Data Plane Development Kit

A packet accelerator framework

A Seminar Report Submitted

in the Partial Fulfilment of the Requirements

for the Degree of

Master Of Technology

by

Prateek Agarwal

Computer Science and Engineering

Indian Institute of Technology Bombay

April, 2020

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my thesis guide Dr. Mythili Vutukuru for her guidance, invaluable suggestions and critical remarks for improving the quality of work.

Prateek Agarwal

April, 2020

Contents

List of Figures **vii**

1 Background **1**

1.1 Introduction 1

1.2 Traditional Linux Stack 2

1.2.1 Working 2

1.2.2 Limitations 3

1.3 Solutions/Alternatives 4

1.4 Organization of the Report 5

2 Data plane Development Kit (DPDK): Introduction **6**

2.1 An overview 6

2.2 Run-to-Completion vs. Pipeline : Models of Execution 7

2.3 Overcoming the Limitations 8

2.3.1 Polling 8

2.3.2 Zero-Copy 8

2.3.3 Flexible Processing 8

2.4 Salient Features 9

2.4.1 Superpages/Huge Pages 9

2.4.2 Cooperative Multiprocessing 9

2.4.3 Pinned Memory 9

2.4.4 Dynamic Memory Management 10

2.4.5 Lockless data structures 10

2.4.6 Better Algorithms 10

3 Poll Mode Driver **11**

3.1 Introduction . . . 11

3.2 Background: Interaction between a Driver and a Device . . . 12

3.2.1 Transfer of Packets . . . 12

3.3 Design of important PMDs . . . 14

3.3.1 igb_uio . . . 14

3.3.2 vfio-pci-driver . . . 14

3.3.3 Software PMDs . . . 15

3.4 Configuration and Device Statistics . . . 15

3.4.1 Configuration . . . 16

3.4.2 Device Statistics . . . 16

4 Core Libraries **17**

4.1 Mempool . . . 17

4.1.1 Use Case . . . 17

4.1.2 Main Features . . . 17

4.2 Mbuf Library . . . 18

4.2.1 Use Case . . . 18

4.2.2 Design Principles . . . 19

4.3 Ring Library . . . 20

4.3.1 CAS Review . . . 21

4.3.2 Key Idea . . . 21

4.3.3 Demonstration: Single Consumer Dequeue . . . 23

4.3.4 Demonstration: Multiproducer Enqueue . . . 23

4.4 Timer Library . . . 25

5 Application Libraries **26**

5.1 Hash Library . . . 26

5.1.1 Use Case: Flow classification . . . 26

5.1.2 Abstract Data Type (ADT) operations . . . 26

5.1.3 Cuckoo Hashing . . . 27

5.1.4 Implementation Details in DPDK . . . 30

5.1.5 Extensible Bucket . . . 32

5.2 Membership Library 32
5.2.1 Use Case: Load Balancing 33
5.2.2 Abstract Data type (ADT) operations 33
5.2.3 Bloom Filters 33
5.2.4 Hash Table Set Summaries (HTSS) 34
5.3 Longest Prefix Match (LPM) Library 35
5.3.1 Abstract Data Type (ADT) Operations 35
5.3.2 Implementation: IPv4 36
5.3.3 Implementation : IPv6 38
5.4 Summary 39

6 Frameworks using DPDK **40**
6.1 Vector Packet Processor 40
6.1.1 Role of DPDK 40
6.1.2 Vectorized Processing and related concepts 41
6.1.3 VPP Principle 41
6.1.4 Features of VPP 42
6.1.5 Code optimizations 43
6.2 Transport Layer Development Kit (TLDK) 44
6.2.1 Features 45

7 Related Work and Comparisons **46**
7.1 Netslices 46
7.2 PF_RING and PF_RING ZC 47
7.3 netmap 48
7.3.1 Working Principles of *netmap* 48
7.4 Closure 49

Bibliography **51**

List of Figures

1.1 Time-critical operation on data packets. 1
1.2 Lifecycle of a packet in Linux. 3
1.3 Network header in *sk_buff* [1] 4

2.1 DPDK core functions. 7
2.2 DPDK Application Layer functions. 7
2.3 (a) Run-to-Completion model (b) Pipeline Model [2] 8

3.1 DPDK: Packet movement 11
3.2 DMA descriptors [3] 13
3.3 Layout of a receive queue [3] 13
3.4 Virtio Queues [3] 14
3.5 Control APIs 15

4.1 Mempool: Memory management operations [4] 18
4.2 (a)rte_mbuf (b) multisegmented rte_mbuf [4] 19
4.3 PseudoCode: CAS instruction [5] 21
4.4 (a) Initialization of local variables. (b) Object is consumed after the *cons_reserver* is moved. (c) *cons_eater* is moved and the dequeue operation is completed. All figures are drawn from [4] and are modified suitably. 22
4.5 (a) Initialization of local variables. (b) Step 2 CAS succeeds for P_1 and fails for P_2. P_2 rereads the global variables (step1) (c) P_2 CAS succeeds and objects are stored in the queue (d) Step3 CAS succeeds for P_1 and fails on P_2. (e) Step 3 CAS succeeds on P_2 All figures are drawn from [4] and are modified suitably. 24

5.1 Search operation in Cuckoo Hash. 27
5.2 Deletion in Cuckoo Hash 28
5.3 Insertion in Cuckoo Hash 28
5.4 (a)Before insertion of the element C (b) After insertion 29
5.5 Failed Insertion in Cuckoo Hash 30
5.6 Structure of an Entry in Hash Table 30
5.7 Cuckoo Hash with Extensible Bucket 32
5.8 Bloom Filter Representation 34
5.9 LPM: IPv4 Implementation 36
5.10 LPM: IPv6 Implementation 39

6.1 VPP Demonstration [6] 41
6.2 Multiloop and prefetching with N = 2. 43
6.3 Branch Prediction 44
6.4 Packet movement in DPDK [7] 45

7.1 Netslices [8] 46
7.2 Vanilla PF_RING [9] 47
7.3 Netmap Rings [10] 48

Chapter 1

Background

1.1 Introduction

The data on the current networks - like the Internet, is transmitted in the form of packets. The packets contain the message and metadata headers. These headers carry the information required for carrying these packets between two nodes. There are two kinds of packets - data packets and control packets. Data packets carry the useful data or payload. Control packets carry the information required for maintaining the network infrastructures. The rate of data packets processing

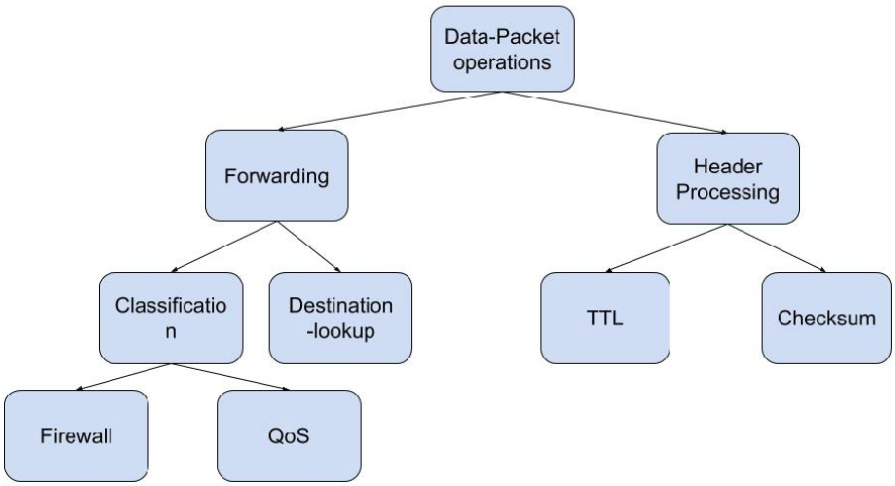


Figure 1.1: Time-critical operation on data packets.

is performance-critical as they form the lion's share of all the transmitted packets. Figure 1.1 shows the time-critical operations performed on the data packets. Check-

sum verification for error detection and decrement of the Time-to-Live field (to avoid indefinite looping of a packet) are examples of header processing functions. The forwarding operations include destination lookup for forwarding packet to the specific output port and packet classification. The destination lookup involves the longest prefix match for the IP address on a router. The packet classification is primarily required for filtering packets for security reasons in a firewall and for enforcing the quality-of-service policies like prioritizing packets from one host over the other.

Communication links have become faster and cheaper over time. The number of communication links interacting with a single physical system has also increased. The faster packet processing at a node is required to make high-speed data transmission possible. Section 1.2 discusses working and problems with the traditional Linux network stack. Section 1.3 discusses possible approaches to mitigate the problem and section 1.4 gives the layout of the report.

1.2 Traditional Linux Stack

Linux has a network stack designed for general-purpose packet processing. It is no match for current line rates. Section 1.2.1 discusses the lifecycle of a packet in the system. Section 1.2.2 discusses the limitations.

1.2.1 Working

When a packet is available on the NIC, the kernel driver copies the packet in the RAM by direct memory access (DMA) without the involvement of the processor (CPU). An interrupt is raised after the packet is copied in a newly allocated structure kernel structure ***sk_buff***. ***sk_buff*** has fields for storing the payload, and protocol headers like IPv4, IPv6, TCP, etc. This interrupt signals the kernel to process the packet. The kernel processes the protocol headers for error detection and packet forwarding operations. The packet is copied into user-space if the destination application is running on the same system or for packet classification operations such as QoS or Firewall.

On the transmit side, the payload is copied into the ***sk_buff*** structure after a *write* system call is invoked by the user application. The address of the ***sk_buff*** is

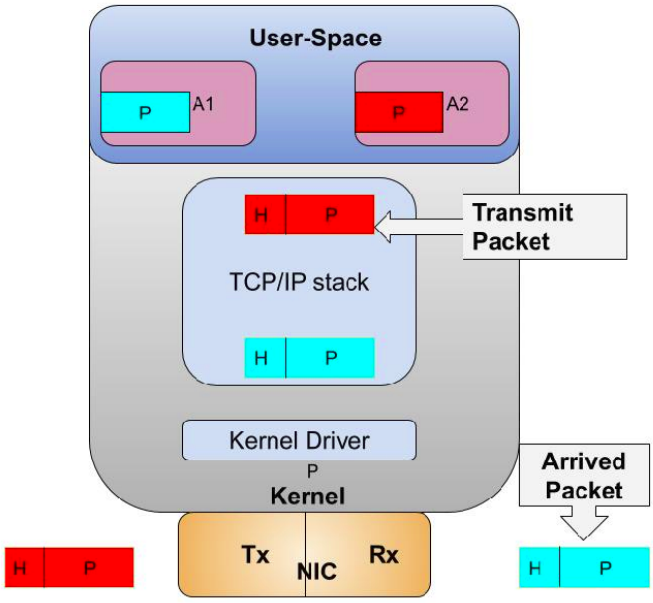


Figure 1.2: Lifecycle of a packet in Linux.

stored in a DMA descriptor and NIC is notified by changing its device register (3.2). After the packet is transmitted, NIC raises an interrupt for CPU to deallocate the structure. Figure 1.2 depicts the lifecycle of Rx and Tx packets in Linux.

