of each listening socket. The monitor distributes the SYN to the listener threads in a round-robin manner.

When the listener does not accept new connections, connections dispatched to that listener may cause starvation. SocksDirect uses a *work stealing* method. When the listener calls accept when the backlog is empty, it requests the monitor to steal from others' backlogs. To avoid race conditions between the listener and the monitor, the monitor sends a request to the listener to steal from the backlog.

Establishing Peer-to-Peer Queues. Upon the first communication between the client and server applications, the server monitor can assist them in establishing a direct connection. For within the host, the monitor allocates shared memory queues and sends the shared memory key to the client and server applications. For between hosts, the client and server monitors establish a new RDMA QP, and send the local and remote keys to the corresponding applications. To reduce latency, when the SYN command is dispatched to the backlog of the listener, the monitor establishes a peer-to-peer queue.

However, if the SYN is stolen by another listener, a new queue needs to be established between the client and the new listener, as shown in the Wait-Server state in Figure 6.10.

Final Steps of Connection Establishment. After the server sets up the peer queue, as shown on the left side of Figure 6.10, the server application sends an ACK to the client. The ACK contains the client file descriptor from the SYN request and its allocated server file descriptor. Similar to a TCP handshake, the server application can send data to the queue after sending the ACK. When the client receives the ACK, as shown on the right side of Figure 6.10, it sets up the file descriptor mapping and can start sending data.

3. Compatibility with Regular TCP/IP Peers

To be compatible with peers that do not support SocksDirect and RDMA, SocksDirect needs to transparently detect SocksDirect capabilities and fall back to regular TCP/IP when the peer does not support it. However, regular Linux sockets do not support adding special options to TCP SYN and ACK packets. Due to middleboxes and network reordering, using another port (like LibVMA[22] does) is also unreliable. For this, libsd first uses a kernel raw socket to directly send SYN and ACK packets with special options, and if there are no special options, it falls back to kernel TCP/IP sockets.

On the client side, the monitor sends a TCP SYN packet with special options over the network. If the peer has SocksDirect capabilities, its monitor will receive the special SYN and know that the client has SocksDirect capabilities. Then, the server responds with a SYN + ACK with special options, including credentials for setting up the RDMA connection, so that the two monitors can communicate via RDMA later. If the client or server monitor discovers that the peer is a regular TCP/IP host, it will use Linux's TCP connection repair feature[290] to create an established TCP connection in the kernel. Then the monitor sends the kernel file descriptor to the application via a Unix domain socket, and libsd can use the kernel file descriptor for future socket operations.

A tricky issue is that received packets are delivered to both the raw socket and the kernel network protocol stack, at which point the kernel will reply with an RST packet because this connection does not exist in the kernel. To avoid this behavior, the monitor installs *iptables* rules to filter out such outbound RST packets.

4. Connection Closure

When an application calls close, libsd removes the file descriptor from the remapping table. However, the socket might still be useful because the file descriptor can be shared with other processes, and there might be unsent data in the buffer. libsd keeps track of the reference count for each socket, incrementing it at fork and decrementing it

at close. To ensure that unsent data has been sent to the peer, a handshake between the peers is needed during connection closure, similar to TCP close. Because sockets are bidirectional, close is equivalent to performing shutdown in both the send and receive directions. As shown in Figure 6.11, when an application closes one direction of a connection, it sends a *shutdown message* to the peer. The peer responds with a shutdown message. When libsd has received shutdown messages in both directions, it deletes the socket.

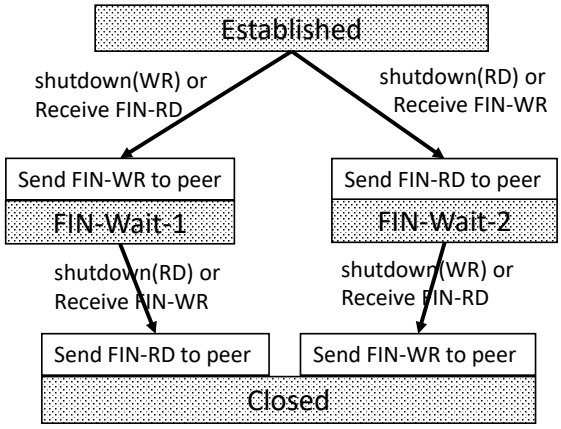


Figure 6.11 State machine for connection closure in libsd.

6.5 System Performance Evaluation

SocksDirect is implemented in three components: a user-space library libsd and a monitoring daemon with 17K lines of C++ code, as well as a kernel module that supports zero-copy. This section evaluates SocksDirect from the following aspects:

Efficiently using shared memory for intra-host sockets. For 8-byte messages, SocksDirect achieves 0.3 μs RTT and a throughput of 23 M messages per second. For large messages, SocksDirect uses zero-copy to achieve 1/13 of Linux's latency and 26x throughput.

Efficiently using RDMA for inter-host sockets. SocksDirect achieves 1.7 μs RTT, close to raw RDMA performance. When zero-copy, a connection can saturate a 100 Gbps link.

Scalable with the number of cores. As the number of cores increases, the throughput can almost linearly scale.

Significantly accelerates unmodified end-to-end applications. For example, SocksDirect reduces the latency of Nginx HTTP requests by 5.5 to 20 times.

6.5.1 Evaluation Methodology

This section evaluates SocksDirect on a server with two Xeon E5-2698 v3 CPUs, 256 GiB memory, and a Mellanox ConnectX-4 network card. The server is interconnected with an Arista 7060CX-32S switch via a 100 Gbps Ethernet interface[291]. Unlike Chapters 4 and 5, this section only uses the commercial card part of the programmable network card, and the commercial card has been upgraded to 100 Gbps, without using FPGA. The server uses Ubuntu 16.04 and Linux 4.15, uses RoCEv2 for the RDMA protocol, and polls the completion queue every 64 messages. Each thread is pinned to a CPU core. Sufficient warm-up tests were conducted before collecting data. For latency, this section reports the average round-trip time of a ping-pong application, with error bars representing the 1% and 99% percentiles. For throughput, one side keeps sending data while the other side keeps receiving data. This section compares Linux, raw RDMA write primitives (write verb), Rsocket [43], and LibVMA [22], which is a user-space TCP/IP protocol stack optimized for Mellanox network cards. We also compared SocksDirect without batching and zero-copy, denoted as "SD (unopt)". This section did not evaluate mTCP [17], because the underlying DPDK library has limited support for Mellanox ConnectX-4 network cards. Due to batching, mTCP has much higher latency than RDMA, and the reported throughput is 1.7 M packets per second [21].

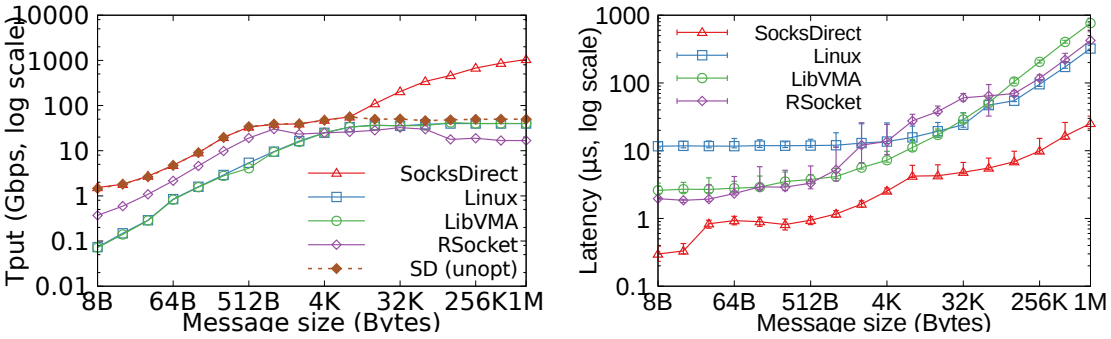
6.5.2 Performance Microbenchmark

1. Latency and Throughput

Figure 6.12 shows the intra-host socket performance between a pair of sender and receiver threads. For 8-byte messages, SocksDirect achieves 0.3 μ s round-trip latency (1/35 of Linux) and 23 M messages per second throughput (20 times of Linux). In comparison, a simple shared memory queue has 0.25 μ s round-trip latency and 27 M throughput, indicating that SocksDirect adds very little overhead. RSocket has 6x latency and 1/4 throughput of SocksDirect, because it uses the network card to forward intra-host packets, which leads to PCIe latency. LibVMA simply uses kernel TCP sockets for intra-host. The one-way latency of SocksDirect is 0.15 μ s, even lower than kernel crossing (0.2 μ s). Kernel-based sockets need to cross the kernel on both the sender and receiver.

Due to memory copying, for 8 KiB messages, the throughput of SocksDirect is only 60

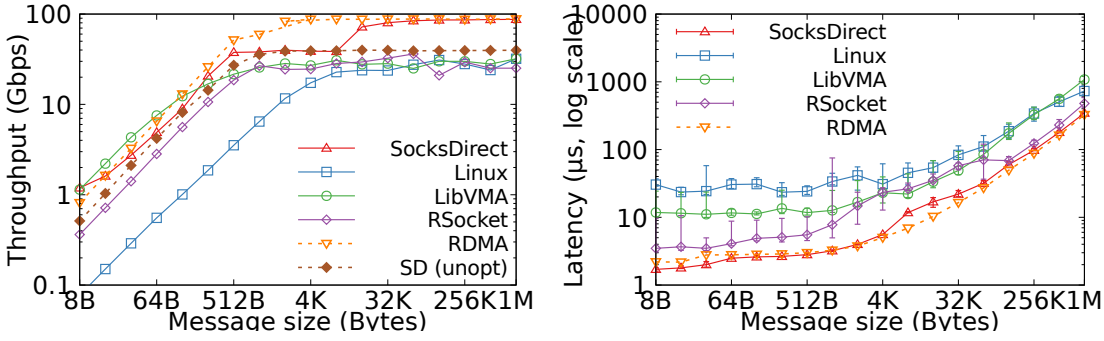
Figure 6.13 shows the inter-host socket performance between a pair of threads. For



(a) Single-machine throughput. **(b)** Single-machine latency.

Figure 6.12 Single-core message performance of intra-machine communication under different message sizes.

8-byte messages, SocksDirect achieves a throughput of 18M messages per second (15 times that of Linux) and a latency of 1.7 microseconds (1/17 of Linux). The throughput and latency are close to the original RDMA write operation (as shown by the dashed line), which does not have socket semantics. Batching does not affect the latency we evaluate, as RDMA write operations are only delayed when the send queue is full, and we only use one message to evaluate latency. Due to batching, the throughput of SocksDirect for 8-byte messages is even higher than RDMA. The message throughput of non-batching SocksDirect is between RSocket and RDMA. LibVMA also uses batching to achieve good performance, but the latency is 7 times that of SocksDirect. For messages smaller than 8 KiB, the throughput of inter-host RDMA is slightly lower than intra-host shared memory because the ring buffer structure is shared. For messages from 512B to 8KiB, and larger messages that do not enable zero-copy, SocksDirect is limited by packet copying, but it is still faster than RSocket and LibVMA due to reduced buffer management overhead. For zero-copy messages (≥ 16 KiB), SocksDirect saturates the network bandwidth, with 3.5 times the throughput of all comparison work and 72



(a) Cross-host throughput. **(b)** Cross-host latency.

Figure 6.13 Single-core message performance of cross-host communication under different message sizes.

2. Latency Decomposition

Table 6.4 explains why the performance of SocksDirect surpasses other systems. Each socket operation in Linux requires a kernel traversal, and all systems except SocksDirect require locking in thread-safe mode. For each packet, SocksDirect saves buffer management overhead and offloads the transport layer and packet processing to the network card. To transmit a packet, SocksDirect uses a one-sided RDMA write operation, requiring only one DMA operation at the sender and receiver respectively. RSocket uses two-sided RDMA, and LibVMA uses a similar packet interface, so the receiver needs to add one DMA operation. LibVMA and RSocket use the network card to forward intra-machine packets, while SocksDirect uses shared memory. The high latency of Linux is mainly due to interrupt handling and process awakening. For larger messages, SocksDirect eliminates data copying, and the overhead of page remapping is significantly lower. RSocket performs better than LibVMA and Linux because it pipelines the data copy operation at the sender, the RDMA send operation, and the data copy operation at the receiver. The connection establishment latency of SocksDirect mainly comes from the initial handshake through the Linux raw socket and the creation of RDMA QP through libibverbs.

3. Multi-core Scalability

SocksDirect has achieved nearly linear scalability for intra-host and inter-host sockets. For intra-host sockets, SocksDirect provides a throughput of 306 M messages per second across 16 pairs of sender and receiver cores, which is 40 times that of Linux and 30 times that of RSocket. LibVMA falls back to Linux for intra-host sockets. Using RDMA as inter-host sockets, SocksDirect achieves a throughput of 276 M messages per second across 16 cores with batching, which is 2.5 times the message throughput of the RDMA network card used in this chapter, and 8 times the throughput of RSocket. Without enabling batching, SocksDirect can only achieve a throughput of 62 M, which is 60

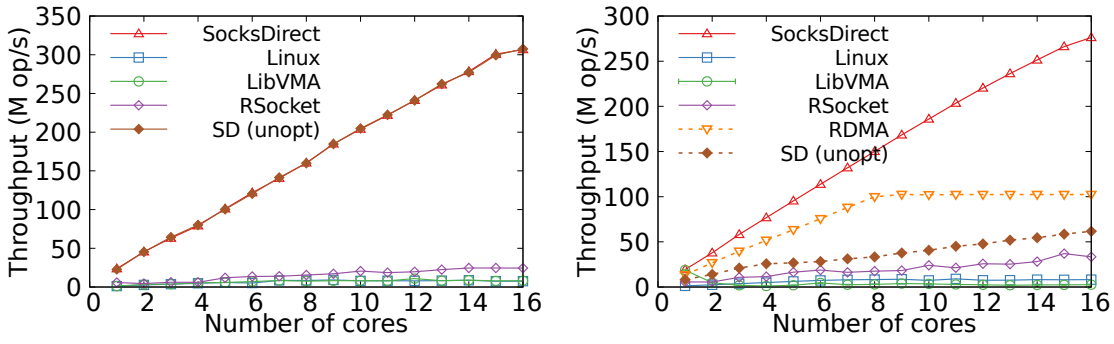
Finally, we evaluate the performance of multiple threads sharing a core. Each thread has to wait for its turn to process messages. As shown in Figure 6.15, although the message processing latency almost linearly increases with the number of active processes, it is still 1/20 to 1/30 of that of Linux.

6.5.3 Practical Application Performance

This section demonstrates that SocksDirect can significantly improve the performance of actual applications without modifying the code. Rsocket [43] is not compatible

Type	Overhead	SocksDirect	LibVMA	RSocket	Linux
Per operation	Total (thread-unsafe)	53	56	71	413
Per operation	Total (thread-safe)	53	177	209	413
Per operation	C library wrapper	15	10	10	12
Per operation	Kernel crossing (system call)	N/A	N/A	N/A	205
Per operation	Socket file descriptor lock	N/A	121	138	160
Per packet	Total (inter-host)	850	2200	1700	15000
Per packet	Total (intra-host)	150	1300	1000	5800
Per packet	Buffer management	50	320	370	430
Per packet	Transport layer protocol	N/A	260	N/A	360
Per packet	Packet handling	N/A	200	N/A	500
Per packet	NIC doorbell and DMA	600	900	900	2100
Per packet	NIC handling & wire		200		
Per packet	Handling NIC interrupt	N/A	N/A	N/A	4000
Per packet	Process wakeup	N/A	N/A	N/A	5000
Per kilobyte	Total (inter-host)	173	540	239	365
Per kilobyte	Total (intra-host)	13	381	212	160
Per kilobyte	Line transmission		160		
Per connection	Total (inter-host)	47000	18000	77000	47000
Per connection	Total (intra-host)	700	3800	33000	14700
Per connection	Initial TCP handshake	16000	16000	47000	N/A
Per connection	Monitor handling	180	N/A	N/A	N/A
Per connection	RDMA QP creation	30000	N/A	30000	N/A

Table 6.4 Latency breakdown of SocksDirect and other systems. The per-operation latency is measured using fcntl()**, the per-packet and per-kilobyte latency is the time from** send() **to** recv()**, and the per-connection latency is the delay of connection creation. The numbers in the table are in nanoseconds and represent rough estimates.**



(a) Single machine throughput.