1.2.2 Limitations

The main limitations are interrupt processing overhead, multiple memory copies of the packet, and the handling of the *sk_buff* structure.

Interrupt Overhead

An interrupt is raised for the receipt and transmission of every packet. The interrupt processing is a time-consuming activity as it involves context switching from user mode to kernel mode and kernel mode to user mode once the interrupt is processed. The context switching entails saving of processor registers, user process stack, etc. so that the process can be restored once the process is scheduled again. If the packet arrives at a high rate, a lot of time is spent in context switching and little useful work is performed. This results in low throughput and dropping of packets.

Multiple Memory Copies

The packet is copied twice: from NIC to kernel space, and from kernel space to user-space on receipt of a packet and vice versa on transmit.

Handling of *sk_buff*

The buffer is allocated and deallocated on the receipt and transmission of every packet. Linux kernel is a general-purpose solution. The same kernel can handle multiple protocols on the same layer. It introduces matching overhead. For example, if the router is only IPv4 compatible, checking the protocol every time introduces an overhead. Figure 1.3 shows the different network layer protocol headers that can be stored in a *sk_buff* structure.

```
union {
        struct iphdr    *iph;
        struct ipv6hdr  *ipv6h;
        struct arphdr   *arph;
        unsigned char   *raw;
} nh;     // <--- Network header address
```

Figure 1.3: Network header in *sk_buff* [1]

1.3 Solutions/Alternatives

There are three possible approaches for faster packet processing [11–13]:

1. Build special-purpose hardware and design customized software for it.
2. Network-Processors: Offloading the bottleneck functions on the network processor present on the NIC and continue using the CPU for other operations.
3. Build fast packet processing solutions entirely in software.

The hardware-based solutions i.e. (1) and (2) are expensive. The existing hardware can not be used for (1). The software solutions (3) are cheaper and existing commercial off the shelf (COTS) hardware can be used. The software solutions are compatible and flexible to match with the advancements in microprocessor technology. This seminar is the study of the Data Plane Development Kit by Intel (DPDK). DPDK is a software solution for faster packet processing.

1.4 Organization of the Report

The layout of this report is as follows. Chapter 2 discusses the main features of DPDK that helps in overcoming the limitations of the Linux stack. Chapter 3 describes the user-space network driver "Poll Mode Drivers (PMD)" developed as a part of the project. Chapter 4 focusses on the core libraries which enable fast packet I/O. Chapter 5 is on the libraries provided to the applications for performing network functions like the destination lookup. Chapter 6 describes the frameworks built over DPDK. Chapter 7 concludes by providing a brief overview and comparison of the alternatives with the DPDK.

Chapter 2

Data plane Development Kit (DPDK): Introduction

A brief overview of DPDK is provided in section 2.1. Section 2.2 discusses the main models of packet processing solutions. In chapter 1, the limitations of the kernel network stack were pointed out. Section 2.3 describes how the DPDK overcomes these limitations. Section 2.4 concludes with additional features/optimizations introduced by DPDK.

2.1 An overview

DPDK is a set of software libraries built for fast packet processing. DPDKs main concern is the low level I/O - by-passing kernel stack and providing packets in userspace quickly. The packets and their headers are processed in the user-space. Note that this does not hold true for Linux stack- TCP/IP processing, checksum calculation, decrement of TTL field occurs in kernel space. Only the payload or application data reaches user-space. The kernel is entirely by-passed in DPDK. The kernel's core functions are unavailable. These functions include memory allocation/deallocation for storing packets, timer facilities (Chapter 4) and a network driver (Chapter 3) for interacting with network interface cards (NICs) (see Figure 2.1).

DPDK also provides additional libraries for processing packet headers and performing forwarding operations in user-space. The main libraries among them are the

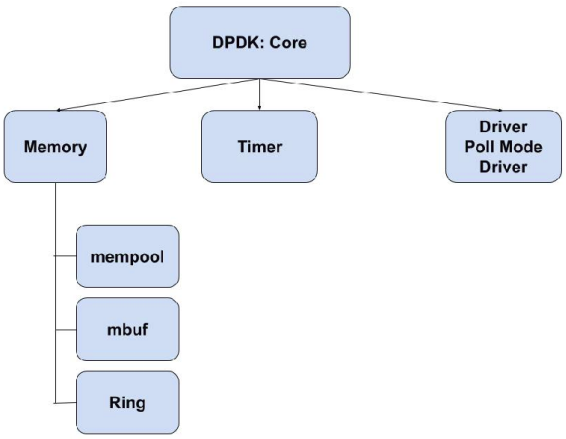


Figure 2.1: DPDK core functions.

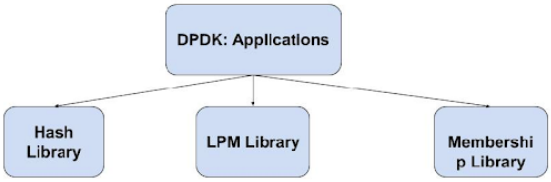
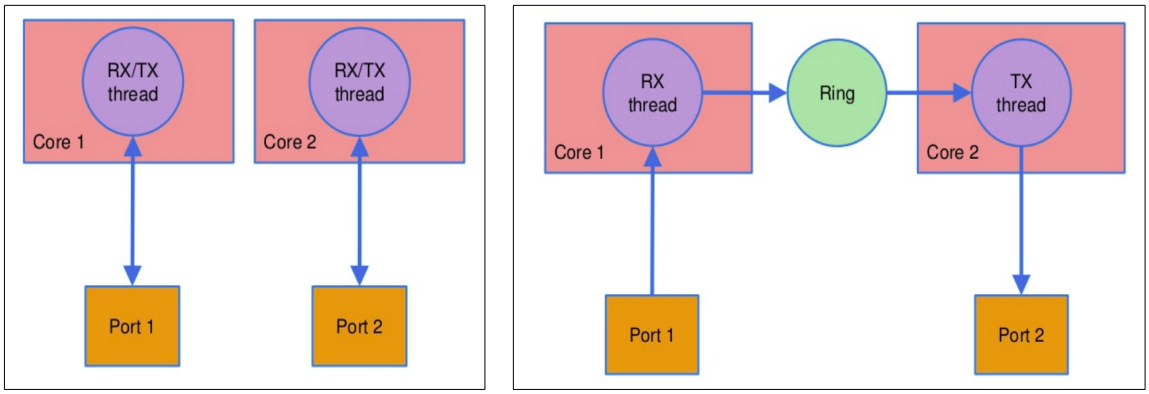


Figure 2.2: DPDK Application Layer functions.

Hash Library, LPM library, and Membership library (Chapter 5). The hash library and LPM libraries help in forwarding the packet to the appropriate core or the VM. The membership library helps in quickly deciding whether the packet belongs to the machine.

2.2 Run-to-Completion vs. Pipeline : Models of Execution

DPDK provides run-to-completion (Fig 2.3(a)) and pipeline (Fig 2.3(b)) models of execution. In the run-to completion model, packet is processed completely on a single assigned core. In the pipeline model, all packets are sent to a single control core. This core assigns packet to different cores which process these packets and transmit them further if required. Run-to-completion model is the widely used mode of execution.



(a) (b)

Figure 2.3: (a) Run-to-Completion model (b) Pipeline Model [2]

2.3 Overcoming the Limitations

2.3.1 Polling

After a time-out, multiple packets are fetched from the NIC and copied into the RAM i.e. the packets are accepted in polling mode. Note that the Linux stack is interrupted on the arrival of every single packet (section 1.2.1). When the packets are received in bulk, the interrupt overhead (section 1.2.2) is distributed over a number of packets which results in a low per-packet processing overhead or amortization of the overhead.

2.3.2 Zero-Copy

No additional copying of packets from kernel space to user space takes place. Packets are directly copied into user-space. Additional copying in the Linux stack hogs both space in the RAM and time (1.2.2).

2.3.3 Flexible Processing

The customized applications can be developed for handling a specific protocol. If the router is only IPv6 compatible, we can build applications optimized for handling IPv6 packets only. This will result in a better performance when compared to the general-purpose Linux stack (1.2.2).

2.4 Salient Features

2.4.1 Superpages/Huge Pages

DPDK requests huge pages during the initialization or run-time if required. Individual requests of applications or drivers are met from these huge pages. The huge pages come in sizes of 2 MB and 1 GB. The default size is generally 4 KB. For faster translation from virtual addresses to actual physical addresses of pages, a hardware component called Translation Look-aside Buffer (TLB) cache is present in memory management units. It does not have more than 128 entries in a COTS machine. Huge pages help in mapping large areas of the RAM directly in TLB. The misses in TLB are rare, leading to better performance.

2.4.2 Cooperative Multiprocessing

A primary process manages all the hardware resources and secondary processes can attach to the primary process. These processes have shared access to the memory, CPU and other resources. So the context switch or preemption is avoided between processes [14].

2.4.3 Pinned Memory

The hugepages/superpages allocated to DPDK are not swapped back to disk [14]. This means that the virtual address to physical address mapping stays intact and does not change during runtime. This has two main advantages-

- **Explicit NUMA-awareness**- NUMA stands for Non-Uniform Memory Access. Multicore architectures are designed such that certain areas of main memory are closer to every core. Accessing this local region of memory is relatively fast for a core than other regions [14]. All data structures in DPDK are allocated in NUMA-aware local region.

- **DMA transfer** Direct Memory Access (DMA) technique is used to transfer data/packets from NIC to RAM without the involvement of CPU. Physical address of the memory region is required for DMA transfer. The virtual address to physical address translation is a time consuming process.

As the physical addresses do not change, DPDK asks for physical addresses of the pages during intialization. These physical addresses are used for DMA transfer directly.

2.4.4 Dynamic Memory Management

Earlier versions of DPDK (17.11 or before) had static memory allocation during intialization. The memory had to be reserved before the start. This lead to inefficient utilization in case of excess memory and memory bottlenecks which hinder performance in case of insufficient memory.

DPDK version 18.11 (stable) and beyond provide dynamic memory management. The excess memory can be returned (reserved) back to (from) the kernel during runtime leading to better efficiency in time and space.

2.4.5 Lockless data structures

Multiple cores are performing computations simultaneously in a modern-day multi-core architecture. These cores require access to a common memory region or variable for accessing a common resource like timers, NIC buffers. Locks are generally used to keep these variables in a consistent state. The contention among different cores to acquire locks causes a significant overhead. The data structures in DPDK are made lockless with the help of hardware instructions like Compare and Swap (CAS) (section 4.3.1).

2.4.6 Better Algorithms

The algorithm techniques used for packet forwarding applications are conducive for network applications working on multi-core architectures. Cuckoo hashing (Chapter 5) is a case in point. It is designed for constant-time lookup in the worst case.

Chapter 3

Poll Mode Driver

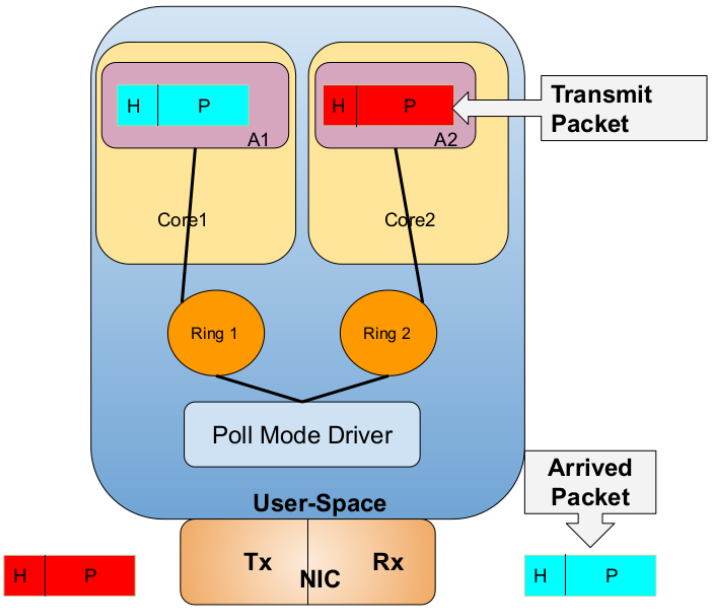


Figure 3.1: DPDK: Packet movement

3.1 Introduction

Poll Mode Drivers (PMD) are the user space network drivers. The device drivers are a piece of software which knows how to interact with an external device like hard disk, CD-ROM, and NIC among others. The network drivers are the device drivers which interact with network devices like NICs. DPDK has drivers compatible with 1 Gigabit, 10 Gigabit and 40 Gigabit Intel NICS. DPDK also has drivers compatible with *virtio* - a virtual device used for interacting with virtual machines.

3.2 Background: Interaction between a Driver and a Device

The device and the driver store data and control information like statistics, debugging or error status at two places-

- Base Address Registers (BARs) on the device (NICs in our case). These are primarily used to store control information.
- Main Memory Locations. The data (packets) are primarily stored here.

The data is transferred with the help of

- **Memory-Mapped IO (MMIO)** The BARs or RAM location used for data transfer are placed in the shared address space. The user process or driver can communicate with BARs or shared memory by reading or writing on these addresses. Similarly the device (NICs) can write (read) data packets to (from) these shared memory regions using DMA transfer.
- **x86 IO ports** Data can also be transferred by writing to registers using *x86 IN and OUT instructions*. This technique is not used with modern NICs. However it is used along with MMIO in virtio implementation by writing memory addresses in IO registers in igb_uio drivers (section 3.3.1).

3.2.1 Transfer of Packets

Hardware NICs

NICs have multiple receive and transmit queues. Incoming traffic is distributed among different receive queues by using hashing technique. Multiple transmit queues generally combine into one queue per interface before transmitting packets. A simple model of one receive queue and one transmit queue is assumed in the following description. The driver allocates space for a circular array for holding pointers to the physical address of the data packets. These pointers are called *DMA* descriptors (see Figure 3.2). Packets are moved by passing these descriptors between NIC and driver with the help of a head (owned by NIC) and a tail (owned by driver) pointer.

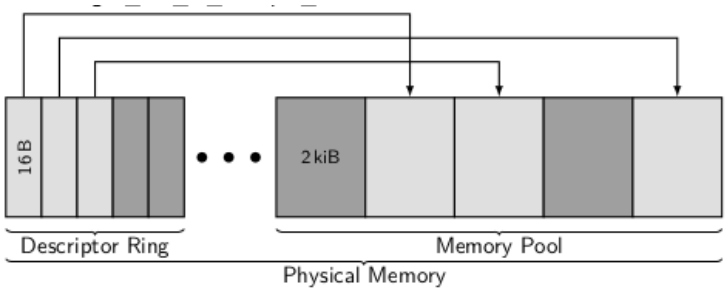


Figure 3.2: DMA descriptors [3]

A buffer table (Figure 3.3) which maps physical addresses to virtual addresses is also maintained. This mapping is required on packet reception as the user process uses virtual addresses. The packets are processed in batches.

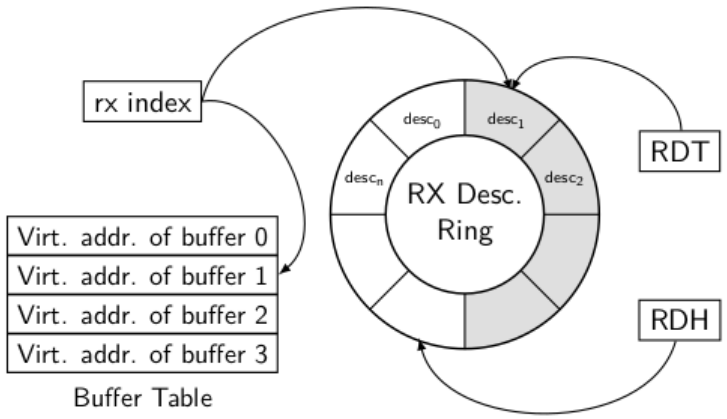


Figure 3.3: Layout of a receive queue [3]

Transmit side is similar except for the asynchronous transfer of packets on to the links. The process returns after filling the packets on the ring. When the packets are transmitted later, the driver is notified by the movement of pointers on the circular buffer. The sent packets are subsequently freed or erased and returned back to memory pools (4.1).

virtio queues

virtio queues are used to transmit data packets between host machine and virtual machines. Packets are received/transmitted by storing them in DMA buffers (section 3.2.1). However virtio queues (that store DMA descriptors) consists of

two rings- *available* and *used* (see Figure 3.4). The **available** ring is used for receipt/transfer of packets. Once the packets are processed, the descriptors are kept in the **used** ring . The network driver then clears the descriptors and which can then be reused. The **available** queue indices are written in device registers using *IN/OUT* x86 instructions. The virtio queues are not exclusively used for packet

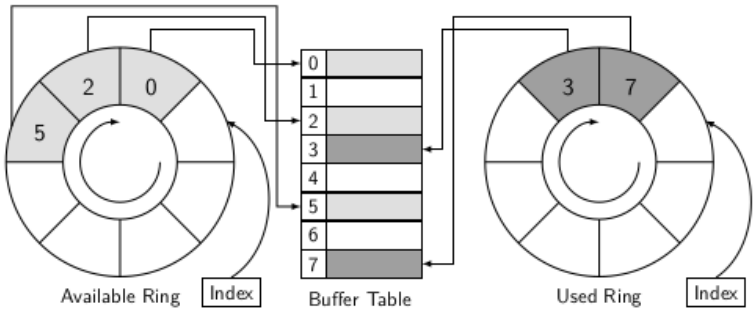


Figure 3.4: Virtio Queues [3]

transfers. For example, virtio queues are also used for reading files from disk. The transmit function prepends a header to the network data. This header helps the driver in differentiating network packets from other devices' data.

3.3 Design of important PMDs

3.3.1 igb_uio

DPDK uses *uio* kernel subsystem to build *igb_uio* [15] driver. This driver is used for both virtio and hardware NICs [3]. *uio* provides full access to the shared registers and shared memory by memory mapping these regions into files. These files are located in *sysfs* virtual-filesystem (VFS). *sysfs* treats an external device as a file.

3.3.2 vfio-pci-driver

DPDK uses *vfio* kernel subsystem to build *vfio-pci* driver. vfio provides IOMMU support for virtualized environments. IOMMU is a memory translation unit present on NICs which directly maps guest virtual addresses to host physical addresses. This enables NICs to perform DMA directly and guest VMs can interact with devices using guest virtual addresses.

3.3.3 Software PMDs

DPDK also has PMDs which do not require kernel subsystems like uio or vfio. These PMDs are built over packet capture libraries like *libpcap* and *AF_PACKET* [16] sockets. The packet capture framework allows DPDK to work with any hardware by providing raw packets to the driver.

3.4 Configuration and Device Statistics

DPDK PMDs provide APIs for configuring and collecting device statistics from NICs (Figure 3.5).

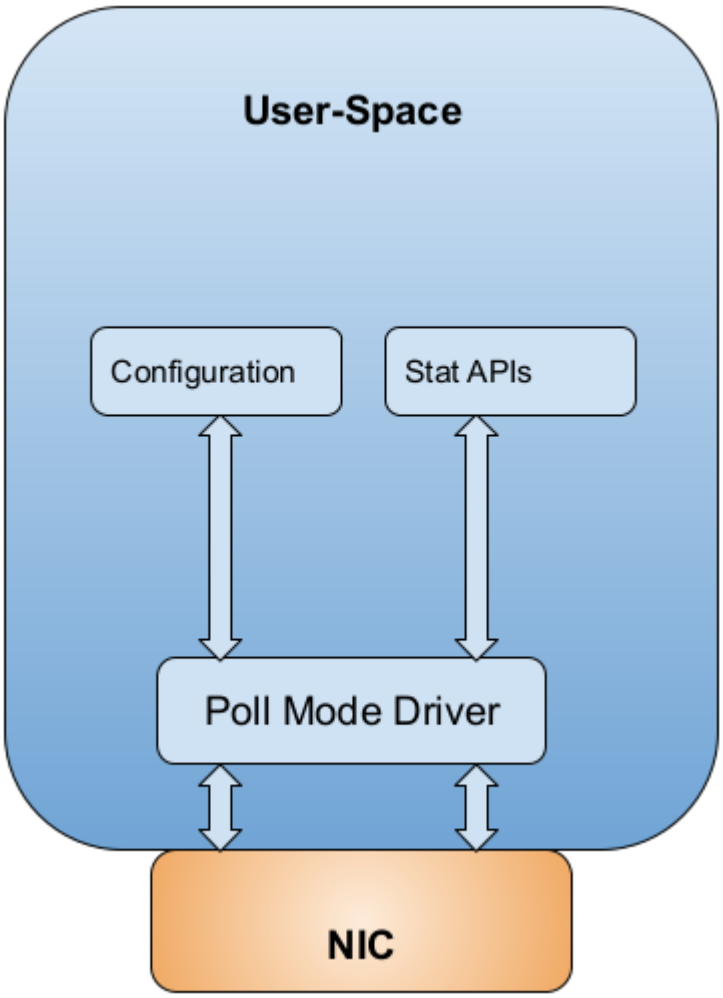


Figure 3.5: Control APIs

3.4.1 Configuration

Depending on the capability of hardware NICs, the NICs can be configured for

- **Offloading Calculations** - Checksums, CRC checks, VLAN tag processing among others.
- **Directing Traffic** - Destination lookup and making forwarding decisions like dropping packets, diverting it to a specific queue, packet encapsulations for tunnel offloads (IPv6 packet inside a IPv4 packet for transmitting on an IPv4 compatible link) and so on.