(b) Cross-host throughput.

Figure 6.14 Throughput of 8-byte messages under different CPU core numbers.

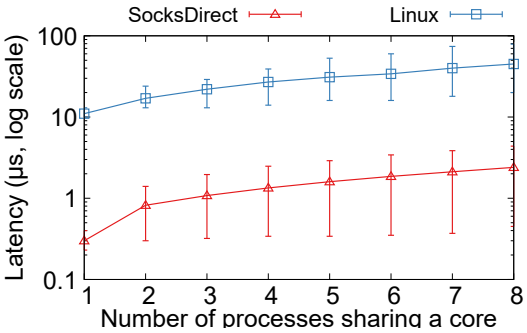


Figure 6.15 Message processing latency when multiple processes share a CPU core.

with any of the following applications.

1. Nginx HTTP Server

To test the typical web service scenario where the client comes from the network and provides services within the host, this section uses Nginx [252] v1.10 as a reverse proxy between the HTTP request generator and the HTTP response generator. Nginx and the response generator are located in the same host, while the request generator is located in a different host. The generator communicates with Nginx using keep-alive TCP connections. Due to fork, LibVMA [22] cannot be used with unmodified Nginx. In Figure 6.16, the request generator measures the time from sending the HTTP request to receiving the entire response. For smaller HTTP response sizes, compared with Linux, SocksDirect can reduce latency by 5.5 times. For large responses, due to zero-copy, SocksDirect can reduce latency by up to 20 times.

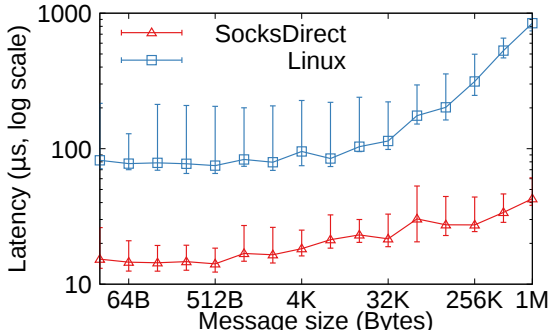


Figure 6.16 Nginx HTTP request end-to-end latency.

2. Redis Key-Value Store

This section uses the redis-benchmark client and 8-byte GET requests to measure the latency of the Redis [78] in-memory key-value store server. When using Linux, the average latency is 38.9 μs, and the 1After using SocksDirect, the average latency is 14.1 μs (64

3. Remote Procedure Call (RPC) Library

This section uses RPClib [292] to measure RPC latency. Running example 1 KiB RPC in two processes within the host with RPClib takes 45 μs. On two hosts, RPC takes 79 μs. Using SocksDirect, the intra-host latency becomes 21 μs (a reduction of 53

However, SocksDirect is not a panacea. Even with libsd, the performance of RPClib is still far below that of the most advanced RPC libraries, such as eRPC[21], due to the overhead of RPClib becoming a performance bottleneck.

4. Network Function Pipeline

64-byte data packets in *pcap* format come from an external packet generator, pass through the Network Function (NF) pipeline, and are sent back to the packet generator. This section implements each NF as a process, which inputs packets from *stdin*, updates local counters, and outputs to *stdout*. For Linux, *pipe* and *TCP socket* are used to connect NF processes within the host. Figure 6.17 shows that the throughput of SocksDirect is 15 times and 20 times that of Linux pipe and TCP socket, respectively. It even approaches the most advanced NF framework, NetBricks[253].

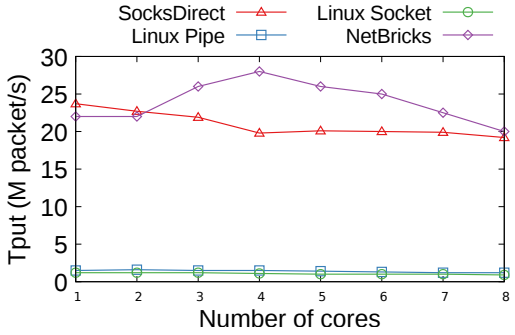


Figure 6.17 Throughput of network function pipeline.

6.6 Discussion: Scalability of Connection Numbers

When using commercial RDMA network cards, SocksDirect scalability for a large number of connections is limited by the underlying transport layer (i.e., shared memory and RDMA). To demonstrate that libsd and the monitor are not bottlenecks, this section creates many connections between two processes that reuse RDMA QP and shared memory. An application thread using libsd can create 1.4 M new connections per second, which is 20 times that of Linux and twice that of mTCP[17]. The monitor can create 5.3 M connections per second.

Since the number of processes within a host is limited, the number of shared memory connections may not be large. However, a host may connect to many other hosts,

and the scalability of RDMA becomes a problem. The scalability of RDMA boils down to two issues. First, RDMA network cards use card memory as a cache to maintain the state of each connection. When there are thousands of concurrent connections, performance is affected by frequent cache misses[21,260,264]. Because RDMA is traditionally deployed in small and medium-sized clusters, the memory capacity on traditional RDMA network cards is small. With the large-scale deployment of RDMA in recent years[41], most network card manufacturers have realized this problem. Therefore, the memory capacity of recent commercial network cards is getting larger and larger, such as Mellanox ConnectX-5[?], which can store the status of thousands of connections[21]. The programmable network card used in this paper even has several terabytes of DRAM [10,137,293]. Therefore, this paper predicts that future data centers will not worry too much about the problem of network card cache misses. The next section will propose a scalable transport layer implementation framework based on programmable network cards. The second issue is that it takes about $30\mu s$ to establish an RDMA connection in the test platform of this paper, which is important for short connections. This process only involves communication between the local CPU and the network card, so this connection establishment latency can be optimized.

Quality of Service (QoS) under a large number of concurrent connections is also an important requirement for data centers. Traditional network protocol stacks implement quality of service guarantees in the operating system kernel. For the hardware transport protocol used in this chapter, offloading data plane performance isolation and congestion control to RDMA network cards is an increasingly popular research direction[12,264-265,294-295], because data center network cards are becoming more and more programmable[10,37,127,137,293], and public clouds have already provided QoS in network functions outside of virtual machines[156,253,296].

Scalability of connection numbers requires storing the transport layer and packet buffer for each connection. For the transport layer state issue, the next two sections propose two solutions: storing connection states and implementing transport layer processing in programmable network cards or user-level libraries on host CPUs. For the packet buffer issue, the last section proposes multiple socket shared queues, merging the buffers of multiple connections between two processes.

6.6.1 Transport Layer Based on Programmable Network Cards

This section implements an RDMA network card with scalable number of connections based on the programmable network card, the network packet processing platform

in Chapter 6, and the in-memory key-value store in Chapter 5. The main challenge of implementing scalable connections is to access the connection status in the host memory at high throughput when the cache miss rate is high, and to hide the access latency. This is exactly the problem solved in Chapter 5, so this section implements scalable connection status storage based on high-performance key-value storage.

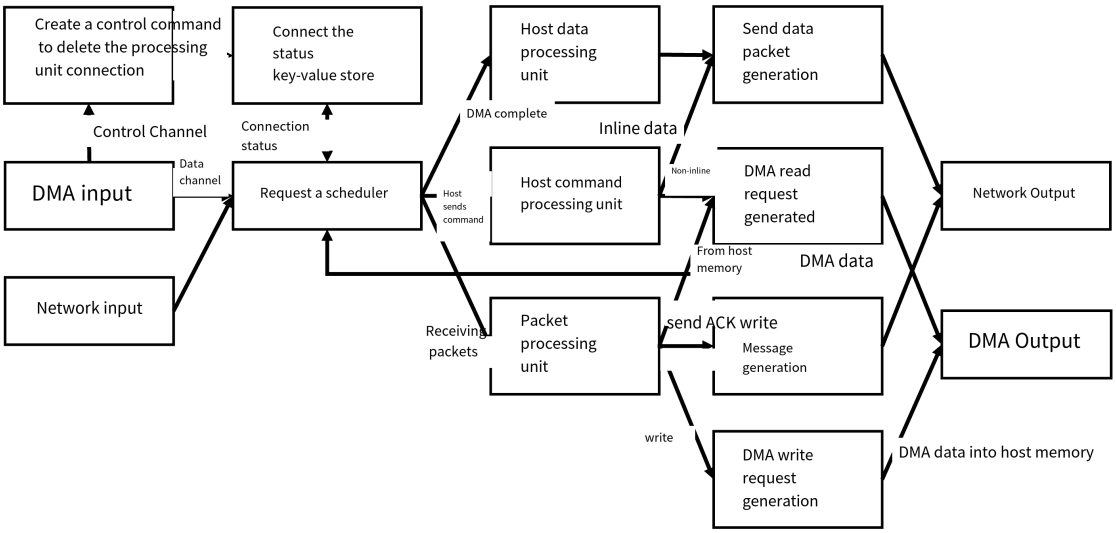


Figure 6.18 Scalable RDMA based on programmable network card.

The architecture of the RDMA network card based on the programmable network card is shown in Figure 6.18. The RDMA network card needs to handle control and data transmission commands (work requests) from the host, and also needs to handle packets from the network. For control and data transmission commands, the host CPU puts the work request into the work queue in the host memory, and then sends it to the network card through PCIe DMA[47]. The packets received through the network interface are also placed in the input buffer of the network card, corresponding to a work request in the work queue. The request scheduler retrieves the connection status information from the connection status key-value store based on the five-tuple information of the packet or the connection number in the host command, and classifies the work request and the current connection status into different work queues inside the network card according to the priority of the connection. Since the processing of RDMA messages is stateful, there may be dependencies between two adjacent packets processed by the same connection. For this reason, the request scheduler records the connections being processed and only schedules work requests for unprocessed connections, which is the same as the way messages with the same key are processed in the key-value store in Chapter 5. For received packets, the receive processing unit processes according to the type of RDMA message. For RDMA one-sided write messages, only need to generate host DMA operations, write data to the corresponding position in the host memory, and

reply with an ACK message. For RDMA one-sided read, atomic, and two-sided send messages, it is necessary to read the corresponding data from the host memory before the next operation can be performed. In order to hide the DMA latency of reading from the host memory, after generating the DMA read request for the host memory, a new work request needs to be generated, waiting for the DMA to complete before proceeding to the next step. This new work request is sent back to the request scheduler, and the waiting condition is marked at the same time. When the DMA is completed, the request scheduler will process this new work request, send data to the network or DMA the data to the host memory. The processing of the send command is similar to the processing of the RDMA one-sided read message. The recv command does not require active processing by the network card, but when a two-sided send or one-sided write with immediate message is received from the network, it is necessary to match the corresponding recv work request.

The performance challenge of the above processing flow is that it is difficult to complete the stateful processing of a work request within a single clock cycle, and the work requests of the same connection cannot be processed in parallel, thereby reducing the single-connection throughput. The solution is to pipeline the processing of work requests, each stage handles different parts of the connection status, so there is no data dependency between stages. A data forwarding mechanism is set up within each stage as in Chapter 5, making the state updates that have not been written back to the request scheduler visible to subsequent work requests. In this way, multiple work requests with dependencies from the same connection can be processed concurrently at different stages of the pipeline. For such dependencies that can be resolved through pipelining and data forwarding, the request scheduler does not need to record the dependencies, but considers all such requests as unrelated.

6.6.2 CPU-based Transport Layer

Another solution to implement connection scalability is to implement the transport layer protocol on the host CPU, so that the network card does not need to store the state for each connection, but only needs to implement stateless offloading. A common scheme for network card stateless offloading is to use the send/receive packet interface between the user-mode protocol stack and the network card, rather than the RDMA remote memory access interface. The packet-based implementation can handle a large number of concurrent connections and can be used on virtualization platforms that do not support RDMA. For example, many virtual machine instances in Microsoft Azure

cloud do not support RDMA, but support high-performance packet interfaces such as DPDK and LibVMA. The packet-based transport layer can be used on these virtualization platforms.

LibVMA uses a high-performance packet send/receive interface with the network card. The compatibility, performance, and multi-core scalability issues of LibVMA are mainly due to its VFS layer. Therefore, this section uses LibVMA to implement transport layer functions and network card interfaces, replacing the ring buffer and RDMA hardware transport layer in libsd. The structure of the LibVMA[22] user-mode socket library is similar to the libsd library in Figure 6.5, which is composed of API encapsulation, VFS layer, queue layer, and transport layer. The queue layer and transport layer of LibVMA are composed of the LwIP lightweight TCP/IP protocol stack and the high-speed packet send/receive interface of the Mellnox network card. To use LibVMA, the queue based on the ring buffer in libsd is replaced with the send/receive interface of LwIP. Tests show that the throughput of the LwIP and network card interface parts in LibVMA for sending and receiving small packets is 18M times per second; the throughput of the API encapsulation and VFS layer in libsd is 27M times per second. This means that the throughput of libsd based on LibVMA can reach about 10.8M small packets per second.

To implement zero-copy based on TCP/IP, the LwIP transport layer in LibVMA needs to be modified. For page remapping, the payload of sending and receiving needs to be aligned to the 4 KiB boundary. When sending, libsd assembles a packet composed of two buffers: first is the packet header assembled from the packet header template through the LwIP transport layer, and then the zero-copy payload. libsd uses the scatter-gather support of the network card to let the network card assemble the two buffers into one packet. When receiving, libsd uses a receive work request containing two buffers, first is a 54-byte buffer that can just accommodate the standard TCP/IP packet header, and then the page-aligned payload buffer. As with the design in Section 6.4.2, libsd replenishes the `recv` work request in time after receiving the packet, keeping the network card always available with the receive buffer.

The above packet interface-based scheme requires the LibVMA library to insert a flow steering rule for each connection, mapping the received packets to the receive work queue. This still requires the network card to maintain the state for each connection, so as shown in the evaluation results in Section 6.5, the performance will still decrease when there are many concurrent connections. In order to make the network card completely save the connection state, an unreliable, congestion control-free one-

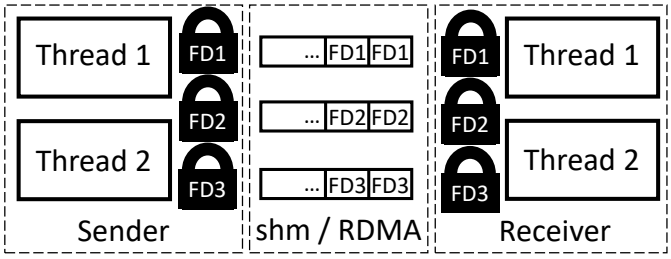
sided RDMA write operation can be implemented with a programmable network card (currently Mellanox RDMA network cards do not support one-sided RDMA based on unreliable datagrams). The one-sided RDMA write operation contains the memory address on the remote host, so the receiving network card only needs to write the payload into the address specified in the packet through PCIe DMA. This way, the ring buffer design in Section 6.4.2 can be applied, and the sender synchronizes the changes in the ring buffer to the receiver through the unreliable channel. The packet loss rate of the RDMA-enabled data center network is very low, so packet loss can be detected by timeout. Specifically, the receiver sends an acknowledgment (ACK) packet as soon as it finds data in the ring buffer; if the sender does not receive the acknowledgment packet after a timeout, it retransmits. To implement window-based congestion control, the sender needs to maintain a send window for each ring buffer (i.e., each connection), and adjust the send window when receiving acknowledgment packets and explicit congestion notifications (ECN). The CPU overhead added by transport layer functions such as packet loss recovery and congestion control is limited. This method can solve the scalability problem of the network card connection number.