This is performed with the help of APIs which are present in Flow Library [4].

3.4.2 Device Statistics

PMDs provide APIs for collecting statistics on the network traffic. The statistics include but are not limited to number of incoming packets, number of transmitted packets, packet drop rate etc. The extended statistics API is used for getting statistics which are unique to a device.

It is the responsibility of application developer to use these statistics or configure the hardware if required. The PMDs act as the translators between application and the hardware.

Chapter 4

Core Libraries

DPDK's core libraries provide the minimum functions required to enabe fast I/O of the packets. The libraries in sections 4.1, 4.2, 4.3 provide the support for **memory management** allocation, movement and freeing of packets through the stack. The timer library 4.4 provides timing facilities in the absence of kernel timers.

4.1 Mempool

Mempool library manages the allocation of fixed size buffers. A mempool stores addresses of free mbufs in a circular ring managed by the Ring Library (section 4.3).

4.1.1 Use Case

Mempool library is primarily used for storing fixed size buffers- *mbufs* (see section 4.2). It can also be used by any application that requires fast moving fixed-size buffers like logging services.

4.1.2 Main Features

Local Cache

When a new payload (in mbuf) arrives on a core or buffers are required for creating a transmit packet, the buffer pointers can be moved to/from a local cache in bulk. This reduces contention over the common ring in mempool. Figure 4.1 shows the memory management operations performed when a local cache is maintained. The size of

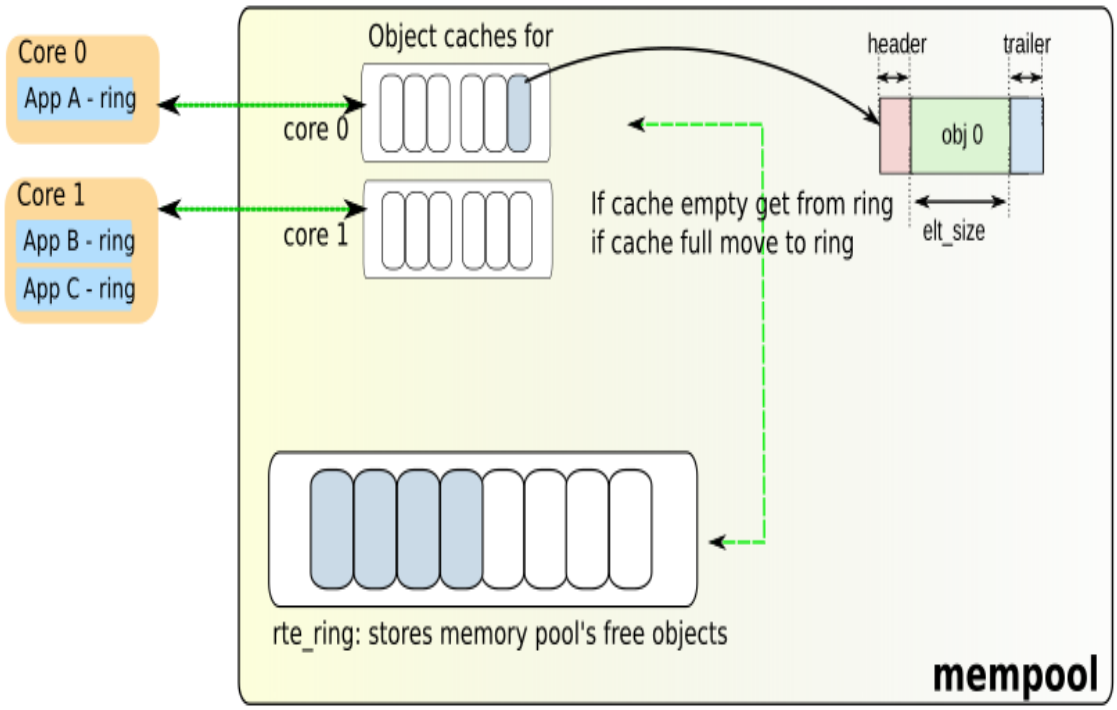


Figure 4.1: Mempool: Memory management operations [4]

the cache can be defined at compile time (`CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE`) and hence is static after intitialization.

Cookies and Stats

The debug mode is enabled by setting the `CONFIG_RTE_LIBRTE_MEMPOOL_DEBUG` flag. In debug mode, cookies or guards are added at the beginning and the end of allocated blocks. This helps in catching *buffer overflow* errors. Statistics of get/put from mempool per core are also logged in the mempool structure.

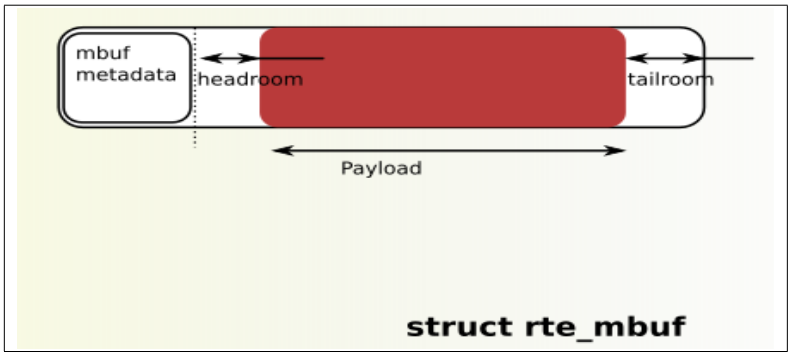
4.2 Mbuf Library

The *mbuf* library defines the buffer structures to hold packets. These structures correspond to ***sk_buff*** (Section 1.2.2) in Linux stack.

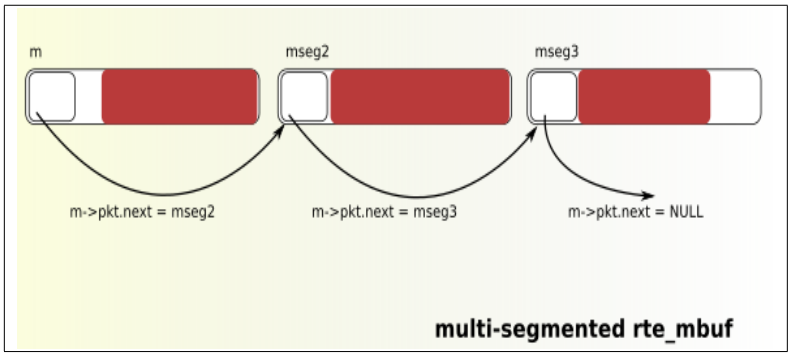
4.2.1 Use Case

Mbufs are used for carrying network packets.

4.2.2 Design Principles



(a)



(b)

Figure 4.2: (a)rte_mbuf (b) multisegmented rte_mbuf [4]

Anatomy of a *mbuf*

mbuf structure has two main components.

- **Metadata header** contains information required for handling the packets. Note that this header is different from network headers like TCP, IP etc. This header contains length of the packet, address of the payload, origin *mempool*, address of the next *mbuf* structure in case of large packets and other required information. This header also contain flags indicating what computations are performed in hardware due to hardware offloading (section 3.4.1)

- **Payload** contains the packet in entirety. No headers are processed with the exception of offloaded operations to hardware (like checksum, CRC checks etc).

Alignment Considerations

mbuf structure is designed with alignment considerations in mind. The metadata header fits in atmost two cache lines (of 64 byte each). Padding is provided at both the start (headroom in Figure 4.2(a)) and end of the payload (tailroom in Figure 4.2(a)) to make the payload and further packets cache aligned respectively.

Figure 4.2 shows the design of mbuf structures.

Multi-segmented Structures

A large-sized packet is fragmented and stored in multiple mbufs. These mbufs are stored in a linked list with metadata header of all but the last buffer containing a pointer to the next mbuf (Figure 4.2(b)).

Direct and Indirect Buffers

A direct buffer contains the actual payload and is complete i.e. it is self-contained. An indirect buffer has a valid metadata header and it points to the payload of another direct buffer. An indirect pointer cannot point to another indirect buffer. Each direct buffer also maintains a reference count of the number of indirect buffers pointing to it. The mbuf is freed and returned to its origin mempool when the count drops to zero. This technique is similar to the concept of hard links of a file in Unix filesystems.

This is used for fast duplication of a packet and its reuse.

4.3 Ring Library

Ring library provides functions for management of circular queues or rings. The rings contain pointers to fixed size buffers. These rings are used by mempool library (section 4.1) to handle mbufs (section 4.2). Rings are designed for concurrent access by multiple producers and multiple consumers. An example - Multiple threads copying data from NIC to userspace rings act as producers and application threads serving these packets act as consumers. Rings have fixed maximum size. Bulk enqueue-dequeue operations are also possible.

Rings are lockless and are designed with the help of compare and swap (CAS) instruction. The section 4.3.1 reviews the use of CAS intruction in lockless techniques.

4.3.1 CAS Review

CAS instruction provides for comparison and swapping of values. If the comparison succeeds, the values are swapped, otherwise the CAS operation is unsuccessful. The entire CAS operation is atomic i.e. the instruction cannot be preempted in between.

```
function cas(p : pointer to int, old : int, new : int) returns bool {
    if *p ≠ old {
        return false
    }
    *p ← new
    return true
}
```

Figure 4.3: PseudoCode: CAS instruction [5]

4.3.2 Key Idea

There are 4 global pointers - $prod\_reserver$,$prod\_filler$, $cons\_reserver$, $cons\_eater$. Producer threads modify $prod\_reserver$ and $prod\_filler$. Consumer threads modify $cons\_reserver$ and $cons\_eater$.

$prod\_reserver$

The $prod\_server$ is moved ahead to reserve locations in ring before filling the buffer pointers.

$prod\_filler$

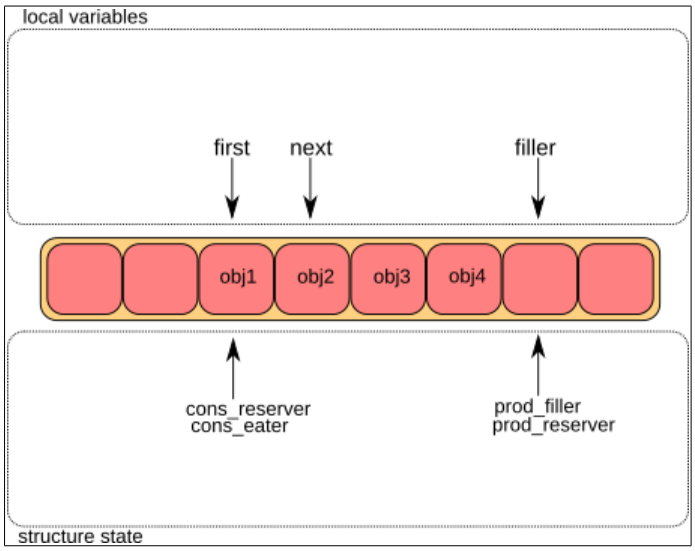
The $prod\_filler$ is moved ahead only when the reserved locations are filled. The consumers can use the filled buffers up to (and not including) the location pointed by the $prod\_filler$.

cons_reserver

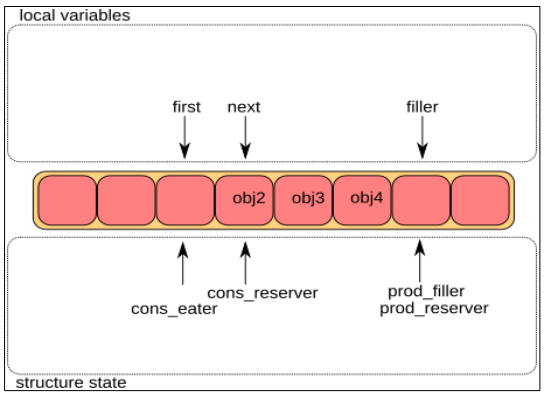
The *cons_reserver* is moved ahead to reserve locations in ring. The consumer thread can only process buffer pointers present in the reserved space.

cons_eater

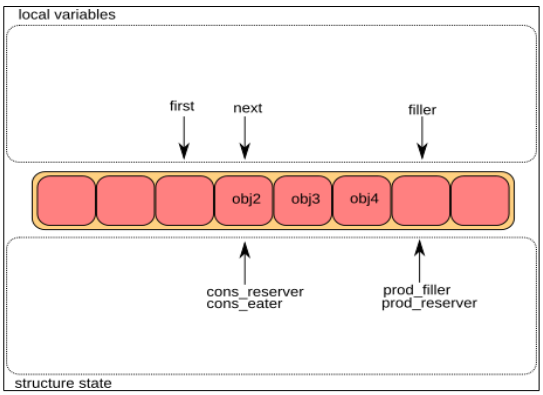
The *cons_eater* is moved ahead only when the reserved locations are processed or emptied. The producers can fill the buffers up to (and not including) the location pointed by the *cons_eater*.



(a)



(b)



(c)

Figure 4.4: (a) Initialization of local variables. (b) Object is consumed after the *cons_reserver* is moved. (c) *cons_eater* is moved and the dequeue operation is completed. All figures are drawn from [4] and are modified suitably.

4.3.3 Demonstration: Single Consumer Dequeue

This sectiond describes the dequeue operation by a single consumer thread. The consumer thread P has three **local** variables - $first$, $next$, and $filler$. The step by step algorithm is as follows (all operations are modulo S, where S is the size of the queue.

1. The consumer copies **global** *cons_reserver* and *prod_filler* pointers in $first$ and $filler$ respectively. If the queue does not have the requested n objects i.e. if $first + n \geq filler$, an error is returned. Otherwise, $next = first + n$. The $next$ points to the location after the n objects are reserved for processing by the consumer thread.

2. **Reserve Space** The consumer changes the global *cons_reserver* to equal $next$.

3. After the objects are processed, the consumer will change the *cons_eater* to equal $next$.

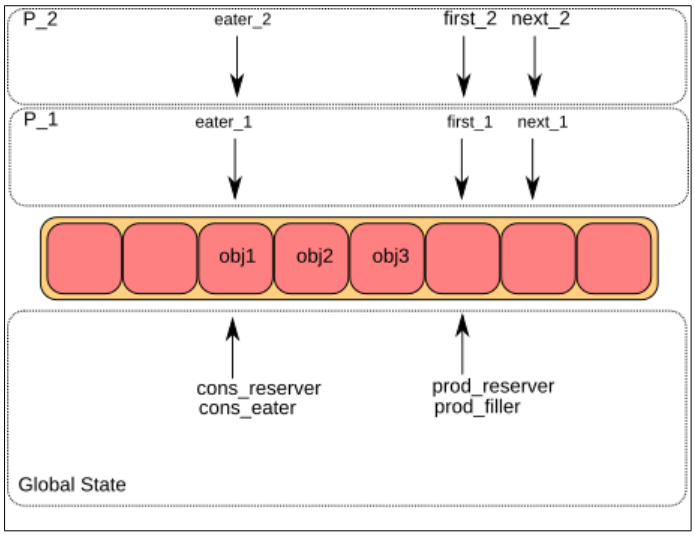
Figure 4.4 shows the dequeue operation by a single consumer. The movement of pointers is performed with the help of CAS instructions. Their significance is observed in the demonstration of multi-producer enqueue (4.3.4).

4.3.4 Demonstration: Multiproducer Enqueue

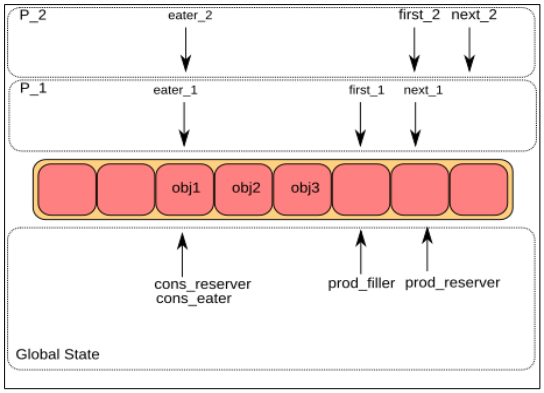
This section describes the enqueuing operation when multiple producers are enqueuing simultaneously. The steps in the process are carried out for 2 producers which can be easily generalized to any m producers. Let the two producers be P_1 and P_2. Each producer P_i thread has three **local** variables - $first_i$, $next_i$, and $eater_i$.

The step by step algorithm is as follows (all operations are modulo S, where S is the size of the queue.)

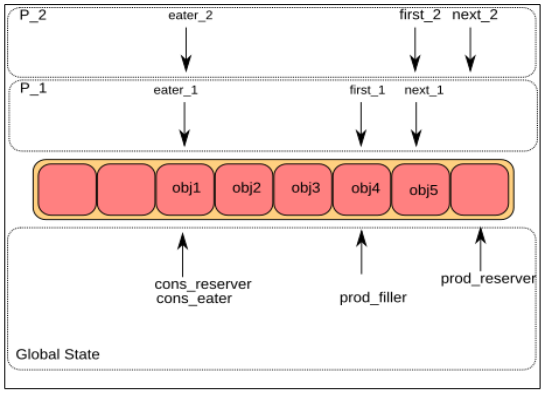
1. Both producers copy **global** *prod_reserver* and *cons_eater* pointers in $first_i$ and $eater_i$ respectively. If the queue does not have the capacity to accomodate the requested n objects i.e. if $first_i + n \geq eater_i$, an error is returned. Otherwise, $next_i = first_i + n$. The $next_i$ points to the location after all the objects are filled by the respective thread.



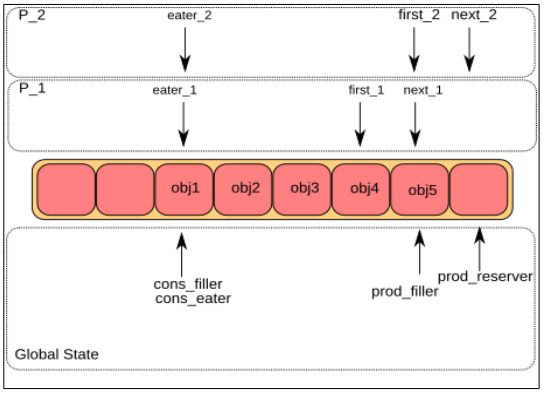
(a)



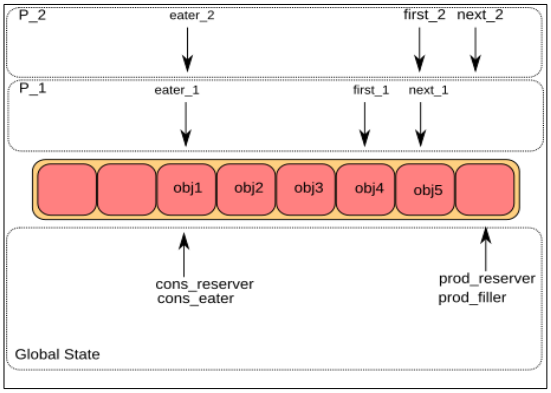
(b)



(c)



(d)



(e)

Figure 4.5: (a) Initialization of local variables. (b) Step 2 CAS succeeds for P_1 and fails for P_2. P_2 rereads the global variables (step1) (c) P_2 CAS succeeds and objects are stored in the queue (d) Step3 CAS succeeds for P_1 and fails on P_2. (e) Step 3 CAS succeeds on P_2 All figures are drawn from [4] and are modified suitably.

2. **Reserve Space** Both producers try to change the global *prod_reserver* to equal $next_i$ using CAS instruction. The *prod_reserver* is compared with the local $first_i$ in CAS instruction. If the CAS fails on some thread, the process is restarted from step 1. The successful producer proceeds on the next step.

   Eventually both the core succeeds in reserving space.

3. After the objects are filled, the producers will try to change the $prod\_filler$ to equal $next_i$. The $prod\_filler$ is compared with the local $first_i$ in CAS instruction. The CAS is repeatedly tried. Eventually the enqueuing operation is completed on both the threads.

Figure 4.5 shows the enqueue operation by two producers. Each producer enqueue 1 object i.e. $(n = 1)$ each in the ring.

4.4 Timer Library

In the Linux stack, the kernel provides software timers based on the hardware timers present in modern processors- time stamp counter (TSC) and high precision event timer (HPET). The timer facilities are required in DPDK for calling asynchronous functions such as garbage collectors. The main features of Timer library are:

- Timers can be one-time or periodic.
- Timers can use either 64-bit Time Stamp Counter (TSC) register in x86 microprocessors using *RDTSC* instruction or high precision event timer (HPET) comparators that are present in current multicore architecture.
- These software timers can be loaded on one core and exectued on another.
- The timers are maintained on a per core basis.

Chapter 5

Application Libraries

DPDK framework provides many libraries for the forwarding and the flow classification applications. This chapter describes the implementation and applications of three major libraries: hash library, LPM library, and membership library.

5.1 Hash Library

Hash Library is used for faster table look-up based on the hashing technique. Hash Library is implemented by using Cuckoo Hashing Algorithm [17]. Hash library is designed for multi-threaded architecture.

5.1.1 Use Case: Flow classification

The packets on the same connection/flow share the 5 tuples: source IP address, source port, destination IP, destination port and protocol. Single flow packets expect similar operations performed by the same application. The redirection of the packet to the destination application should be fast.