6.6.3 Multiple Sockets Sharing Queue

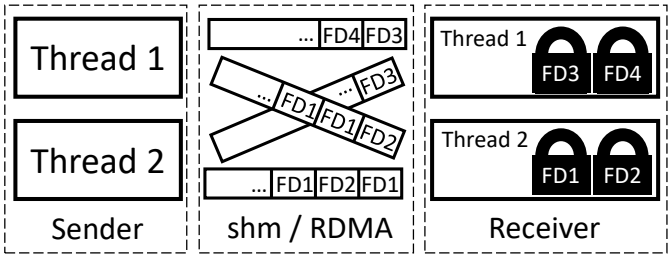
Many applications establish multiple socket connections between two processes. For example, multiple client threads and server threads of a database may establish pairwise connections. HTTP load balancers often establish a connection for each HTTP request with backend web services. Some other transport layer protocols (such as SCTP) and application layer protocols (such as QUIC and many RPC libraries) also provide multi-connection abstraction. In the traditional design, each connection requires independent buffers, so the number of buffers required is relatively large.

To reduce memory usage and improve memory access locality, as shown in Figure 6.19, this paper uses a queue to share all connections between a pair of threads. Each data element in the queue is identified by its file descriptor. By using a queue, the memory usage, random memory access, and cache misses of each socket can be reduced.

Message format in the queue. Based on the traditional queue structure in Section 6.4.2, a file descriptor field is added to the header of each message, indicating the file descriptor of the receiver. In this way, messages from different file descriptors can share a queue. A *next message pointer* field is also added to the header of each message for the following event polling; a *delete* bit is added for the following message retrieval from the middle of the queue.



(a) Traditional queue structure.



(b) Multiple sockets sharing queue structure.

Figure 6.19 Comparison of queue structures. Assume that both the sender and receiver have two threads. First, create peer queues between each pair of sender and receiver threads. Instead of using locks to protect the queue, assign each file descriptor to the receiver thread to ensure ordering. Secondly, data from all connections (file descriptors) is multiplexed through the shared queue, rather than a queue for each file descriptor.

Event polling. Maintain a bitmap for each epoll file descriptor set. When epoll_wait is called, scan all data queues in turn, and check the file descriptor of each data message in the bitmap. If the file descriptor is in the bitmap, return an event to the application. Maintain a global pointer to resume scanning the data queue from the position of the last scan. To avoid scanning the same message multiple times, set a pointer for each queue to save the position of the last scan. Since the application may repeatedly perform receive operations on a file descriptor until the queue of the file descriptor is empty, this paper scans and creates a message linked list for each file descriptor to speed up repeated receive operations. Each file descriptor maintains two pointers, namely the first and last messages scanned but not yet received for this file descriptor. When a new message of the file descriptor is scanned, the next message pointer field in the message header is updated to point to the newly scanned message, forming a message linked list of the same file descriptor.

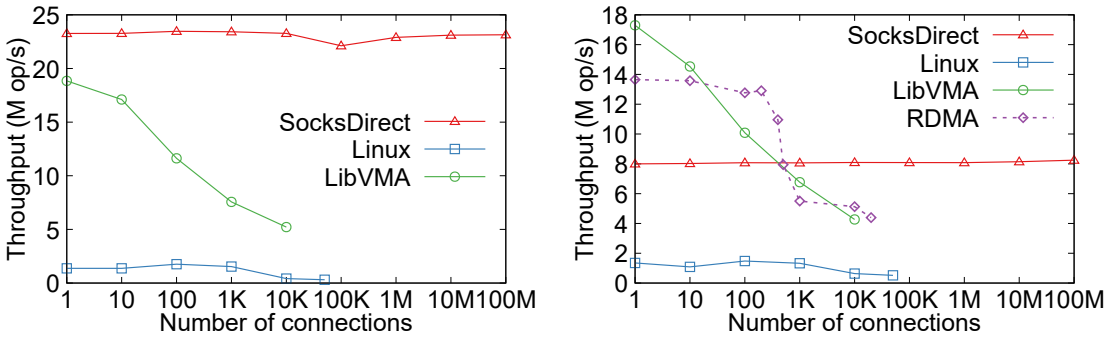
Retrieving messages from the middle of the queue. In order to receive data from any file descriptor, the queue needs to support taking a message from the middle. Fortunately, this does not happen often. Event-driven applications usually process incoming events in a first-come-first-served order. For the level-triggered epoll_wait operation, libsd scans all messages in the queue and returns those messages whose file descriptors have been registered in the epoll file descriptor set. Therefore, when the

application calls recv, the message usually taken is at the head of the queue.

To find a specific file descriptor's message from the middle of the queue, if the message linked list of the file descriptor is not empty, the head of the linked list is the message to be found; if it is empty, it is necessary to traverse the messages in the queue, from the head pointer of the circular buffer to the unallocated space (marked by the *valid* bit). Therefore, when a message is taken from the middle of the queue, its *valid* bit cannot be cleared. Therefore, a *delete* bit is added to each message. When a message is taken from the middle, the *delete* bit is set.

Fragmentation consolidation. If the application does not receive data from a certain file descriptor for a long time, the free space in the queue will become fragmented. When there is no available space in the circular buffer, there may still be many deleted messages in it, but because they are located between other messages of file descriptors that have not been received, the space of these messages cannot be used. When there is no available space in the circular buffer, the sender notifies the receiver to consolidate fragments through the control register in shared memory. The receiver scans the available space in the circular buffer, concentrates the messages that have not been received, and returns the free space to the sender.

The following evaluates the scalability of the number of connections shared by multiple sockets. Before the test, a specified number of connections are pre-established between two processes, and then these connections are used in a round-robin manner to send and receive data in a ping-pong pattern. Figure 6.20 shows the single-core throughput under different concurrent connection numbers. SocksDirect can support 100 M concurrent connections with 16 GB of host memory, and the throughput does not decrease at such a high concurrency. In contrast, the performance of RDMA, LibVMA, and Linux decreases rapidly with the increase of the number of connections. For RDMA, the performance drops rapidly after exceeding 512 concurrent connections, which is due to the RDMA transport layer state filling up the network card buffer. Although LibVMA and Linux do not use RDMA as the transport layer, they maintain buffers for each connection, which leads to CPU cache and TLB misses when there are thousands of concurrent connections. In addition, LibVMA installs a connection redirection rule (flow steering rule) for each connection in the network card, which also causes network card cache misses.



(a) Single-machine throughput.

(b) Cross-host throughput.

Figure 6.20 Single-core throughput under different concurrent connection numbers.

6.7 Limitations

In addition to the performance limitations under high concurrency discussed in §6.6, this chapter will discuss the limitations of SocksDirect in terms of compatibility and CPU overhead.

6.7.1 Compatibility Limitations

Transport Layer. SocksDirect offloads the transport layer mechanism to the RDMA network card. Readers may have some questions about the transport layer mechanism of the RDMA network card. For example, most commercial network cards rely on Priority-based Flow Control (PFC) to eliminate packet loss on Ethernet due to congestion. PFC brings many problems, such as head-of-line blocking, congestion spreading, and even deadlock[41], making the network difficult to manage and understand. We note that many works aim to improve the performance of the RDMA transport layer. The RDMA congestion control algorithms proposed in recent years[264? -266] not only improve throughput and latency, but also reduce the number of PFC pause frames. Many advanced packet loss recovery mechanisms[294-295] also make RDMA no longer need PFC on networks with packet loss. Therefore, we expect future RDMA network cards to provide low-latency and high-throughput transport layers on data center networks with packet loss.

Priority and Quality of Service Guarantee. When multiple threads share the same CPU core, SocksDirect uses non-preemptive scheduling. However, to ensure real-time performance and performance isolation, tasks of different priorities in the data center are usually scheduled to be processed on different CPU cores. Processes running on the same CPU core generally handle similar work tasks, and the working processes of existing software (such as Nginx load balancer, Memcached key-value storage, etc.) usually

process requests in a first-come-first-served order, without setting process priority.

Compatibility limitations also exist in other user-space protocol stacks. First, like other user-space protocol stacks, libsd uses LD_PRELOAD to intercept the glibc API of the application program, and cannot intercept direct system calls, so statically linked applications cannot be used. Second, the sockets created by SocksDirect are not visible in the /proc file system, so some network monitoring tools cannot work. Third, SocksDirect lacks some functions of the kernel protocol stack, such as netfilter and traffic control. However, modern data center network cards already support QoS and ACL offloading [112], so these functions can be offloaded to hardware.

6.7.2 CPU Overhead

SocksDirect eliminates many overheads in existing protocol stacks, but introduces some new ones.

Monitor polling overhead. The polling of the monitor occupies a CPU core. If the monitor is implemented in the kernel and accessed through system calls, the polling overhead will be eliminated, but the overhead of kernel traversal (system calls) and multi-core synchronization in the kernel will increase. Since most control plane operations do not need to go through the monitor, the per-operation overhead added by implementing the monitor in the kernel is acceptable, but it can save the fixed overhead of a CPU core.

Idle process polling overhead. The cooperative non-preemptive scheduling of libsd has two shortcomings. First, if many processes share a CPU core and the arrival of events is relatively random, the above polling method will wake up a large number of processes without pending events, causing an increase in latency. For this, the kernel's cooperative scheduling needs to become more "intelligent", scheduling processes that have tasks to do based on incoming requests, without reintroducing the series of overheads of the original preemptive scheduling. The core method is to adjust the kernel's scheduling queue based on messages. Consider two situations: first, within a single machine, a dispatch process sends messages to multiple worker processes running on the same CPU core. This is a common communication pattern, such as a task dispatcher distributing tasks to different customer network function processes, or a message source distributing events to multiple subscriber processes. The dispatch process manages the scheduling order of the worker processes. The operating system kernel organizes the worker processes running on the same CPU core into a process group, represented by a bitmap, and maps it to the user space of the dispatch process. After the dispatch process

writes data to the shared memory queue, it sets the bit corresponding to the worker process in the bitmap. Modify the kernel scheduler, no longer schedule all ready processes in turn, but scan the bitmap and schedule the next set process. In order to prevent other processes on the same CPU core from starving, the worker process group is treated as a traditional process that is continuously ready and bound to the CPU core. Since non-worker processes are usually in a non-ready (blocked) state, they will not waste CPU time scheduling them.

The second situation is cross-machine communication. At this time, the network card acts as a central dispatcher, and the supported communication mode is arbitrary, not limited to a dispatch process and several worker processes. The event queue of the network card provides the scheduling order of the operating system kernel. The monitor establishes an event queue for each CPU core, which summarizes the completion queue events of all RDMA connections of the processes on the CPU core, written by the network card, and read out by the operating system kernel. The kernel schedules processes according to the order of the event queue, so the scheduled processes are exactly those with events to process. In addition, programmable network cards can observe the length of the event queue on each CPU core, so in a situation where an RDMA message can be distributed to any of the multiple CPU cores, the message can be distributed to the CPU core with the shortest event queue, achieving better load balancing.

6.8 Future Work

6.8.1 Interface Abstraction between Applications, Protocol Stacks, and Network Cards

In this chapter, applications communicate with user-level protocol stacks through socket interfaces, and protocol stacks communicate with network cards through RDMA interfaces. This is to be compatible with existing applications and RDMA network cards. However, applications often have higher-level communication abstractions on the socket layer, such as the key-value storage primitives in Chapter 5, and remote procedure call (RPC), message queue primitives, etc. With the emergence of programmable network cards, the boundary between tasks divided between the host CPU and the network card does not necessarily follow the RDMA interface. Therefore, the interface abstraction between applications, protocol stacks, and network cards can be considered as a whole.

The interface abstraction between applications, protocol stacks, and network cards

not only needs to consider performance issues, but also whether it is easy to program. If only considering from the perspective of performance, for existing network cards with smaller memory capacity, a better task division is to implement the transport layer of high-bandwidth and low-latency connections on the network card, and implement the transport layer of a large number of other connections on the host CPU. However, this requires developers to specify which connections need high bandwidth and low latency, which increases the programming burden; or the protocol stack and network card automatically divide and migrate, which will also increase the complexity of the system.

If the protocol stack and applications can not follow the socket interface, there will be a larger design space. Many related works were introduced in Section 6.2 of this chapter. For example, in terms of zero-copy, if the application can give more hints to the protocol stack, many unnecessary memory copies can be avoided. When the application calls send, it may continue to read and write the send buffer. In order to ensure that the application can read the contents of the buffer, the zero-copy page must be set to read-only, which requires copy-on-write when the receiver in the same host modifies the received content in place. When the application writes to the send buffer, the protocol stack does not know whether the unwritten part of the buffer will be read by the application, so it cannot map an empty page, but needs copy-on-write. This paper intercepts memcpy to optimize the case of whole page writing, but cannot optimize the case where the page is partially written. Many applications actually do not need to read the buffer content after sending. The best solution to the above problems is for the application to inform the protocol stack whether the content of the buffer to be sent needs to be read. This can be achieved by adding an option to the send call, or an additional mem_is_junk API.

The message-based RDMA primitives of the network card and the byte-stream-based socket primitives are mismatched. The ring buffer between the sender and the receiver does not need software explicit synchronization in the shared memory of the CPU. But in the shared memory based on one-sided RDMA, software needs to explicitly send RDMA operations to synchronize the two buffers, that is, to synchronize the sender's data to the receiver, and to synchronize the buffer space released by the receiver to the sender. Compared with hardware-implemented coherent shared memory, software explicit synchronization increases CPU overhead. Implementing ring buffer synchronization in hardware can achieve higher throughput, especially when messages are very small.

The current division of transport layer functions between commercial RDMA network cards and host CPUs is not flexible enough. As discussed in Section 6.6.2, the one-sided RDMA operations in Mellanox RDMA network cards only support Reliable Connection (RC) and do not support Unreliable Datagram (UD). This means that if you want to implement the transport layer on the host CPU, you must use two-sided send and recv operations or other packet sending and receiving interfaces provided by the network card, and you cannot use remote memory access primitives. In addition, the functions in the transport layer such as sequential transmission, congestion control, and packet retransmission are also tightly coupled. They either use the hardware implementation fixed by the network card manufacturer, or they are all implemented in software on the CPU. Programmable network cards provide an opportunity to decouple transport layer functions.

If the network card has the ability to handle a large number of concurrent connections, implementing connection establishment in the network card can save the overhead and delay of the CPU in the process of creating connections.

6.8.2 Modular Network Protocol Stack

The network protocol stack is composed of interface abstraction, congestion control, packet loss recovery, Quality of Service (QoS), Access Control List (ACL), packet format, and other components, as shown in Table 6.5.

Table 6.5 Component selection of the network protocol stack

Component	Selection
Interface Abstraction	RDMA, BSD socket, StackMap, RPC, Message Queue, …
Congestion Control	TCP, DCTCP, DCQCN, TIMELY, MP-RDMA, IRN, …
Packet Loss Recovery	Go-back-0, Go-back-N, Selective Retransmission, Cut-Payload, …
QoS	Strict Priority, RR, WFQ, Multi-Level Feedback Queue, …
ACL	netfilter, OpenFlow, P4, …
Packet Format	TCP/IP, RDMA, RoCE, RoCEv2, …

The most representative RDMA protocol stack and TCP protocol stack each implement a set of different components. The RoCEv2 protocol stack, which is widely deployed in data centers, is derived from the RDMA protocol stack, but replaces the packet format to be compatible with the existing data center network's addressing method based on IP addresses and port numbers. SocksDirect integrates the components of the two protocol stacks, using the socket interface abstraction of the TCP protocol stack, but the rest of the components all use the corresponding components of RDMA. As discussed in Section 6.7, the RDMA congestion control and packet loss recovery algorithms used in SocksDirect may have fairness issues with standard TCP, and also lack QoS, ACL,

and other functions. Therefore, the above components should be modularized and users should be allowed to flexibly combine them. Each component may be implemented in the CPU user mode, CPU kernel mode, or programmable network card. A network protocol stack combination includes the division of tasks between user mode, kernel mode, and programmable network card, as well as the selection of each component. A flexible combination of modular protocol stacks can be implemented using a modular network function programming framework, such as ClickNP in Chapter 4.