5.1.2 Abstract Data Type (ADT) operations

- Insertion,
- And deletion of a (key, value) pair.
- Search using a key.

All of the operations should be O(1) time operations.

5.1.3 Cuckoo Hashing

The search, insertion, and deletion operations should take $O(1)$ worst-case time for a perfect hashing technique. The perfect hashing technique requires that every key should be mapped to a unique location- there should be no collision. The set of possible keys is exponential in the size of the key (in bits). It is inefficient, impractical and sometimes impossible to reserve such a large space when the actual number of keys stored is relatively small. When collisions are possible, the ADT operations (5.1.2) can take $O(n)$ time in the worst case.

DPDK uses the Cuckoo hashing technique [17] as it provides worst-case $O(1)$ search. The lookups for search or deletion require at most two accesses to the table. Cuckoo hashing technique provides expected $O(1)$ insertion with $O(n)$ time in the worst case.

The key (with value) is stored in two tables T_1 and T_2 such that $|T_1| = |T_2| = r$ H_1 and H_2 are two hash functions $H_1, H_2 : K \rightarrow \{0, 1, 2, ..., r-1\}$. $H_1(k)$ and $H_2(k)$ are respectively the indices in the tables T_1 and T_2 where the key entries are stored. The key can be present in only one table at a time.

Search

```
def search(k: key)-> bool:
  if T1[h1(k)] == k:
    return True
  if T2[h2(k)] == k:
    return True
  return False
```

Figure 5.1: Search operation in Cuckoo Hash.

For a given key x, the indices in the hash tables i.e. $h_1(x)$ and $h_2(x)$ are calculated. If the key is present in either of the tables T_1 or T_2, the search operation is successful (Figure 5.1).

Delete

Delete operation is similar to a search. The entry in the table is deleted on a successful search. (Figure 5.2)

```
def delete(k: key)->bool:
  if T1[h1(k)] == k:
    T1[h1(k)] = None #deletion
    return True
  if T2[h2(k)] == k:
    T2[h2(k)] = None #deletion
    return True
  return False
```

Figure 5.2: Deletion in Cuckoo Hash

Insertion

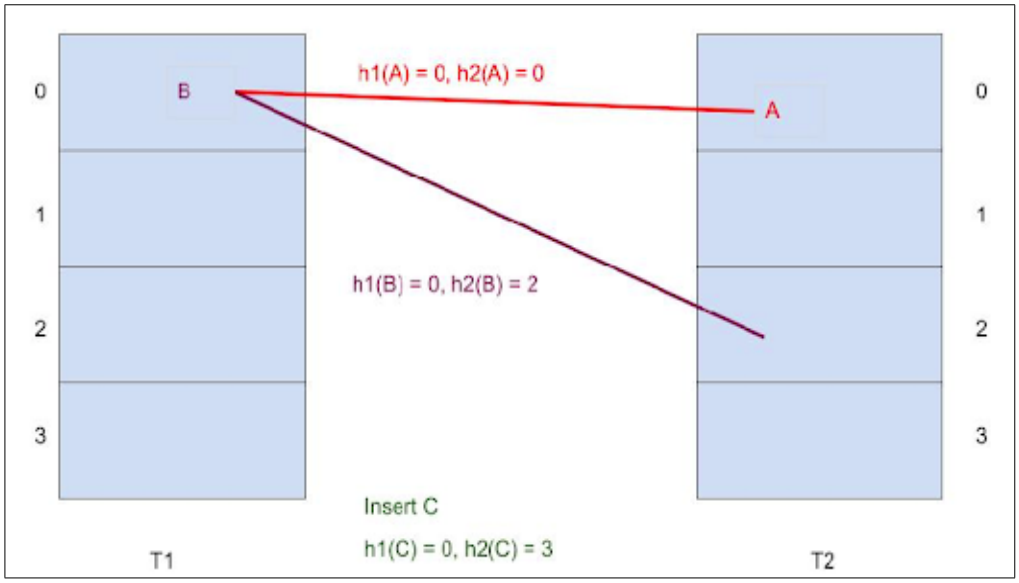
The insertion routine returns if the key already exists in any table (T_1 or T_2). Otherwise, the key is inserted in T_1. If the entry in T_1 was empty, insertion is successful and the procedure returns. Otherwise, the evicted entry is inserted in T_2. If the entry in T_2 is empty, the procedure returns, else the evicted entry is inserted in T_1. This process goes on until a cell is found empty or till the MAXLOOP iterations (Figure 5.3). The table entries are rehashed with a new pair of hash functions, and the insertion is tried again.

```
def insert(k:key)-> None:
  if search(k) == True:
    # key already exists
    return
  for i in range(MAXLOOP):
    k, T1[h1(k)] = T1[h1(k)], k # swap
    if k == None:
      # T1 cell was empty- insertion successful
      return
    k, T2[h2(k)] = T2[h2(k)], k
    if k == None:
      # T2 cell was empty- successful insertion
      return
  rehash() # insertion failed, use new hash functions
  insert(x) # try to insert again
  return
```

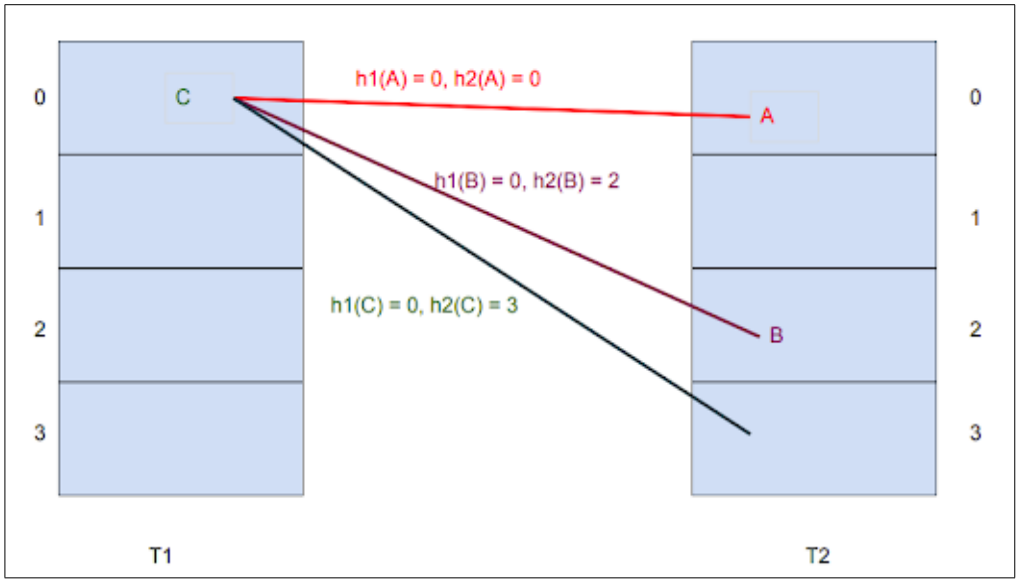
Figure 5.3: Insertion in Cuckoo Hash

Insertion Demonstration

The insertion sequence in the Figure 5.4 is A, B, C.



(a)



(b)

Figure 5.4: (a)Before insertion of the element C (b) After insertion

Failed insertion

The insertion sequence in the Figure 5.5 is A, B,C,D, and E. Figure 5.5 shows when an insertion (key E) fails because of repeated evictions. A rehashing with new hash functions is required in such a case.

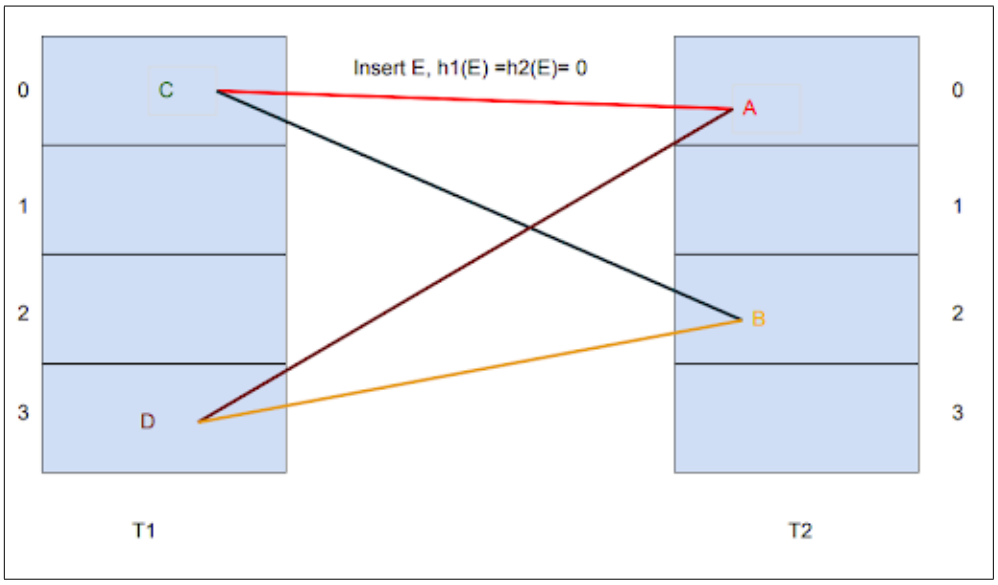


Figure 5.5: Failed Insertion in Cuckoo Hash

5.1.4 Implementation Details in DPDK

Hash Table

Cuckoo algorithm uses two tables with two different hash functions in the standard implementation. DPDK has a single table with two hash functions mapping to two disjoint sets of indices (buckets) within the same table.

Structure of an Entry

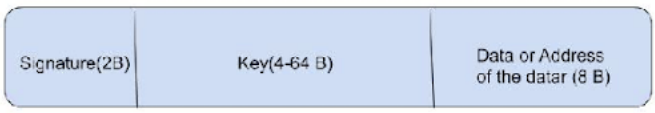


Figure 5.6: Structure of an Entry in Hash Table

The hash table entry has three fields (Figure 5.6):

- Signature- Optional 2- bytes field storing the partial hash signature of the key.
- Key- Variable sized (4-64 bytes).
- Value- Data field either contains an 8-byte integer data associated with the key. If the size of the data is greater than 8 bytes, it stores the memory address of the data.

Signature is used before matching the actual key. If the signatures do not match, the key does not match the table entry. This saves time as keys are longer than the signature.

Configuration Parameters

- Size of the Table
- Size of the key in bytes
- Flags - for providing multi-threading support and extensible bucket optimization in Cuckoo Hashing Technique (see 5.1.4).