As shown in Table 6.6, the interface abstraction of the network protocol stack can be further divided into multiple components. Different combinations of components can form different interface abstractions, suitable for different types of applications, and have different performance characteristics. Many designs of SocksDirect aim to implement the interface abstraction of Linux sockets, and pay a performance price for implementing some of these abstractions (for example, to ensure message ordering, the connection needs to be implicitly exclusive to a thread; due to user-managed buffers, non-page-aligned buffers cannot use zero-copy). We look forward to future work proposing a flexible combination of modular network protocol stack interface abstractions.

Table 6.6 Component selection of network protocol stack interface abstraction

Component	Selection
Addressing Method	IP Address + Port Number, Infiniband Address, Memory Address, Node ID Based on Metadata, …
Connection Abstraction	Byte Stream, Message Stream, Shared Memory, Connectionless, …
Reliability Guarantee	Reliable Order, Tolerate Disorder, Tolerate Packet Loss, Tolerate Errors, …
Connection Sharing Scope	Single Thread, Between Threads, Fork Parent-Child Process, All Processes within Container, …
Connection Sharing Method	Implicit Sharing, Explicit Sharing, Exclusive and Explicit Transfer of Ownership, …
Message Order in Shared Connection	Full Order, Causal Order, No Order, Synchronization Barrier, …
Buffer Management	User Management (RDMA), Protocol Stack Management (socket), User Allocation Protocol Stack Release, Protocol Stack Allocation User Release, …
Notification Method	Blocking, Polling (select), Notify When Ready (epoll), Notify After Completion (aio), Completion Queue (RDMA CQ), …

6.9 Chapter Summary

SocksDirect is a high-performance user-space socket system compatible with Linux. To ensure the reliability of the control plane, this paper designs a monitoring daemon for each host; a point-to-point, synchronization-free data plane that fully supports fork and multi-threaded socket sharing; and a ring buffer that effectively utilizes shared memory and RDMA. SocksDirect achieves performance close to hardware limits

and improves the end-to-end performance of actual applications.

Chapter 7 Conclusion and Future Work

7.1 Summary

Over the past few decades, the evolution of custom hardware has seen its fair share of highs and lows. A decade ago, the idea of incorporating a custom computing device into each server within a data center was nothing short of a pipe dream. However, in recent years, the trend towards large-scale cloud computing, the demands of data center applications, and the performance limitations of general-purpose processors have accelerated the development of custom hardware. This has resulted in a significant improvement in the performance of data center networks.

The advancement of custom hardware and the communication requirements of distributed systems have led to the widespread deployment of programmable network cards in data centers. Microsoft uses FPGAs to boost the performance of search engines, virtual networks, compression, machine learning inference, and so on. Amazon and Alibaba Cloud enhance virtual networks, virtual storage, and virtual machine monitors. Tencent Cloud employs FPGAs, while Huawei Cloud utilizes network processors to speed up virtual networks. Looking back, network virtualization may have been the first major application of programmable network cards, but this only scratches the surface of the potential of these devices.

To fully leverage the high performance of data center networks, it is crucial to minimize the ”data center tax”. This includes not just network virtualization, but also network functions and operating system communication primitives. This paper proposes the acceleration of network functions using FPGA-based programmable network cards. To simplify FPGA programming, this paper introduces the first FPGA programming framework suitable for high-speed network packet processing based on high-level languages. This improves throughput tenfold and reduces latency to a tenth compared to traditional CPU-based network functions. To decrease the overhead of operating system communication primitives, this paper suggests a combined software-hardware user-space socket system. This system is fully compatible with existing applications and can achieve throughput and latency close to hardware limits. This resolves the longstanding conflict between the low performance of general protocol stacks and the poor compatibility of dedicated protocol stacks.

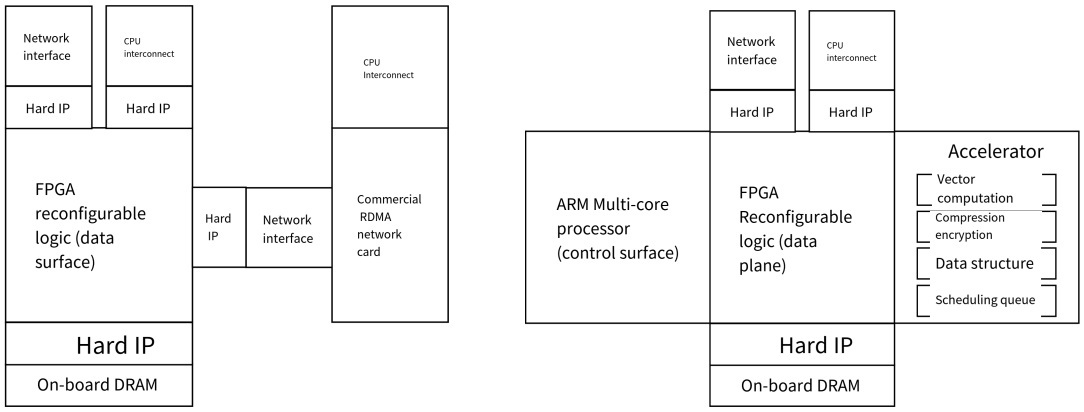
The term ”programmable network card” is derived from network acceleration, but its influence extends beyond the network and continues to permeate various areas of

the system. Memory data structure storage is a crucial foundational component of distributed systems. This paper introduces a remote direct key-value access primitive, an extension of the remote direct memory access (RDMA) primitive. By bypassing the server-side CPU and directly accessing the host memory with the programmable network card, along with a series of performance optimizations, this paper achieves a throughput ten times that of the CPU key-value storage system and microsecond-level latency. This makes it the first general-purpose key-value storage system with a single-machine performance reaching one billion operations per second.

Without a doubt, programmable network cards can enhance system performance and reduce data center costs. The three systems proposed in this paper establish new performance benchmarks for virtual network functions, general memory key-value storage, and socket network protocol stacks. However, the aim of this paper is not to break performance records, but to inspire readers to consider: How can we build a programmable network card ecosystem that includes hardware, development toolchains, and operating systems? How will new hardware, such as programmable network cards, alter the architecture of data centers and the programming paradigm of distributed systems? As it has been said, ”The best way to predict the future is to create it”. The story of programmable network cards is just beginning.

7.2 Future Work

High-performance data center systems based on programmable network cards necessitate a combined software-hardware ecosystem, primarily composed of hardware, development toolchains, and operating systems. Section 7.2.1 will discuss the future hardware architecture of programmable network cards. The development toolchain, including programming frameworks, compilers, runtime libraries, debugging tools, etc., is vital in software-hardware co-design. Section 7.2.2 will discuss future works in the development toolchain. The operating system, including virtualization, scheduling, monitoring, high availability, flexible scaling, etc., will be discussed in Section 7.2.3. Finally, as a first-class citizen in the data center, the programmable network card not only enhances system performance but also prompts us to rethink the overall architecture of distributed systems, which may lead to system innovation. This will be discussed in Section 7.2.4.



(a) The Catapult programmable network card used in this paper.

(b) Future system-on-chip.

Figure 7.1 Comparison of programmable network card structures.

7.2.1 Programmable Network Card Based on System-on-Chip

This paper utilizes the Catapult programmable network card depicted in Figure 7.1a. This architecture has three limitations. Firstly, the performance of existing commercial RDMA network cards significantly decreases when the number of concurrent connections is large[264]. We aim to use the scalable key-value storage technology in Chapter 5 to implement the RDMA hardware transmission protocol in FPGA reconfigurable logic to achieve high performance under high concurrent connection numbers. This has been discussed in Section 6.6. Secondly, FPGA is only suitable for accelerating the data plane, and the control plane is still left on the host CPU. Although its computing power is not large, for performance isolation, computing nodes still need to reserve a small number of CPU cores for control plane processing. Chapter 1 has pointed out that even reserving a physical CPU core is quite expensive. For this reason, we hope to add ARM multi-core processors to the programmable network card to implement the control plane, thereby completely eliminating the virtualization overhead on the host CPU. The cost of ARM multi-core processors is tens of dollars, far lower than the cost of a physical CPU core. Finally, some types of workloads are not very efficient when implemented in FPGA and should be solidified in ASIC accelerators. The first type is computationally intensive operations such as vector operations, encryption and decryption operations in deep learning and machine learning. For example, the RSA asymmetric encryption based on the Intel QuickAssist accelerator card[142] is about 10 times higher in throughput than the FPGA-based implementation in Chapter 4; the LZ77 compression algorithm based on ASIC is also an order of magnitude higher in throughput than the FPGA-based implementation in this paper. The power consumption, area, and process of the used ASIC and FPGA chips are close. The second type is common

data structures and scheduling queues. The lookup table based on Content-Addressable Memory (CAM) is a necessary component of many common data structures such as hash tables, out-of-order execution engines, caches, fuzzy matching tables, etc. CAM can be implemented with ternary gates in ASIC, but the efficiency of implementation in FPGA is low[297]. In addition, priority queues (which can be implemented with shift register sequences or heaps), round-robin scheduling queues, out-of-order execution schedulers considering dependency relationships, timers, and other structures are widely used in many applications, so they can learn from the architecture of network processors, harden these general structures, and let FPGA reconfigurable logic focus on customized computing and flexible interconnection.

Therefore, this paper anticipates that future programmable network cards will utilize the system-on-chip architecture as depicted in Figure 7.1b. Compared to separate components interconnected by off-chip buses, the system-on-chip can offer higher bandwidth and lower latency for inter-component communication, making it more suitable for dividing computations into finer granules to more appropriate processing components. The FPGA at the heart of the system-on-chip not only provides programmability and computational capabilities, but also flexibly interconnects and combines various on-chip computational accelerators, forms customized memory hierarchy, and flexibly interconnects various hardware devices inside and outside the host to form an intelligent fabric for data centers.

At present, there are programmable network card architectures based on system-on-chip in the industry. For instance, Xilinx's Versal architecture[298-300] integrates reconfigurable hardware (FPGA), deep learning and traditional machine learning accelerators based on Very Long Instruction Word (VLIW), Digital Signal Processors (DSP) and hard IP, as well as multi-core general processors on a single chip to form a System on Chip. Compared to traditional FPGAs, the most significant difference of the Versal architecture is that it forms a system-on-chip, which is reflected in three aspects: Firstly, it hardens the control logic of external interfaces such as memory controllers and PCIe into digital logic, reducing the area overhead of FPGA and enabling plug-and-play for FPGA. Secondly, it recognizes the low efficiency of implementing common computations such as vector operations in big data and machine learning on FPGA, and accelerates them with hardened digital logic. Thirdly, it adds general processors that can handle complex logic and control planes without having to loop back to the CPU, enabling the Versal system-on-chip to directly drive Flash storage, etc., to form low-cost storage servers without traditional components such as x86 CPUs. The components of

the system-on-chip are interconnected through an on-chip network[300-301]. The Versal architecture accelerates various applications of data center servers, and developers can decompose applications into control planes on general processors, data planes on reconfigurable hardware, and vector computation data planes, using the appropriate architecture to handle the corresponding parts of the application.

7.2.2 Development Toolchain

At present, programmable network cards are a burgeoning technology, primarily propelled by cloud computing manufacturers. However, their ecosystem is still in its infancy. Firstly, the development toolchain for programmable network cards, including compilers, debugging tools, code libraries, and so forth, lacks flexibility and user-friendliness. Moreover, the support provided by relevant manufacturers is not yet comprehensive. Recent high-level synthesis tools are primarily focused on the programmability of FPGA in computation-intensive processing (such as deep learning), with less emphasis on communication-intensive processing. Although this paper's ClickNP in Chapter 4 has made some strides in this direction, it is still a considerable distance from large-scale commercial applications.

Secondly, in current research, the task division between programmable network cards and applications is rather arbitrary. There is a need for quantitative research methods to determine which workloads are suitable for offloading to programmable network cards. For an existing application to utilize a programmable network card to accelerate its data plane functions, a significant amount of code needs to be rewritten. This includes not only implementing the data plane processing logic from scratch within the programmable network card but also modifying the control plane code on the host CPU to fully exploit the network card's parallelism and hide processing latency. Future development toolchains need to reduce the secondary development cost of existing applications.

1. PCIe Debugging Tools Based on Programmable Network Cards

Data center servers are increasingly loaded with more PCIe devices, such as GPUs, NVMe SSDs, network cards, accelerators, and FPGAs, among others. For high throughput and low latency communication between PCIe devices, technologies like GPUDirect, NVMe over Fabrics are gaining popularity. However, many PCIe devices can only communicate with device drivers on the CPU. Their PCIe registers and DMA interfaces are complex and may lack documentation. To capture packets on PCIe and debug the implementation of PCIe protocols, developers often require expensive phys-

ical layer PCIe protocol analyzers (worth approximately 250,000 US dollars). These protocol analyzers necessitate a laboratory environment and are challenging to debug dynamically in a production environment. Furthermore, protocol analyzers cannot modify PCIe packets and lack sufficient programmability to detect anomalies or statistical patterns from a large volume of traffic data.

A potential future direction involves the implementation of a transparent PCIe Transport Layer Protocol (TLP) debugger based on programmable network cards. This PCIe debugger would capture communication packets exchanged between the PCIe device and the CPU. The challenge in this endeavor lies in the fact that the physical topology and routing of PCIe are fixed, making it impossible to implement attacks similar to ARP in local area networks on PCIe. However, by deceiving the device driver, PCIe traffic can be redirected to the PCIe debugger. Depending on the initiator of the request, the communication between PCIe and CPU can be divided into two categories.

The first category encompasses Memory Mapped I/O (MMIO) operations initiated by the CPU. In these operations, the CPU accesses the memory area pointed to by the PCIe Base Address Register (BAR). The driver program obtains the BAR address from the operating system kernel routine, and it can modify this operating system kernel routine to return the address of the PCIe debugger, rather than the address of the device itself. Subsequently, an address mapping is established in the PCIe debugger, allowing the CPU's memory-mapped I/O operations to be transmitted to the PCIe debugger. The PCIe debugger then acts as a proxy to send the request to the target device.

The second category involves device-initiated DMA operations used to access host memory. At first glance, it may seem impossible to predict which memory address the device will access. However, well-defined devices should only access addresses allocated to them by the driver. In Linux, there are two methods for device drivers to obtain DMA memory areas and their physical addresses. The plan is to modify these two operating system routines separately, replacing the host memory address with the PCIe debugger's address when allocating DMA memory areas, and establishing address mapping in the PCIe debugger. Consequently, when the device attempts to DMA to the host memory, it is actually DMAing to the PCIe debugger, which then DMAs the data back to the host memory according to the mapping table.

By employing this method, the communication between the host driver and the PCIe device will be intercepted by the PCIe debugger. FPGA-based PCIe transport layer protocol debuggers possess sufficient flexibility to modify, count, filter, and inject packets, thereby facilitating fuzz testing and stress testing of PCIe devices.

2. Microsecond Latency Hiding

When using customized hardware to accelerate CPU processing in applications, the original CPU software processing logic is replaced with three steps: sending commands to the accelerator, waiting for the accelerator to process, and receiving results from the accelerator. During the waiting period, the CPU thread is blocked. Similarly, in distributed systems, remote procedure calls (RPCs) are often needed and wait for results from other microservices or nodes. Traditionally, developers generally use the method of adding more threads to hide the latency of accelerator processing and remote procedure calls, that is, letting the operating system switch to other threads for processing during this period. However, with the improvement of data center accelerator performance and the reduction of acceleration task granularity, some acceleration tasks only take a few microseconds to tens of microseconds. Similarly, the network latency of remote procedure calls has also dropped from previous milliseconds to microseconds to tens of microseconds. Operating system thread scheduling also requires 3 to 5 microseconds, which is almost equivalent to the execution time of acceleration tasks and the network latency of remote procedure calls. This means that switching to other threads during the waiting period is not economical, and it may be better to let the CPU wait for the completion of the acceleration task on the current thread. However, this also means a waste of CPU time during the waiting period, which to some extent affects the effect of customized hardware accelerators saving CPU.

A future research direction is to implement microsecond latency hiding of applications from the perspective of compilation. We have two main observations: first, the application may have multiple independent hardware acceleration tasks to be processed, so it is possible to mine these independent acceleration tasks for concurrent processing. Second, many applications are event-driven, that is, they process incoming events in a permanent loop. There may be no dependencies between different event processing, so you can temporarily suspend the event being processed and process the next unrelated event.

The complexity of these two latency hiding schemes resides in the determination of ”dependencies”. In functional programming languages, the dependencies between pure functions are relatively straightforward to discern. However, in most programming languages frequently utilized by developers, memory is shared, and many codes are interdependent. For instance, object creation necessitates memory allocation, which influences the memory layout. Therefore, strictly speaking, the creation order of any two objects is dependent. Whether there is a dependency between two remote procedure

calls often hinges on their semantics. Consequently, the central challenge of the problem is for developers to specify which dependencies are genuinely unnecessary.

A potential solution is the ”async” decorator, which enables developers to designate that a function can be executed asynchronously. Async functions can internally use wait calls to register events, relinquish the CPU, and awaken when the event is established (for example, awaiting the return of the accelerator or remote procedure call). Functions that can be executed asynchronously will not be interrupted during execution (unless a wait is called, or there is an asynchronously executable subroutine), thus eliminating concerns about reentry problems. Each async function execution is implemented with a coroutine. Furthermore, the ”async pure” decorator is proposed, which allows developers to specify that a function can not only be executed asynchronously but also has no side effects, so it can be speculated that it is executed, that is, it is executed when the execution conditions are not yet determined, without worrying about it producing irreversible side effects.

For instance, offloading stateless computations to accelerators, read-only remote procedure calls, and opening files are async pure functions. And executing write operation remote procedure calls, processing an event routine is a general async function. If there is a logical dependency between async functions, for example, different events initiated by the same user need to be processed in order, then a lock can be set for each user, and the lock is added at the beginning of event processing and unlocked after the end. The lock is implemented with a wait call, so the overhead is very minimal.

In addition to mining the internal parallelism of applications from the perspective of compilation, another future research direction is to implement high-performance context switching and scheduling managed by hardware from the perspective of architecture[6]. The network processor hardware scheduler introduced in Section 2.3.2 can be used as a useful reference.

3. Translation from High-Level Language to Low-Level Language

Modern software enjoys the dividends of Moore’s Law. For the sake of development efficiency, it generally uses high-level language modular programming, and the compiler’s optimization of software is not sufficient. Modern software written in high-level languages, even if it has similar functions to software based on low-level languages (such as C language) many years ago, its performance often differs a lot. The ”Andy-Bill Law”[302] vividly depicts this phenomenon, that is, the performance increase brought by high-performance new processors (represented by Intel’s CEO Andy) is often consumed by software (represented by Microsoft’s founder Bill Gates), and the

performance perceived by end users is still similar. There is also a lot of room for optimizing software performance from the perspectives of programming frameworks and compilers. David Patterson pointed out that rewriting Python language as C language can improve the performance of applications by 50 times, and if a series of optimizations are used, achieving a performance improvement of 1000 times compared to Python is not a dream[303].

The future research direction is the automatic translation from high-level languages to low-level languages. Although many high-level languages are dynamically typed and have high-level language features such as introspection, the type of input for a given high-level language application is often relatively certain.

4. Automatic Generation of Network Application Data Plane

In order to enhance the efficiency of network applications and minimize CPU overhead, data centers have begun to incorporate programmable switches and network cards to offload virtualized network functions, transport protocols, key-value storage, distributed consistency protocols, and so forth. Compared to general-purpose processors, programmable switches and programmable network cards have fewer resources and more restricted programming models.

Consequently, developers typically partition a network function into a data plane that manages common case packets and a control plane that deals with the remainder. The data plane function is implemented in a packet processing language (such as P4) and offloaded to hardware.

Creating packet processing programs for network application offloading is labor-intensive. Initially, even with protocol specifications or source code, developers still have to sift through thousands of pages of documents or code to identify the common function. Additionally, many implementations and protocol specifications have subtle differences, so developers often need to examine packet capture records and manually reverse engineer behavior specific to an implementation.

The future research direction is to automatically learn the behavior of a specified network application and thereby automatically generate reference code for the data plane. In this manner, developers only need to design some simple data plane test cases and run the specified network application. The data plane automatic generation system will capture the input and output packets and search for a packet program to generate the output tested for the specified input test cases.

Clearly, passing the test cases does not imply that the program can correctly generalize in other input situations, so the automatically generated code can only serve as

a reference for developers, who can supplement the details of special case handling based on it. Nevertheless, the automatically generated reference program can assist developers in understanding the usual working mode of the protocol and save a significant amount of development time.

In general, generating programs through examples is considered challenging due to the vast search space and the theoretically undecidable halting problem. Fortunately, packet programs that can be offloaded to hardware are typically quite simple. Commercial programmable switches and network cards do not support loops and recursion, eliminating the issue of determining the halting problem. Additionally, for each persistent state, each packet is only allowed one read-write operation on the data plane. Furthermore, the logical depth from packet input to output is limited by the hardware pipeline depth. These restrictions significantly reduce the program's search space. More importantly, to reduce the search space, test cases can be generated to eliminate some potential search directions. To generalize test cases as much as possible, the generate and test method is used to observe the behavior of the specified application. To select one of the infinitely many programs that can generate the specified output, Occam's razor is used to choose the program with the shortest description length. When multiple programs have the same description length, the system can generate deterministic test cases to determine the correct one, or report to the user.

5. Task Division of Heterogeneous Distributed Systems

The data center is a distributed system composed of heterogeneous hardware. Each type of hardware possesses certain computing, storage, and network interconnection resources. Different hardware can perform different types of calculations and have different calculation efficiencies, for example, the CPU is suitable for control-intensive calculations, the GPU is suitable for general Single Instruction Multiple Data (SIMD) type calculations, the TPU is suitable for convolution and matrix multiplication type calculations, and the FPGA is suitable for communication-intensive calculations. The ability of heterogeneous hardware to communicate with each other also varies, for example, GPUs can communicate directly through NVLink, while the FPGA, as a programmable network card, is a necessary pass-through between the server host and the data center network.

Given a computational flow graph and a high-level language description of each component within it, a crucial question arises: how should these components be mapped onto heterogeneous computing hardware? Clearly, considering only the execution efficiency of each component on various computing hardware is insufficient. The commu-

nication overhead between components must also be taken into account. For instance, the normalization operation between convolution layers in neural networks may have a higher execution efficiency on a GPU than on a TPU. However, the data migration overhead between the GPU and TPU may outweigh the performance loss of executing the normalization operation on the TPU. Therefore, it might be more performance-optimized to fuse the convolution and normalization operations on the TPU.

Generally, a heterogeneous computing cluster can be formalized as a topology graph. The vertices represent computing devices, memory and storage devices, and network switching devices, while the edges represent data paths between nodes. Each computing device supports several computing types and possesses the bandwidth and latency of each type of computing. The attributes of the data path include bandwidth and latency. Each vertex in the computational flow graph represents the amount and type of computation, and each edge represents the amount of data to be transmitted. The task division problem aims to find a mapping from the computational flow graph to the topology graph of the heterogeneous computing cluster, ensuring that the latency and throughput meet the application's constraints.

For components with large computational scales, they also need to be divided across multiple hardware for parallel or pipeline execution. A component's computation may have multiple splitting methods, and different splitting methods require different communication overheads. It is necessary to calculate the amount of computation required on each hardware based on system performance requirements or the limitation of the number of heterogeneous hardware. Then, the optimized component splitting scheme can be obtained based on the communication and computation overhead model.

7.2.3 Operating System

The "operating system" of a distributed system encompasses the conventional operating system on the host, the scheduling, management, monitoring system of the distributed system, and shared basic service middleware. This paper investigates the optimization of the operating system network protocol stack and key-value storage in the distributed system. However, there are also multiple subsystems within the operating system and various middleware in the distributed system, such as storage and message queues. These subsystems and middleware can evidently also be accelerated with programmable network cards.

Moreover, in traditional distributed systems, due to high communication costs, hot migration and high availability often necessitate developers to utilize specific program-

ming frameworks. In data centers with high-performance data center networks, based on programmable network cards, it might be feasible to achieve efficient hot migration and high availability of general applications. This will render the data center more akin to a colossal computer, where applications can fully exploit heterogeneous computing and storage resources, and are almost imperceptible to hardware failures.

1. Acceleration of Storage Virtualization

Virtual storage in cloud computing comprises local storage and remote storage. Remote storage is a distributed storage system virtualized by storage nodes, offering high reliability, high availability, scalable capacity, and scalable throughput, which is the primary storage method in the cloud platform. Local storage includes Non-Volatile Memory (NVM) and NVMe high-speed flash storage, primarily used for distributed databases and other applications that necessitate extreme performance but do not require high reliability storage.

The most fundamental service that virtual storage provides to customers is block storage, which can be mounted as a block device to a virtual machine for use as a disk. Cloud services also provide other storage services such as object storage and file storage. Most of these services provide an abstraction similar to key-value mapping, i.e., the user specifies a key to read (GET) or write (PUT) the corresponding value. Key-value storage, as a basic data structure, can be divided into persistent storage and temporary storage; depending on whether replication and disaster recovery are needed, whether transactions are supported, whether strong consistency or eventual consistency is provided[304], and whether range indexing, secondary indexing, content indexing, etc., are supported, a variety of storage systems can be combined to meet the needs of different applications.

The two fundamental logical concepts of the virtual storage system are the client and the server. As depicted in Figure 7.2, the client is the user of the cloud storage service, such as the computing node hosting the customer's virtual machine in the cloud; the server provides abstractions like block storage, object storage, and file storage, mapping logical storage read and write requests to physical storage media read and write requests. Multiple clients may share the same virtual storage, for instance, multiple computing nodes in a distributed data processing system may need to access shared raw data and configuration parameters, and the intermediate results of data processing can also be passed through storage. The same virtual storage may correspond to multiple storage servers, used to achieve scalable storage capacity, scalable throughput, fault tolerance, and high availability.

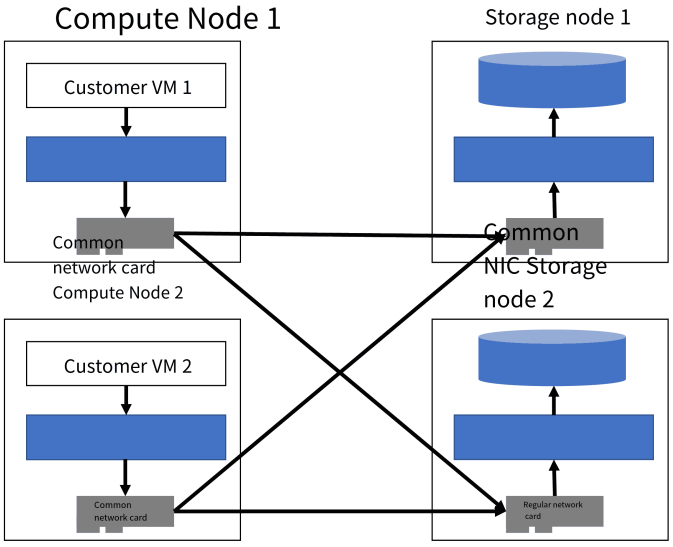


Figure 7.2 A brief architecture of data center cloud storage.

The above storage server structure is relatively simplified, in reality, it is often divided into multiple levels. For example, Microsoft Azure's cloud storage service is divided into front-end nodes, middle nodes, and back-end nodes[147]. The front-end node is responsible for parsing and verifying requests, and based on the data shard mapping table (such as the hash value of the key), it distributes to the middle node of the data shard. The middle node is responsible for implementing the processing of requests and storage data structures, mapping user requests into a series of storage read and write operations, and distributing them to the corresponding back-end nodes. The back-end node is responsible for implementing data replication and storage on physical media.

In addition to software processing, data center storage also has significant network overhead. In the data center, because the storage server needs to install a large capacity of storage media, the hardware configuration of the storage node is generally different from that of the computing node. Moreover, because the virtual machine monitor software on the computing node often needs to be upgraded to add new features and patch security vulnerabilities, the stability of the computing node is generally lower than that of the storage node. To ensure the high availability of storage, the storage node and the computing node are usually separated on different physical hosts. Therefore, the storage client software in the virtual machine monitor on the computing node usually needs to move the data from the storage server over the network. That is, each I/O request of the customer's virtual machine needs to be captured by the virtual machine monitor, and then starts from the storage client software in the virtual machine monitor on the computing node, and goes through the processing of the front-end, middle, and back-end nodes of the storage server before it can reach the storage medium. To

ensure the security of data, the data on physical storage media generally needs to be encrypted. To save storage space and reduce the cost of unit storage capacity, many cloud manufacturers also compress the content of storage. Compression and encryption are generally performed on the storage server and are computationally intensive operations. For example, according to experiments, under the good compression rate of LZ77, a server CPU core can usually only compress 100 MB of data per second; for a 1 KB block, AES encryption and SHA-256 signatures can also only process 100 MB of data per second.

Due to the overhead of software processing and network transmission, the latency of block storage on the cloud computing platform is generally 0.5 to 1 millisecond, and the latency of object storage is generally 1 to 10 milliseconds[100], which is significantly higher than the latency of physical storage media (for instance, the latency of SSD is generally 0.1 millisecond). Moreover, the throughput of cloud storage is also lower than the corresponding physical storage media. For example, the highest throughput of SSD cloud disk is 50 K I/O per second, while the throughput of a single data center-level SSD has reached hundreds of K I/O per second[100]. To fully utilize the performance of the latest data center storage hardware, cloud storage services need to be optimized across the stack. For instance, many data centers have already used the RDMA protocol to reduce the CPU overhead and latency of the network protocol stack in the storage protocol stack[41]. Some data centers also reduce the number of layers by improving the protocol stack of cloud storage and appropriately integrating the functions of the client, server front-end, middle, and back-end nodes[151]. HyperLoop[305] uses RDMA network cards and Non-Volatile Memory (NVM) to reduce the latency of storage write transactions.

2. Acceleration of Remote Procedure Call and Message Queue

Message passing in distributed systems usually adopts the Remote Procedure Call (RPC) or Message Queue model, or a combination of both. In the RPC model, the server registers a procedure to respond to the client's RPC request. In the message queue model, the producer broadcasts or distributes messages to several consumers. To achieve decoupling of producers and consumers, buffering of messages, and reliable delivery, the message queue model often introduces a broker service, such as Kafka[77]. In terms of programming interfaces, distributed applications usually use RPC libraries and message queue middleware, which rely on the operating system's socket interface to send and receive messages. This paper studies the acceleration of the operating system socket interface, but does not consider higher-level RPC and message queue middle-

ware. Google's research[6] shows that these message middleware often add tens of microseconds of delay, accounting for a large part of the entire end-to-end network delay. To reduce the end-to-end message passing delay of distributed systems, it is necessary to use programmable network cards and other hardware and user-space libraries to achieve high-performance RPC and message queues. One solution is to implement high-level abstractions such as RPC and message queues based on the user-space socket system in Chapter 6; another solution is full-stack optimization that breaks the boundaries of traditional network protocol stacks, such as the recent eRPC[21] is a meaningful exploration in this regard. For simpler applications like message queues, it is even possible to explore implementation in programmable network cards, bypassing the host CPU.