Supported Features/Optimizations

- **Precomputed Hash** The user application can provide the precomputed hash along with the key. This enables the user to use the hash functions of his/her choosing. This also helps in performing lookups faster in the case of flow classification use case. All packets with the same 5-tuple (5.1.1) have the same hash.
- **Batching** The user can perform a search operation for batches of packets in a single call. This reduces memory access overheads by bringing contiguously allocated packets in cache together.
- **Multi-thread support** Thread level parallelism (TLP) allows multiple threads to work simultaneously on different cores. This increases the throughput significantly. DPDK provides TLP by
  - **Lock-Free concurrency** (Section 2.4.5)

    The deleted element should not be freed as some users might still be using them. The application can use RCU mechanisms to make sure that no reader is referring to the entry before deleting it.
  - **Read-Write Locks with Hardware Transactional Memory** Read-write locks are preferred when hardware transactional memory (HTM) support is available. The HTM support means that a group of instructions can be performed in an atomic unit with the hardware support .

HTM provides higher performance than other mutual exclusion mechanisms (refer [18]).

5.1.5 Extensible Bucket

The insertion procedure of the Cuckoo algorithm 5.1.3 is modified to eliminate the cost of rehashing in case of multiple evictions. The final evicted element is inserted in a linked list (based on [19]). The element is looked up in the primary bucket, secondary bucket, and finally in the linked list for search or delete operation. Look-up time is now not $O(1)$ in worst case. Figure 5.7 builds on the failed insertion case in 5.1.3. E is inserted in the linked list. P_1 and P_2 are present from previously failed insertions.

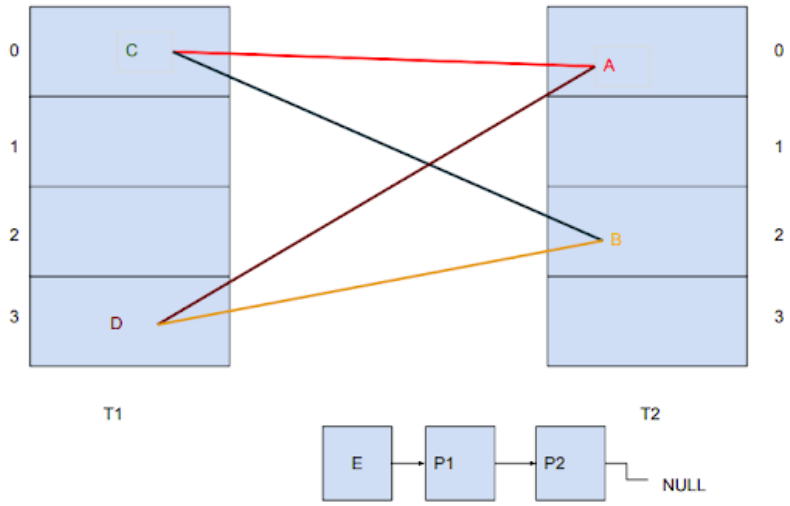


Figure 5.7: Cuckoo Hash with Extensible Bucket

5.2 Membership Library

DPDK provides the membership library for testing the membership of an element in the set. The idea is to use minimum space, minimum time to insert a new member, and search an existing member of the set. The membership library has two major differences from the hash library- the entire key is not stored (saving space) and faster lookup (saving time). There exists a *non-zero false positive probability* with its use - the probability that the membership test returns *True* even when the element is

not present. There are use cases which can handle false positive probability making the tradeoff with the hash library worthwhile. It is implemented in two ways.

- Bloom Filters
- Hash table Set Summaries (HTSS)

5.2.1 Use Case: Load Balancing

Each worker thread has an associated membership set. A newly arrived packet is checked to see whether it belongs to an existing flow or is the first packet of a new connection/flow. If the packet belongs to a new flow, the packet is directed to the current least loaded worker thread. This helps in uniform distribution of the workload among different worker threads. Note that this use case can deal with the false positive cases. The systems can handle the situation in which a newly packet with a small probability is forwarded to a relatively busy thread.

5.2.2 Abstract Data type (ADT) operations

- Insertion,
- And search for an element.

5.2.3 Bloom Filters

Bloom filters [20] are a space-efficient, probabilistic data structure with a non zero false positive probability. Bloom filters do not give false-negative results. If the membership query output is false, the member does not exist in the set. All ADT operations are O(1) time operation.

A Bloom filter is an array of m bits. Initially, all bits are 0. $H_1, H_2, ..., H_k$ are k hash functions such that $H_i : K \rightarrow 0, 1, 2, ..., m - 1$.

Insertion

$H_1, H_2...H_k$ are computed on a new element x. All bits with indices corresponding to the values $H_1(x), H_2(x), H_3(x)..H_k(x)$ are set to 1.

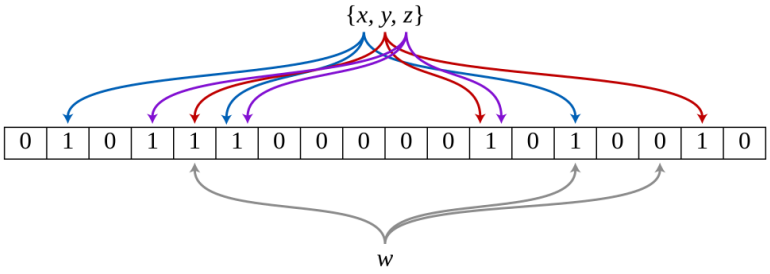


Figure 5.8: Bloom Filter Representation

Search

For a search key x, all hash functions are computed and are checked whether the corresponding bit is set to 1. If any bit is equal to 0, then the element is not present.

Deletion is not feasible

Deletion of a particular member is not possible in standard bloom filter. A bit in Bloom filter can correspond to multiple members of the set due to *collision* of hash values.

The bloom filter in Figure 5.8 has x, y, z as the members of the set. The elment z is not a member of the set as one of the queried bits is 0.

Analysis of False Positive Probability

It is possible that all bits corresponding to a particular query element are set by other members of the set even when the element is not present in the set. The probability increases with the number of hash functions k, the number of members present in the set and decrease with the size of Bloom filter, m.

A vector of bloom filters is maintained in the case of Load Balancing 5.2.1. Each bloom filter corresponds to one core. This vector can be looked up sequentially or simultaneously using thread level parallelism.

5.2.4 Hash Table Set Summaries (HTSS)

HTSS are based on hash table techniques like Cuckoo hashing. HTSS is preferable over bloom filters in cases where the number of queried bloom filters are large for a single query. HTSS differs from Hash tables as only the signature of the key (and

not the full key) is stored in HTSS. This enables space efficient and time efficient lookup operations.

Lookup operations have a non zero *false positive probability* as two keys might have the same signature. This probability of collision increases with the increase in the number of members in the set and decreases with the increase in size of the signature.

HTSS might also have a *false negative probability* - the lookup operation returns member as not present when actually it is there. This occurs when overwriting on an existing entry is allowed. This behavior is similar to a cache and can be used in Domain Name Server (DNS) systems.

5.3 Longest Prefix Match (LPM) Library

LPM is used for making forwarding decisions about an arrived packet on a router. The forwarding table stores address of subnetworks inside the core of the network and the final destinations on the edge. The address of subnetworks are stored in classless inter-domain routing (CIDR) format - 192.168.10.0/24 points to the destination router for addresses in the range 192.168.122.1 – 192.168.122.255. An arrived packet on a router may match more than one entry in forwarding table. The longest prefix match rule helps in determining the router closest to the destination.

LPM is generally implemented using a trie. Trie is a binary tree which stores one bit of the address on every node. The address is matched against every bit on the node. The leaf contains the next hop address.

5.3.1 Abstract Data Type (ADT) Operations

- Lookup of the next best hop.
- Insertion of a rule in the LPM data structure
- Deletion of a rule.

5.3.2 Implementation: IPv4

DPDK uses DIR 24-8 technique [21] for matching prefixes of IPv4 addresses. This technique makes a simple tradeoff - use more space to save time. Two tables are maintained (Figure 5.9):

- ***tbl24***: A table with 2^{24} entries . Each entry contains next hop or an index into tbl8, valid flag, external entry flag and depth or length of the rule.
- ***tbl8***: A table with 2^8 entries. All fields are same except external entry field is not present.

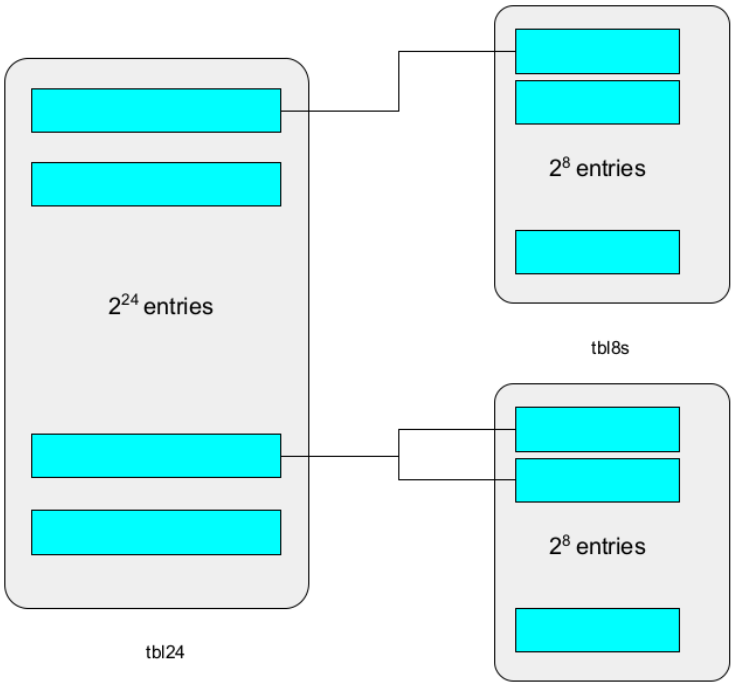


Figure 5.9: LPM: IPv4 Implementation

Flags

- **valid** If the *valid* flag is 0, then the entry is empty. Otherwise, it contains a rule.
- **external** If the *external* flag is 0, the lookup process ends at this entry.

Insertion

Insertion depends on the length l of CIDR prefix or the rule.

- $l = 24$ If the *tbl24* entry indexed by the the l bits of the rule is empty, set the valid flag to 1 and external flag to 0
- $l = 32$ Access the *tbl24*. If the valid or the external flag is 0, allocate a free *tbl8* and fill its entries according to the new rule and any previous rule present in the *tbl24* entry.
- $l < 24$ The prefix expansion is required. For $l = 21$, the prefix expansion leads to access of 8 entries in *tbl24*. All empty entries (valid flag is 0) are filled with the new rule.
- $l > 24$ The same process from the previous field is followed in the corresponding *tbl8* entries.

Search

A valid IPv4 address has 32 bits. First 24 bits are used to access a *tbl24* entry. If the entry is invalid, then the default route is taken for forwarding.

If the external flag is 0, the rule is accessed from *tbl24*. Else, the corresponding entry is accessed from *tbl8*.

Deletion

Deletion of the rule in the forwarding table should be followed by the deletion in LPM tables.