3. User-space Operating System Based on Microkernel

Chapter 6 proposes a user-space network protocol stack, SocksDirect. The technology in Chapter 6 can be used to accelerate more abstractions of the operating system.

In addition to the network protocol stack, the storage protocol stack of the operating system also has high overhead. The Linux storage protocol stack is logically composed of five layers. First, there is a virtual file system layer similar to the network protocol stack, providing an API based on file descriptors. Second, the file system layer implements the abstraction of the file system, providing functions such as file path lookup, permission management, and space allocation. Third, the cache buffer layer is closely integrated with Linux's memory management mechanism, responsible for managing read cache and write buffer, as well as the page swapping mechanism. Fourth, the block device layer abstracts the storage device into several "blocks", implementing the merging and sorting of block access. Fifth, at the device driver layer, the storage medium driver communicates with the hardware to read and write disk blocks. In the storage protocol stack, the virtual file system layer is also an important source of overhead. For many applications that use direct I/O, such as databases, the file system and cache buffer layers are unnecessary. Similar to the network protocol stack, there are multiple data copies in the storage protocol stack. For many applications, there is also a copy between the storage and network protocol stacks. Zero-copy technology based on page remapping can be used in network and storage protocol stacks, so that each piece of data only has one copy in physical memory, and only the mapping relationship of virtual memory is copied between protocol stacks.

The technology in Chapter 6 has a broader application prospect: a user-space operating system based on microkernel. The operating system mainly includes three functions: resource virtualization, inter-process communication, and high-level abstraction.

Chapter 6 implements the virtualization of network resources and inter-process message passing, providing a high-level abstraction of sockets. A user-space storage protocol stack can implement the virtualization of storage resources and the high-level abstraction of the file system. The remaining functions of the operating system include the virtualization of computing resources (i.e., process scheduling) and inter-process synchronization (such as locks and semaphores). These functions can be implemented in user-space daemons or in programmable network cards. After the functions of the traditional operating system are moved to user-space and programmable network cards, a microkernel can be adopted while maintaining compatibility with existing applications.

An operating system based on a microkernel not only has higher performance but also facilitates the implementation of high availability for general distributed applications, which will be the topic of discussion in the next section.

4. High Availability of General Distributed Applications

Hardware failures and operating system crashes can cause some nodes of distributed applications to fail. High availability of distributed applications is very important. Many existing request processing and batch processing systems can simplify fault-tolerant programming of distributed applications. These programs usually require programmers to explicitly separate computation from state and store the state in a fault-tolerant storage system. However, many existing applications (such as Node.js, Memcached, and Python logic in Tensorflow) do not natively support fault tolerance. In addition, fault-tolerant programming frameworks usually have lower performance than non-fault-tolerant versions. We hope to solve the challenge of transparent and efficient fault tolerance for general distributed applications. Specifically, the challenges are divided into process migration, deterministic replay, and distributed snapshots.

First, there is a trade-off between fault tolerance at different levels. Fault tolerance at the architectural level requires customized hardware. Fault tolerance mechanisms at the virtual machine level consider all network communications to be bidirectional (because there are data transmission and ACK confirmation messages), and cannot discover high-level semantics such as inter-process communication. Fault tolerance at the system call level requires modifications to the operating system kernel to implement process migration, for example, extracting the process state from the source host and injecting it into the destination host. Process migration in Linux is complex because states from different processes are mixed in a macrokernel. The Unikernel method cannot support many existing inter-process communication mechanisms. For this, future research can draw on the SocksDirect architecture in Chapter 6 to design a distributed

user-space runtime library operating system that is compatible with existing Linux application programming interfaces. The process memory snapshot simultaneously obtains the state of the runtime library and the application, retains high-level semantics, and is easy to optimize.

Secondly, State Machine Replication (SMR) and snapshot replay are two primary methods to achieve fault tolerance. SMR necessitates at least two hosts to run the same application, thereby introducing CPU overhead. Snapshot-based systems usually buffer an application's output during the interval between two consecutive snapshots, as the system cannot guarantee deterministic execution since the last snapshot when a host fails. This so-called output commit problem introduces a significant request service delay for transparent fault-tolerant systems. Alternatively, recording all non-deterministic events of an application still incurs substantial overhead. A future research direction is to predict an application's non-deterministic events based on its recent execution history. If the prediction is accurate, the application continues. Otherwise, it waits for a short time to realize the prediction, as many uncertainties arise from tiny time fluctuations. In this way, the system only needs to record the events of incorrect prediction when the waiting times out, reducing the recording overhead.

Thirdly, transparent fault tolerance mechanisms need to take snapshots of distributed applications without pausing the entire system. Consistent snapshot algorithms require all hosts to take snapshots at the same speed and roll back simultaneously when any host fails. This global synchronization behavior contradicts the goal of fault tolerance, which requires the system to continue providing delay-sensitive requests when a host fails. How to asynchronously generate consistent snapshots for distributed systems is a future research direction.

5. Data Center Resource Packing Based on Hot Migration

The resource utilization of modern data centers is low, with a large room for optimization. For instance, the average usage rate of most physical servers in a data center is only about 10

The primary challenge of hot migration lies in creating a consistent snapshot of the virtual machine state. In a data center composed of diverse hardware, the state of a virtual machine encompasses not only its CPU, memory, and local storage state, but also the state of hardware such as GPUs and network cards. These hardware components often lack efficient snapshot and recovery functions, necessitating the reloading of hardware drivers and initialization of the hardware's internal state on the new physical node, which results in high latency. The ClickNP framework discussed in Chapter

4 enables the snapshot and migration of the internal state of network elements, allowing programmable network cards written with the ClickNP framework to efficiently hot migrate. Technologies such as GPU-Direct RDMA can facilitate efficient transmission of GPU internal storage. For local storage, the concept of storage disaggregation can be employed, eliminating the need to wait for data migration to complete. Instead, the virtual machine's access request on the new node can be redirected to the original storage during the migration process.

6. Distributed Operating System for Edge-Cloud Convergence

Smart terminals (such as smartphones and PCs) and clouds (data centers) currently represent the two most significant types of computing and storage devices. The computing and storage capabilities of both the edge and the cloud are rapidly expanding, and with the advancement of 5G technology, the communication cost between the edge and the cloud will significantly decrease, while bandwidth and latency will see substantial improvements. Consequently, edge-cloud convergence will emerge as a crucial trend. On one hand, edge applications will be able to invoke cloud services and access cloud data at a finer granularity; on the other hand, technologies used for high-performance communication and computing on the cloud will gradually be applied to the edge. For instance, by deploying programmable network cards on the edge, power consumption for 5G network communication can be reduced, performance bottlenecks can be eliminated, the performance of accessing Flash storage can be enhanced, and the high-performance user-space socket technology discussed in Chapter 6 can also be applied on the edge.

The abstraction level of the distributed operating system is worth discussing. If abstracted at the level of the Linux operating system, compatibility will be the best, and automatic distributed processing of existing parallel programs can be achieved. However, the abstraction level of Linux system calls is relatively low. If developers do not explicitly provide more information, it is difficult to predict the resources that the application will access in the future, and the performance for some types of applications will be poor. One possible way to implement a distributed Linux operating system is remote system calls, i.e., for each process migrated to a remote system, a shadow process is retained locally; system calls on the remote system are captured, sent to the local shadow process and actually called, and the results of the system calls are sent to the remote system. The application process needs to wait for the remote system call to return, i.e., the delay of the system call is greatly increased, so the performance of this implementation scheme may be poor.

7.2.4 System Innovation

Envisioning the entire world as a large computer is the vision of Microsoft CEO Satya Nadella and the dream of many system researchers. The success of cloud computing has led data centers to absorb most of the computing and storage of the human world, and data centers can be seen as a large-scale computer composed of computing, memory, storage, and network and interconnection. NVIDIA CEO Jensen Huang[306], Google Engineering Vice President Luiz Barroso[4] and others have already seen data centers as large-scale computers. System innovation is to consider from a global perspective how various hardware and software components work efficiently and reliably together, and what kind of abstraction to provide to users.

1. Memory Disaggregation and Second-tier Memory Based on Programmable Network Cards

Memory disaggregation refers to the ability of a computer's CPU to freely and transparently share the memory of a remote computer efficiently, which can greatly increase the utilization of memory and reduce the cost of cloud computing platforms. Although the performance of the current data center network is far lower than the performance of the CPU accessing the host memory, fortunately, by utilizing the locality of memory access, if a part of the hot data is still local and the remaining data is accessed remotely, the bandwidth and latency requirements of the remote memory can be greatly reduced compared to the local memory. Research from the University of California, Berkeley, points out that in order to keep the performance difference between the system after memory disaggregation and the system using all local memory within 5%, the bandwidth needs to reach 40 Gbps, and the end-to-end round-trip delay needs to be no more than 3 to 5 microseconds, which is achievable by the current data center network.

Non-Volatile Memory (NVM) is a prominent research area in the field of memory and storage. Compared to traditional NAND Flash, Non-Volatile Memory boasts a significantly faster access speed. Although it cannot entirely replace DRAM in the short term, it is expected to substitute some of the conventional memory in the near future. Non-Volatile Memory offers advantages over DRAM such as lower cost, larger capacity, lower power consumption, and data retention after power off. However, it also has limitations such as slower access speed and limited write cycles. Non-Volatile Memory, serving as a storage tier between DRAM and NAND Flash, can be used to expand the capacity of DRAM memory and also function as fast persistent storage. The effective use of Non-Volatile Memory is currently a significant research direction.

Memory disaggregation and Non-Volatile Memory make up second-tier memory, which is slower but has a larger capacity than DRAM[307]. To expand memory capacity and minimize the impact on application performance, second-tier memory systems need to place hot data in local DRAM and cold data in disaggregated remote memory or Non-Volatile Memory. Most of the current memory disaggregation systems (such as Infiniswap[308]) and second-tier memory systems (such as Thermostat[309]) use page swapping. Firstly, page swapping needs to go through the operating system kernel, and each page swap in increases the kernel overhead by about 2.5 microseconds, while the allowable end-to-end access delay is only 3 to 5 microseconds. Secondly, the memory disaggregated to remote storage is generally cold data, and the access granularity of these data may be smaller than the page size, which is generally 4 KB, so transmitting a whole page not only wastes network bandwidth but also increases latency. Finally, the decision to swap pages in and out is made in software, making it difficult to accurately count the access frequency of each page.

The future research direction is based on programmable network card memory disaggregation and secondary memory. By using direct memory mapping instead of page swapping, the overhead of the operating system kernel is avoided, and the granularity of memory access is reduced from 4 KB pages to 64-byte cache lines. Local and remote memory are still in units of pages, relying on page tables to maintain mapping relationships. Programmable network cards can count the remote memory access of each page, thereby timely migrating hot data to local memory to avoid long-term performance impact.

There are a series of technical challenges in implementing memory disaggregation based on direct memory mapping based on existing CPU and PCIe architectures. Fortunately, CPU manufacturers have realized the same problem. We expect that with the implementation of host interconnection protocols such as CCIX, the programmable network card with direct memory mapping will achieve better throughput and latency with the CPU, and the direct memory mapping area can run all instructions like host memory.

2. Scalable total order communication based on data center networks

The latency in traditional data center networks is arbitrary, so messages cannot be guaranteed to be delivered in a consistent order. For instance, multiple shards of a distributed database send logs to multiple replicas. Each replica may receive logs from each shard in a different order. If not handled specially, this inconsistent order may break data consistency. The solution to this problem often introduces synchronization

overhead and complicates the design of distributed systems.

Total order communication provides an abstraction that guarantees that different receivers process messages from senders in a consistent order. Totally ordering (but unreliable) a set of messages can simplify and speed up many distributed applications, such as reducing conflicts in Multi-Version Concurrency Control (MVCC) protocols, speeding up distributed consensus protocols, implementing scalable log replication without central bottlenecks, early detection of TCP tail packet loss, and reducing the tail latency of scatter-gather mode Remote Procedure Call (RPC). For example, in recent years, the performance of distributed consensus protocols and distributed transactions has been greatly improved by improving the orderliness of transmission within the data center. Fast Paxos[310-313] protocol adopts a best-effort method to improve the orderliness of transmission. Speculative Paxos[314] and NOPaxos[315] use programmable switches as centralized sequence number generators or serialization points. NetPaxos[316-317] and[318] implement the traditional Paxos protocol in network switches. Eris[229] proposes to use network switches as sequence number generators, implement concurrency control in the network, and achieve fast transaction processing. The work on HotOS '19[319] proposes to build a synchronous, i.e., fixed network latency data center network, which can simplify the design of distributed systems.

Since the advent of distributed system research, total order broadcast and multicast issues have garnered significant attention. However, existing solutions are constrained by scalability or efficiency. One strand of research employs logically centralized coordination, such as centralized sequence number generators, or tokens circulated among senders or receivers. Recent research on the co-design of distributed systems and data center networks falls into this category. However, these centralized solutions struggle with scalability. Another strand of research employs entirely distributed coordination, such as exchanging timestamps before the receiver begins processing messages. This results in additional network communication overhead and latency, reducing system efficiency. Moreover, the semantics of multicast have a limitation that all receivers must receive the same message.

Compared to total order multicast, the application scope of Total-Order Message Scattering (TOMS) primitives is broader. Message scattering is a communication primitive where a host sends a group of (potentially different) messages to multiple hosts simultaneously. Message scattering is common in distributed systems. For instance, in distributed storage, a client writes metadata to one storage site and data to another storage site; concurrently, another client reads them. The consistency between metadata

and data necessitates these operations to be atomically scattered to two storage sites. Total order message scattering scatters a group of messages from one to many in the data center network, maintaining a linearizable order, and each message is delivered at most once.

To support improved scalability, and to expedite more distributed applications besides distributed transactions, scalable total order communication based on data center networks is an intriguing research direction. In the data center environment, the network topology is regular, and the switch generally has good programmability. Total order message scattering assigns work to each switch and terminal server, thereby achieving high scalability. The core design principle is to separate the processing of order information from message forwarding. To obtain order information, use programmable switches to aggregate order information in the network, forming the system's ”control plane”. On the ”data plane”, total order message scattering forwards messages as usual, and buffers and rearranges received messages at the receiver. The sender stamps an increasing timestamp on each group of scattered messages, and the receiver needs to deliver messages to the application in the order of timestamps. The control plane's order information provides a ”barrier” for the receiver that ”all messages received after this are later than a certain timestamp”, allowing it to deliver messages in the order of timestamps.

The preliminary work of this research has been published by collaborator Zuo Gefei at the ACM SOSP 2017 Student Research Competition (SRC)[320].

A significant challenge in the study of total order communication is reliability. Ensuring reliable total order communication in a network with packet loss and node failures is at least as difficult as the distributed consensus problem. It requires more complex fault tolerance and failure recovery mechanisms, and local failures can easily affect global communication efficiency. If the reliability of communication is not guaranteed, but only the order of received packets is guaranteed, the application scope will be significantly reduced. It must be combined with other traditional methods to ensure the correctness of the distributed system, but it can greatly reduce the out-of-order situation and improve efficiency.

Other aspects of distributed transactions can also benefit from the co-design with data center networks. Hyperloop[305] uses programmable network cards on storage nodes to write operations into the buffer of non-volatile memory, and immediately replies to the computing node with a confirmation message. The software on the storage node then asynchronously processes the write operations in the non-volatile memory.

This eliminates the delay of write operations waiting for the storage node software to process. Google Spanner[321] uses globally synchronized GPS clocks to achieve a high-performance database with cross-geographical area replication.

3. Database combining online transactions, batch and stream processing

Modern big data processing primarily has three paradigms: online transaction processing (OLTP), batch processing, and stream processing. Online transaction processing is used for transactions that require a faster response time and stronger consistency. Generally, each transaction only involves a small part of the dataset and updates are frequent. Batch processing is mainly used for offline data analysis, characterized by large amounts of data and computation. Stream processing is suitable for analysis tasks that require high real-time performance, and can incrementally update states and output results based on changes in data.

Traditionally, big data processing systems typically employ lambda architecture, wherein online transaction processing serves as the data source for batch and stream processing, and its generated data updates are synchronized to both the batch and stream processing components. The batch processing component periodically recalculates the results, while the stream processing component continuously updates the output based on the last batch processing results and the updated data from the stream input. Ultimately, the outputs of the batch processing and stream processing components are merged and delivered to the user. Firstly, the lambda architecture necessitates data analysts to explicitly segregate the data into online, batch, and stream components, write processing programs independently, and merge the results. This development process is complex and susceptible to inconsistency. Secondly, the stream processing in the lambda architecture may rely on the results of the last batch processing, and the delay of batch processing may result in inaccuracies in the results, and this delay is not necessarily required in terms of performance.

In recent years, HTAP (Hybrid Transactional and Analytical Processing) databases that amalgamate online transaction processing (OLTP) and offline data analysis processing (OLAP) transactions within the same database have gained popularity. HTAP databases address the delay issue from online transaction processing to batch processing analysis, but still do not support stream processing. Users need to explicitly rerun queries to obtain updated batch processing results, and the processing is based on the state of the database at the commencement of the query, and cannot reflect the real-time state of the database. The responsive databases proposed by the academic community, such as DBToaster, integrate online transaction processing and stream processing, but

all intermediate results are cached and processed incrementally, and the overhead is substantial. For instance, some types of batch processing are challenging to update incrementally, and a more performance-wise reasonable approach is to allow a certain delay in data updates.

The future research direction is a responsive database system that efficiently supports online transaction processing, offline data analysis, and stream processing simultaneously. Responsiveness is reflected in three aspects. First, each stored procedure transaction is responsive to the update operations of other parallel transactions. The updates of the basic tables are synchronized to the running offline data analysis and stream processing transactions. These running transactions save the appropriate intermediate state and update it incrementally. Therefore, each transaction is naturally serialized at the transaction completion time, that is, the query result of the stored procedure transaction reflects the real-time state of the database. Stream processing transactions are considered to be continuously running, and can report changes in the query results caused by database incremental updates to users in real time.

Secondly, the ”push” and ”pull” of the computational flow graph of transaction processing are responsive. In the internal computational flow graph of the database, each operator of the traditional database is ”pull” mode, that is, each time the user needs a query result, the computational flow graph is re-executed; while in stream processing and responsive databases, each operator is ”push” mode, that is, each time the data of the basic table is updated, all intermediate operator results are updated and saved until the final query result is updated, regardless of whether the user needs real-time updates. According to the user’s requirements for update timeliness, the database dynamically adjusts the ”push” and ”pull” modes of each operator in the computational flow graph, as well as the frequency of ”push”.

Finally, the physical data storage structure and index respond to data access patterns. The data update log of the basic table is used as the data source, and the row-based and column-based data storage structures are both caches, optimized for point queries and analytical queries respectively. Indexes are also considered caches. Views and intermediate results of analytical queries may also be cached. The database needs to adjust the choice of whether to cache or not based on the data access pattern, because caching can speed up read operations, but it adds a burden to write operations.

Bibliography

[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: Vol. 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. Big Sky, Montana, USA, 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: Vol. 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]// Vol. 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.[EB/OL]. 2019. https://linux.die.net/man/7/.

[44] PINKERTON J. Sockets direct protocol v1. 0 rdma consortium[Z]. 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: Vol. 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.co

m/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: Vol. 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[M]. Twente University Press, 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: Vol. 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: Vol. 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[Z]. 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: Vol. 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 convolutional 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] GUREVICH V. Programmable data plane at terabit speeds[EB/OL]. 2017. https://p4.org/assets/p4_d2_2017_programmable_data_plane_at_terabit_speeds.pdf.

[94] YOUSEF KHALIDI A N, CVP. Sonic: The networking switch software that powers the microsoft global cloud[EB/OL]. 2017. https://azure.microsoft.com/en-us/blog/sonic-the-networking-switch-software-that-powers-the-microsoft-global-cloud/.

[95] 阿里云. 阿里巴巴建成全球超大规模数据中心内 "RDMA 高速网"，以支撑人工智能科学计算[EB/OL]. 2019. https://yq.aliyun.com/articles/693509.

[96] 华为技术有限公司. 突破极限，开创未来：华为常务董事、产品投资评审委员会主任、ICT 战略与 Marketing 总裁汪涛在 2019 年全球分析师大会上的讲话[EB/OL]. 2019. https://www.huawei.com/cn/press-events/events/has2019/speech-by-wangtao-at-the-has2019.

[97] 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.

[98] XIAO W, XUE J, MIAO Y, et al. TuX2: Distributed graph computation for machine learning [C]//NSDI '17. 2017.

[99] LI J, MICHAEL E, PORTS D R. Eris: Coordination-free consistent transactions using in-network concurrency control[C]//Proceedings of the 26th Symposium on Operating Systems

Principles. ACM, 2017: 104-120.

[100] JONAS E, SCHLEIER-SMITH J, SREEKANTI V, et al. Cloud programming simplified: A berkeley view on serverless computing[A]. 2019.

[101] 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.

[102] KOPONEN T, CASADO M, GUDE N, et al. Onix: A distributed control platform for large-scale production networks.[C]//OSDI: Vol. 10. 2010: 1-6.

[103] VOELLMY A, AGARWAL A, HUDAK P. Nettle: Functional reactive programming for openflow networks[R]. YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE, 2010.

[104] FOSTER N, HARRISON R, FREEDMAN M J, et al. Frenetic: A network programming language[J]. ACM Sigplan Notices, 2011, 46(9): 279-291.

[105] 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.

[106] 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.

[107] 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.

[108] KAUFMANN A, PETER S, SHARMA N K, et al. High performance packet processing with flexnic[C]//ACM SIGARCH Computer Architecture News: Vol. 44. ACM, 2016: 67-81.

[109] 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.

[110] 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.

[111] NETWORKS B. Tofino: World's fastest p4-programmable ethernet switch asics[EB/OL]. 2019. https://www.barefootnetworks.com/products/brief-tofino/.

[112] Mellanox adapters programmer's reference manual (prm)[EB/OL]. 2019. http://www.mellanox.com/related-docs/user_manuals/Ethernet_Adapters_Programming_Manual.pdf.

[113] XILINX. Sdnet packet processor user guide[EB/OL]. 2017. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/UG1012-sdnet-packet-processor.pdf.

[114] 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.

[115] 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.

[116] LAN C. An architecture for network function virtualization[M]. UC Berkeley, 2018.

[117] NETWORKS F. F5 load balancer[EB/OL]. https://www.f5.com/services/resources/glossary/load-balancer.

[118] 3RD GENERATION PARTNERSHIP PROJECT (3GPP). 3gpp ts 23.501, system architecture for the 5g system (5gs), release 16[Z]. 2018.

[119] 3RD GENERATION PARTNERSHIP PROJECT (3GPP). 3gpp ts 38.300, technical specification group radio access network; nr; nr and ng-ran overall description, release 16[Z]. 2018.

[120] KOHLER E, MORRIS R, CHEN B, et al. The click modular router[J]. ACM Transactions on Computer Systems (TOCS), 2000, 18(3): 263-297.

[121] 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.

[122] 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.

[123] PALKAR S, LAN C, HAN S, et al. E2: a framework for nfv applications[C]//Proceedings of the 25th Symposium on Operating Systems Principles. ACM, 2015: 121-136.

[124] CLARK D D, JACOBSON V, ROMKEY J, et al. An analysis of tcp processing overhead[J]. IEEE Communications magazine, 1989, 27(6): 23-29.

[125] BOYD-WICKIZER S, CLEMENTS A T, MAO Y, et al. An analysis of linux scalability to many cores.[C]//OSDI: Vol. 10. 2010: 86-93.

[126] 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.

[127] KAUFMANN A, PETER S, ANDERSON T E, et al. Flexnic: Rethinking network dma.[C]//HotOS '15. 2015.

[128] 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.

[129] 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.

[130] 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.

[131] 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.

[132] 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.

[133] 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.

[134] 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.

[135] 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.

[136] LU G, GUO C, LI Y, et al. Serverswitch: a programmable and high performance platform for data center networks.[C]//Nsdi: Vol. 11. 2011: 2-2.

[137] TECHNOLOGIES M. Mellanox bluefield(tm) smartnic vpi[EB/OL]. 2019. http://www.mellanox.com/page/bluefield_smartnic_vpi?mtag=smartnic_vpi1.

[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] 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.

[165] 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.

[166] 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/.

[167] OUYANG J. Xpu: A programmable fpga accelerator for diverse workloads[C]//2017 IEEE Hot Chips 29 Symposium. 2017.

[168] 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.

[169] RUSSELL R. virtio: towards a de-facto standard for virtual i/o devices[J]. ACM SIGOPS

Operating Systems Review, 2008, 42(5): 95-103.

[170] NISHTALA R, FUGAL H, GRIMM S, et al. Scaling memcache at facebook[C]//NSDI '13. 2013.

[171] ALIZADEH M, YANG S, SHARIF M, et al. pfabric: Minimal near-optimal datacenter transport[C]//ACM SIGCOMM Computer Communication Review: Vol. 43. ACM, 2013: 435-446.

[172] PFAFF B, PETTIT J, KOPONEN T, et al. The design and implementation of open vswitch. [C]//NSDI: Vol. 15. 2015: 117-130.

[173] 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.

[174] 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.

[175] 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.

[176] 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.

[177] 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.

[178] 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.

[179] 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.

[180] SHERRY J, GAO P X, BASU S, et al. Rollback-recovery for middleboxes[C]//ACM SIGCOMM Computer Communication Review: Vol. 45. ACM, 2015: 227-240.

[181] 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.

[182] 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.

[183] 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.

[184] 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.

[185] SCHELLE G, GRUNWALD D. Cusp: a modular framework for high speed network applications on fpgas[C]//Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays. ACM, 2005: 246-257.

[186] 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.

[187] 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.

[188] YOSHINO T, SUGAWARA Y, INAGAMI K, et al. Performance optimization of tcp/ip over 10 gigabit ethernet by precise instrumentation[C]//SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. IEEE, 2008: 1-12.

[189] Ethernet switch series[Z]. 2013.

[190] Openflow-data plane abstraction networking software[Z]. 2014.

[191] WIRBEL L. Xilinx sdnet: A new way to specify network hardware[Z]. Xilinx whitepaper, 2014.

[192] FEIST T. Vivado design suite[J]. Xilinx White Paper, 2012.

[193] CZAJKOWSKI T S, NETO D, KINSNER M, et al. Opencl for fpgas: Prototyping a compiler [C]//Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA). The Steering Committee of The World Congress in Computer Science, Computer, 2012: 1.

[194] STRATIX V. Device handbook, volume 1: Device interfaces and integration[J]. Altera, June, 2012.

[195] 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.

[196] The OpenCL Specifications ver 2.1.[Z].

[197] Dell networking s6000 spec sheet[EB/OL]. http://dell.com/.

[198] 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.

[199] 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.

[200] Strongswan ipsec-based vpn[Z].

[201] Linux virtual server[Z].

[202] MOON S W, REXFORD J, SHIN K G. Scalable hardware priority queue architectures for high-speed packet switches[J]. IEEE Transactions on Computers, 2000.

[203] 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.

[204] 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.

[205] FITZPATRICK B. Distributed caching with memcached[J]. Linux journal, 2004, 2004(124): 5.

[206] INTEL D. Data plane development kit[Z]. 2014.

[207] GHARACHORLOO K, GUPTA A, HENNESSY J. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors: Vol. 20[M]. ACM, 1992.

[208] HAN S, JANG K, PARK K, et al. Packetshader: a GPU-accelerated software router[C]// ACM SIGCOMM Computer Communication Review: Vol. 40. ACM, 2010: 195-206.

[209] 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.

[210] NARULA N, CUTLER C, KOHLER E, et al. Phase reconciliation for contended in-memory transactions.[C]//OSDI '14: Vol. 14. 2014: 511-524.

[211] SUTTER H. The free lunch is over: A fundamental turn toward concurrency in software[J]. Dr Dobb's journal, 2005, 30(3): 202-210.

[212] ESMAEILZADEH H, BLEM E, AMANT R S, et al. Power challenges may end the multicore era[J]. Communications of the ACM, 2013, 56(2): 93-102.

[213] 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.

[214] LIU S, DU Z, TAO J, et al. Cambricon: An instruction set architecture for neural networks [C]//Proceedings of the 43rd International Symposium on Computer Architecture. IEEE Press, 2016: 393-405.

[215] NORMAN P. JOUPPI E A. In-datacenter performance analysis of a tensor processing unit [M]//Proceedings of the 44th International Symposium on Computer Architecture (ISCA). 2017.

[216] MELLANOX. Mellanox infiniband adapters[EB/OL]. 2017. http://www.mellanox.com/page/infiniband_cards_overview.

[217] GREENBERG A. SDN for the cloud[C]//Keynote in the 2015 ACM Conference on Special Interest Group on Data Communication. 2015.

[218] 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.

[219] 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.

[220] BEREZECKI M, FRACHTENBERG E, PALECZNY M, et al. Many-core key-value store [C]//Green Computing Conference and Workshops (IGCC), 2011 International. IEEE, 2011: 1-8.

[221] 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.

[222] 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.

[223] LAVASANI M, ANGEPAT H, CHIOU D. An FPGA-based in-line accelerator for memcached [J]. IEEE Computer Architecture Letters, 2014, 13(2): 57-60.

[224] 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.

[225] ISTVAN Z, SIDLER D, ALONSO G, et al. Consensus in a box: Inexpensive coordination in hardware[C]//NSDI '16. 2016: 425-438.

[226] 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.

[227] FIRESTONE D. VFP: A virtual switch platform for host SDN in the public cloud[C]// NSDI '17. Boston, MA, 2017: 315-328.

[228] 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.

[229] LI J, MICHAEL E, PORTS D R K. Eris: Coordination-free consistent transactions using in-network concurrency control[C]//SOSP '17. Shanghai, China, 2017.

[230] ATIKOGLU B, XU Y, FRACHTENBERG E, et al. Workload analysis of a large-scale key-value store[C]//ACM SIGMETRICS Performance Evaluation Review: Vol. 40. ACM, 2012: 53-64.

[231] PERRY J, OUSTERHOUT A, BALAKRISHNAN H, et al. Fastpass: A centralized zero-queue datacenter network[C]//ACM SIGCOMM Computer Communication Review: Vol. 44. ACM, 2014: 307-318.

[232] ZHU Y, KANG N, CAO J, et al. Packet-level telemetry in large datacenter networks[C]//ACM SIGCOMM Computer Communication Review: Vol. 45. ACM, 2015: 479-491.

[233] 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.

[234] COUNCIL T. tpc-c benchmark, revision 5.11[M]. Feb, 2010.

[235] PAGH R, RODLER F F. Cuckoo hashing[J]. Journal of Algorithms, 2004, 51(2): 122-144.

[236] HERLIHY M, SHAVIT N, TZAFRIR M. Hopscotch hashing[C]//International Symposium on Distributed Computing. Springer, 2008: 350-364.

[237] BONWICK J, et al. The slab allocator: An object-caching kernel memory allocator.[C]//USENIX summer: Vol. 16. Boston, MA, USA, 1994.

[238] 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.

[239] 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.

[240] 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.

[241] JIN X, LI X, ZHANG H, et al. NetCache: Balancing Key-Value Stores with Fast In-Network Caching[C]//SOSP '17. Shanghai, China, 2017.

[242] 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.

[243] 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.

[244] 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.

[245] 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.

[246] LI X, SETHI R, KAMINSKY M, et al. Be fast, cheap and in control with switchkv.[C]// NSDI. 2016: 31-44.

[247] 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.

[248] KEJRIWAL A, GOPALAN A, GUPTA A, et al. SLIK: Scalable low-latency indexes for a key-value store[C]//USENIX ATC '16. 2016.

[249] LIN X, CHEN Y, LI X, et al. Scalable kernel tcp design and implementation for short-lived connections[C]//ACM SIGPLAN Notices: Vol. 51. ACM, 2016: 339-352.

[250] HAN S, MARSHALL S, CHUN B G, et al. Megapipe: A new programming interface for scalable network i/o.[C]//OSDI: Vol. 12. 2012: 135-148.

[251] 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.

[252] REESE W. Nginx: the high-performance web server and reverse proxy[J]. Linux Journal, 2008, 2008(173): 2.

[253] PANDA A, HAN S, JANG K, et al. Netbricks: Taking the v out of nfv.[C]//OSDI. 2016: 203-216.

[254] REESE W. Nginx: the high-performance web server and reverse proxy[J]. Linux Journal, 2008, 2008(173): 2.

[255] CARLSON J L. Redis in action[M]. Manning Publications Co., 2013.

[256] 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.

[257] CORBET J. Kaiser: hiding the kernel from user space[EB/OL]. 2017. https://lwn.net/Articl es/738975/.

[258] Thread pools in nginx boost performance 9x![EB/OL]. 2015. https://www.nginx.com/blog/t hread-pools-boost-performance-9x/.

[259] 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.

[260] KAMINSKY A K M, ANDERSEN D G. Design guidelines for high performance rdma systems[C]//2016 USENIX Annual Technical Conference. 2016: 437.

[261] IHM S, PAI V S. Towards understanding modern web traffic[C]//Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference. ACM, 2011: 295-312.

[262] JANG K, HAN S, HAN S, et al. Sslshader: Cheap ssl acceleration with commodity processors.[C]//NSDI. 2011.

[263] DAVID M. How skbs work[J]. URL http://vger kernel org/~ davem/skb html.

[264] 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.

[265] ZHU Y, ERAN H, FIRESTONE D, et al. Congestion control for large-scale rdma deployments[C]//ACM SIGCOMM Computer Communication Review: Vol. 45. ACM, 2015: 523-536.

[266] MITTAL R, DUKKIPATI N, BLEM E, et al. Timely: Rtt-based congestion control for the datacenter[C]//ACM SIGCOMM Computer Communication Review: Vol. 45. ACM, 2015: 537-550.

[267] THOMPSON K, MILLER G J, WILDER R. Wide-area internet traffic patterns and characteristics[J]. IEEE network, 1997, 11(6): 10-23.

[268] CORBET J. Large receive offload[EB/OL]. 2007. https://lwn.net/Articles/243949/.

[269] PFEFFERLE J, STUEDI P, TRIVEDI A, et al. A hybrid i/o virtualization framework for rdma-capable network interfaces[C]//ACM SIGPLAN Notices: Vol. 50. ACM, 2015: 17-30.

[270] 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.

[271] 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.

[272] 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.

[273] 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.

[274] SOARES L, STUMM M. Flexsc: Flexible system call scheduling with exception-less sys-

tem calls[C]//Proceedings of the 9th USENIX conference on Operating systems design and implementation. USENIX Association, 2010: 33-46.

[275] THADANI M N, KHALIDI Y A. An efficient zero-copy i/o framework for unix[M]. Sun Microsystems Laboratories, 1995.

[276] 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.

[277] CORBET J. Zero-copy networking[EB/OL]. 2017. https://lwn.net/Articles/726917/.

[278] DUNKELS A. Design and implementation of the lwip tcp/ip stack[J]. Swedish Institute of Computer Science, 2001, 2: 77.

[279] 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.

[280] Apache http server[EB/OL]. 2019. https://httpd.apache.org/.

[281] Fastcgi process manager for php[EB/OL]. 2019. https://php-fpm.org.

[282] Python wsgi http server for unix[EB/OL]. 2019. https://gunicorn.org.

[283] vsftpd[EB/OL]. 2019. https://security.appspot.com/vsftpd.html.

[284] WIKIPEDIA. iscsi host adapter (hba)[EB/OL]. 2019. https://en.wikipedia.org/wiki/Host_adapter.

[285] 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.

[286] HINTJENS P. Zeromq: messaging for many applications[M]. " O'Reilly Media, Inc.", 2013.

[287] RABBITMQ A. Rabbitmq-messaging that just works[J]. URL: https://wwwrabbitmqcom, 2017.

[288] 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.

[289] CORPORATION I. Intel 64 and ia-32 architectures software developer manual, volume 3 [Z]. 2019.

[290] CORBET J. Tcp connection repair[EB/OL]. 2012. https://lwn.net/Articles/495304/.

[291] ARISTA. Arista 7060x series[EB/OL]. 2019. https://www.arista.com/en/products/7060x-series.

[292] rpclib - modern msgpack-rpc for c++[EB/OL]. 2019. http://rpclib.net.

[293] 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.

[294] 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.

[295] MITTAL R, SHPINER A, PANDA A, et al. Revisiting network support for rdma[A]. 2018.

[296] 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.

[297] 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.

[298] 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.

[299] 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.

[300] 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.

[301] 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.

[302] 吴军. 浪潮之巅[M]. 电子工业出版社, 2011.

[303] PERRY T S. David patterson says it's time for new computer architectures and software languages[EB/OL]. 2018. https://spectrum.ieee.org/view-from-the-valley/computing/hardware/david-patterson-says-its-time-for-new-computer-architectures-and-software-languages.

[304] WU C, FALEIRO J M, LIN Y, et al. Anna: A kvs for any scale[M]. ICDE'18.

[305] 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.

[306] 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-mel

lanox-future-datacenters-will-belike-high-performance-computers/.

[307] 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.

[308] 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.

[309] AGARWAL N, WENISCH T F. Thermostat: Application-transparent page management for two-tiered main memory[C]//ACM SIGARCH Computer Architecture News: Vol. 45. ACM, 2017: 631-644.

[310] LAMPORT L. Fast paxos[J]. Distributed Computing, 2006, 19(2): 79-103.

[311] 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.

[312] 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.

[313] PEDONE F, SCHIPER A. Optimistic atomic broadcast[C]//International Symposium on Distributed Computing. Springer, 1998: 318-332.

[314] PORTS D R, LI J, LIU V, et al. Designing distributed systems using approximate synchrony in data center networks.[C]//NSDI. 2015: 43-57.

[315] 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.

[316] 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.

[317] DANG H T, CANINI M, PEDONE F, et al. Paxos made switch-y[J]. ACM SIGCOMM Computer Communication Review, 2016, 46(1): 18-24.

[318] DANG H T, BRESSANA P, WANG H, et al. Network hardware-accelerated consensus[A]. 2016.

[319] 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.

[320] ZUO G. Near-optimal total order message scattering in data center networks[M]//SOSP Student Research Competition 2017. 2017.

[321] 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.

Acknowledgements

I would like to express my gratitude to my alma mater, the University of Science and Technology of China, and Microsoft Research Asia for providing me with valuable learning opportunities. These allowed me to engage with the world-leading programmable network card experimental platform and data center application scenarios during my joint doctoral training, and to conduct cutting-edge research in the field of data center systems.

I am grateful to my supervisor at the University of Science and Technology of China, Professor Chen Enhong. Over the past six years, starting from my senior year of undergraduate studies, Professor Chen has consistently supported my joint training internship at Microsoft, helping me to determine my research direction and doctoral topic. During my doctoral studies, Professor Chen assisted me in formulating my training plan, funded my participation in international academic conferences, recommended me for numerous awards such as the Microsoft Scholar Award and National Scholarship, and helped me revise my proposal and dissertation. Whether in life or in scientific research, Professor Chen has done everything possible to support and assist me, allowing me to focus on academic issues in research. The modest academic achievements I have made are not only due to Professor Chen's guidance on the overall direction, but also inseparable from his silent support and assistance. I sincerely thank Professor Chen Enhong for his support and help.

I would like to thank my supervisor at Microsoft Research Asia, Dr. Zhang Lintao, a principal researcher. During the three years we worked together, Dr. Zhang led me into the field of system research, not only teaching me the knowledge and thinking methods of computer systems, but also training my ability to think independently, identify problems, and lead research. Dr. Zhang guided me to complete my second research project, KV-Direct, and among several possible innovation points in the field of key-value storage, he selected the acceleration of memory data structure access, which best highlights the role of programmable network cards. During the implementation process in collaboration with my classmate Ruan Zhenyuan, he helped us refine and summarize system design and optimization techniques, revised the paper and presentation from beginning to end, and published it in a top academic conference in the system field, allowing me to have good research results in the middle of my doctorate. Subsequently, Dr. Zhang gave me enough space to think independently and explore freely, broadened

my horizons, cultivated my overall view of the system, and helped me recruit interns to cooperate in implementing my innovations, researched several new topics, and had papers accepted by conferences like SIGCOMM. Whether it's internal or external reports, Dr. Zhang is always able to understand and raise profound questions keenly. He gave me valuable opportunities to tell stories and receive feedback in front of renowned professors, reminding me not to get lost in technical details and forget the audience's background and the overall picture of the system. Dr. Zhang guided me to clarify the main line of research during my doctorate, understand the deeper connotations and broader extensions of my research, and the gap with high-impact work. Dr. Zhang took me on my first trip to the United States at Microsoft's headquarters, and often shared with me his experiences in system research, the workplace, and life. He is my good teacher and helpful friend. I sincerely thank Dr. Zhang Lintao for his guidance and help.

I would like to thank my former supervisor at Microsoft Research Asia, former senior researcher Dr. Tan Kun. Dr. Tan is my scientific research enlightenment mentor, who not only taught me the knowledge and thinking methods of computer networks, but also taught me the methodology of "analytical thinking" in scientific research and the attitude of "discarding the false and retaining the true" in scholarship. Dr. Tan established the research direction of the network research group in the field of data centers and built a world-leading data center network and programmable network card experimental platform. Dr. Tan guided me to complete my first research project, ClickNP, hand in hand. He proposed the academic problem of accelerating network functions with programmable network cards, determined the basic framework and technical route of high-level language programming, helped me write papers, and published them in top academic conferences in the network field, giving me a high starting point in research. Dr. Tan allowed me to explain to researchers in different fields to exercise my overall view, and on the other hand, he paid attention to details, discussing code style, experimental data, and presentation sentences at group meetings. When my thinking was too divergent, he timely let me converge to draw conclusions, allowing me to continuously produce efficiently. After the completion of the ClickNP project, when I was torn between FPGA programming and the direction of systems and networks, Dr. Tan guided me to position myself in the field of systems and focus on projects that can have a real impact. Dr. Tan also shared with me many philosophical thoughts on research and is also my good teacher and helpful friend. I sincerely thank Dr. Tan Kun for his guidance and help.

I would like to thank the teachers and classmates who co-authored papers with me. In addition to my supervisors, they also include researchers and intern students at Microsoft Research Asia. In the ClickNP project, I would like to thank Dr. Luo Layong, a researcher, for his early exploration in the field of FPGA programming framework, Peng Yanqing from Shanghai Jiaotong University, Luo Renqian from the University of Science and Technology of China for their cooperation in developing compilers, network elements, and applications, Dr. Xu Ningyi, a senior researcher, Dr. Xiong Yongqiang, a senior researcher, Dr. Cheng Peng, a researcher, for their discussions and help, and He Tong from Beijing University of Aeronautics and Astronautics for the development of communication pipelines between FPGA and CPU. In the KV-Direct project, I would like to thank Ruan Zhenyuan, a co-first author from the University of Science and Technology of China, for designing and implementing the system with me, Xiao Wencong from Beijing University of Aeronautics and Astronautics for writing the introduction, Lu Yuanwei from the University of Science and Technology of China for his discussions and help, Dr. Xiong Yongqiang, a senior researcher, and Dr. Andrew Putnam, a chief hardware engineer, for their discussions and support for the hardware experimental environment. In the SocksDirect project, I would like to thank Cui Tianyi, a co-first author from the University of Science and Technology of China, for designing and implementing the system with me, Dr. Bai Wei, a researcher, for writing the introduction and sorting out the logic of the paper, and Wang Zibo from the University of Science and Technology of China for implementing the first version of the system prototype. In the TOMS project, I would like to thank Zuo Gefei from the University of Science and Technology of China for designing and implementing the system with me, and Dr. Bai Wei, a researcher, for discussing and revising most of the paper content with me. I would like to thank Dr. Ren Jinglei, a researcher from the system group at Microsoft Research Asia, Dr. Chen Liang, Wang Yang from the University of Electronic Science and Technology, Taekyung Heo from KAIST, and Lu Yuanwei, Xiao Wencong, Ruan Zhenyuan, Cui Tianyi, Li Yishuai, Cao Shijie and other classmates for their cooperation and help in the yet-to-be-published research projects. I would also like to give special thanks to Dr. Chen Guo, a researcher at Microsoft Research Asia, and Dr. Zhang Jiansong for their multiple paper collaborations. Dr. Chen Guo's FUSO was my first paper in my academic career as the fourth author, and his rigorous academic attitude has benefited me for life. Dr. Zhang Jiansong guided me in FPGA development, and I was fortunate to participate in the Feniks FPGA OS and SSD acceleration projects proposed by Dr. Zhang Jiansong. I sincerely thank all the co-authors of my doctoral thesis for

their guidance and help. Without you, I could not have achieved these results.

In addition to the co-authors of this thesis, I would also like to thank Dr. Lidong Zhou, the deputy director of Microsoft Research Asia, Dr. Yunxin Liu and Dr. Chen Zhang from the Systems Group, Dr. Ran Shu and Dr. Zhixiong Niu from the Networking Group, and Associate Professor Xiaoliang Wang from Nanjing University for their guidance and assistance in my research projects. I am grateful to Andrew Putnam, Tanj Bennett, Derek Chiou, Qi Luo, Daniel Firestone from Microsoft's headquarters in the United States, and Wei Cui, Wenqiang Wang and other colleagues from Microsoft Research Asia for their support and help. I would like to thank Dr. Yongguang Zhang, a chief researcher, for giving me the opportunity to be jointly trained by the University of Science and Technology of China and Microsoft Research Asia. I appreciate the valuable comments from the anonymous reviewers of the conference papers and the doctoral thesis review committee. I am grateful to the teachers, experts, and classmates who gave me valuable guidance in academic conferences and interviews.

I would like to thank the teachers, classmates, and friends I met at the University of Science and Technology of China and Microsoft Research Asia. Especially the "Rice Ball" students who are the core of the joint doctoral training at Microsoft Research Asia, thank you for accompanying me through the colorful doctoral life. I am grateful to the colleagues of the Academic Cooperation Department of Microsoft Research Asia for their help in my doctoral training and internship life. I would like to thank Mr. Huanjie Zhang and all the partners of the Linux User Association of the University of Science and Technology of China, for giving me the opportunity to develop and maintain various network services during my undergraduate years, which honed my technical ability in computer systems and the psychological quality of handling faults, and cultivated my interest in cloud computing. I am grateful to my high school classmate Xiaoshikang for teaching me how to make websites, and friends in the blockchain field for cultivating my interest in customized hardware. I would like to give special thanks to my girlfriend. Not only did we get to know each other through research and co-authored papers, but she also strongly supports my academic research and career development planning. My girlfriend has made me more mature in life and filled our life together with happiness. She is like a beam of light, illuminating everything with brightness.

Finally, I would like to thank my parents, grandparents, and family. It is you who silently support me from behind, giving me a strong backing. I am grateful for the kindness of my parents and grandparents for more than twenty years, as well as the help and support from other family members. Without you, I could not have achieved what

I have today. You are the greatest, thank you for your dedication.

Publications

Published First-Author Papers

These three publications constitute the main technical contributions of this thesis.

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.
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.
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.

Other Published Papers

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.
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.
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.
4. 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).
5. 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).

6. 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).
7. 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).
8. 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.
9. 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.

Papers to be Published

1. B. Li, G. Zuo, W. Bai, L. Zhang, “Efficient and Scalable Total-Order Message Scattering in Data Center Networks”.