Configuration Parameters

All rules with length less than 24 bits can be accomodated in *tbl24*. The number of *tbl8* tables depend on the number of rules longer than 24 bits which did not have previously allocated *tbl8*. A maximum of 256 *tbl8* tables is permitted by default. No rules longer than 24 bits are accomodated when the quota of *tbl8* is exhausted. This works most of the time as longer rules are unlikely in the core routers.

Analysis

The main advantage of DIR 24-8 implemenation is atmost 2 memory accesses for a single lookup. The second memory access is required when matching rule is longer

than 24 bits.

The traditional **trie** requires $O(\log l)$ memory accesses, where l is the length of the rule.

5.3.3 Implementation : IPv6

IPv6 is implemented in **LPM6 library**. The implementation blends both the techniques of trie and DIR 24-8 to obtain space efficient faster lookup. The main data structure (Figure 5.10) contains:

- A primary table with 2^{24} entries as the root node of the trie.
- Each internal node or the leaf node, contains 2^8 entries.

The entries in the table are identical to IPv4 case 5.3.2. Table 5.1 shows the relation between number of memory accesses and the length of the rule in the forwarding table. The insertion, search and deletion operations are similar to IPv4. The max-

Length of the Rule	Number of Memory Accesses
$l \leq 24$	1
$24 < l \leq 32$	2
$32 < l \leq 40$	3
$40 < l \leq 48$	4
$48 < l \leq 56$	5
$56 < l \leq 64$	6
$64 < l \leq 72$	7
$72 < l \leq 80$	8
$80 < l \leq 88$	9
$88 < l \leq 96$	10
$96 < l \leq 104$	11
$104 < l \leq 112$	12
$112 < l \leq 120$	13
$120 < l \leq 128$	14

Table 5.1: Length of the Rule v/s. Number of memory accesses

imum depth is 14 for 128 bit Ipv6 address ($1 * 24 + 13 * 8 = 128$). The number of required *tbl8* tables are higher than the IPv4 case because of longer matching rules.

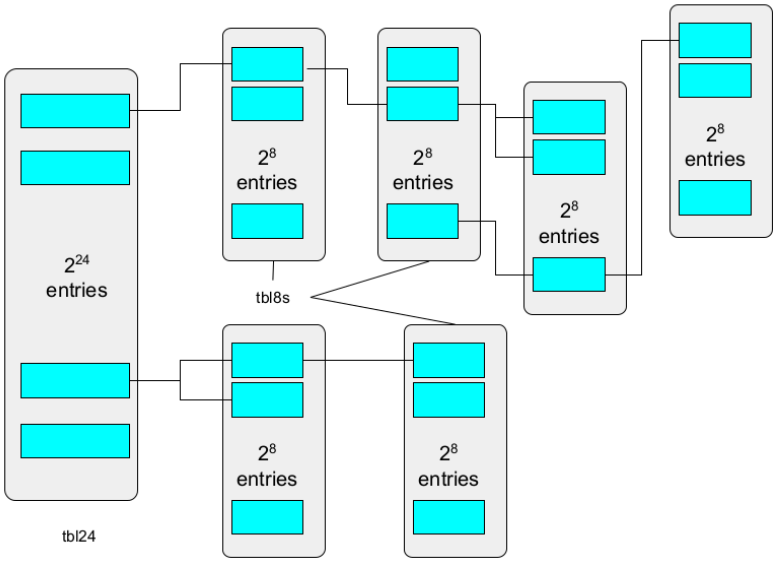


Figure 5.10: LPM: IPv6 Implementation

5.4 Summary

This chapter discussed the main libraries provided by the DPDK for performing network functions in the data plane. Note that users/vendors can build their own customized solutions for these applications (Chapter 6).

Chapter 6

Frameworks using DPDK

6.1 Vector Packet Processor

The Vector packet processor (VPP) [6] is a user-space framework for processing data packets. It is built over kernel-bypass software solutions - DPDK and netmap. VPP enables fast processing of packets in bulk mode."Processing" includes network functions e.g. - IPv4/IPv6 header processing, error detection/correction on data packets.

6.1.1 Role of DPDK

DPDKs main concern is low-level I.O - by-passing kernel stack and providing packets in userspace quickly. DPDK uses polling, receive-side scaling (RSS), and zero-copy techniques. DPDK enables I/O batching by polling for packets at regular intervals or after a threshold number of packets are ready for receipt/transmission. RSS is used for dividing multiple NIC Rx/Tx queues among different cores. It enables the classification and forwarding of packets to the respective core. Finally, DPDK directly copies the packets into userspace from NIC. An extra copy from kernel space to userspace is avoided. Zero-copying requires that the processing rate by the userspace driver is faster than the packet arrival rate. The majority of packet processing occurs in user applications such as VPP.

6.1.2 Vectorized Processing and related concepts

Vectorized processing is different from the batching performed by DPDK. The DPDK batches packets to reduce interrupt overheads. Vectorized processing uses batches to reduce the number of clock cycles during computation. Vectorized processing is the building block of VPP. Vectors are the fundamental primitive of VPP. Vectors are preallocated contiguous arrays, reused and are never freed. Fastclick [22] is a software router that provides batch processing - "Compute batching". However, it can happen only for certain functions designed for batch processing. It stores the batch in a linked list. The linked list occupies more space and is not contiguous.

6.1.3 VPP Principle

VPP implements the full network stack i.e. L2/L3 processing. VPP uses DPDK as an input (output) node during the receipt (transmission) of packets. VPP is designed to work with general-purpose CPUs. VPP follows a "run-to-completion" model - each packet is processed completely on a single core. VPP process packets in batches. The input batch is received from the DPDK ring. The batch is pushed to another node of the directed graph. These batches may be further subdivided and forwarded to other nodes until the packet is completely processed.

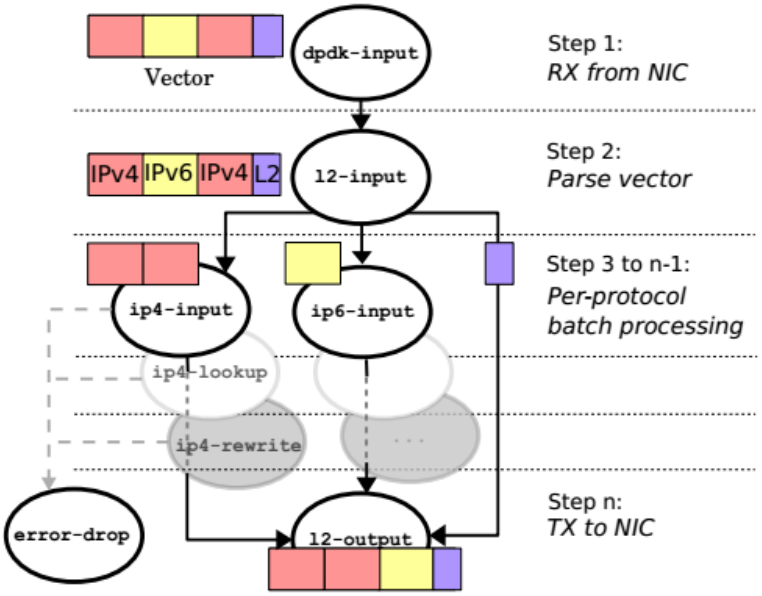


Figure 6.1: VPP Demonstration [6]

In the figure 6.1, the l2-input node parses the vector and segregates packets according to the l2 protocol i.e. IPV4 or IPV6. These packets are then forwarded to respective nodes. Error is detected at each node and the packet is dropped if needed.

6.1.4 Features of VPP

In the run-to-completion model, every single packet undergoes different functions sequentially. This causes a significant performance penalty because of:

- I-cache misses on the loading of new instructions for every new function call.
- Function call overhead - The creation of a new activation record and reshuffling of registers occurs for every new function call.
- D-cache misses: It becomes difficult to decide what portions of the packet to prefetch for computation.

I-cache misses

The same computation is performed over multiple packets simultaneously. The misses occur for only the first packet of the batch. The processing of the rest of the packets does not require fetching of instructions from memory. The miss-penalty is hence amortized over a batch of packets.

Function calls

A network function is called only once per entire batch, the total number of function calls for a given number of packets is reduced. Inline functions (Section 4.4) are used as code optimization.

Data cache misses

The same operation is performed on multiple packets sequentially on a given core. The headers required for processing of further packets are known after the first packet. So data for the $i + 1^{th}$ packet can be prefetched during the processing of the i^{th} packet.

6.1.5 Code optimizations

VPP uses multi-loop, data prefetching, branch prediction, function flattening and buffer allocation strategy for direct cache access.

Multi-loop

Loops are written for parallel processing of the same functions on N (>1) packets simultaneously on different cores.

Data Prefetching

Discussed in section 6.1.4 on the discussion of data cache-misses. It can be combined with a multi-loop which keeps the CPU pipeline full. Prefetching is not possible for the first N packets and there is no packet available for prefetching during the last N packets. But this penalty is amortized over many packets.

Figure 6.2 demonstrates multiloop and prefetching with $N = 2$. P_3, P_4 are prefetched while P_1 and P_2 are processed.

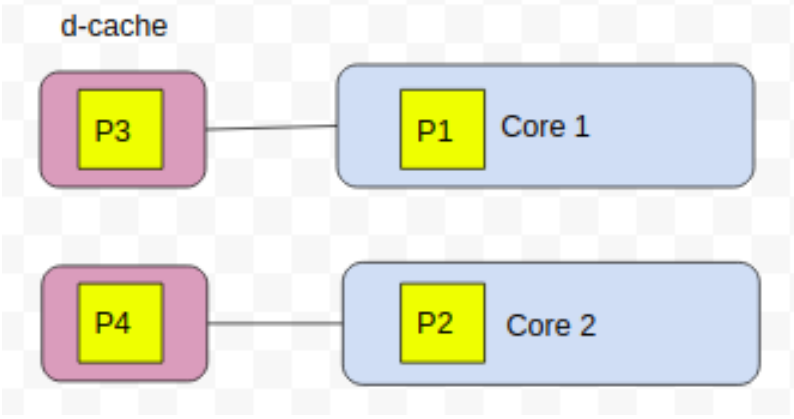


Figure 6.2: Multiloop and prefetching with N = 2.

Branch Prediction

Branch prediction works on the principle that the majority of packets follow the same path on a conditional statement. The developers can give a hint to the compiler about the branch that will be taken most of the time. The keyword likely (unlikely) is used to give hints in the C language (Figure 6.3). The modern-day CPUs have branch predictor implemented in hardware- so many hints are unnecessary.

```
if (likely(success)){
  ...
}
if (unlikely(error)){
...
}
```

Figure 6.3: Branch Prediction

Function flattening

Most of the graph nodes in VPP are inline functions. The inline functions are placed in the calling function code and no new function calls are made. It also enables the compiler to make further optimizations.

Direct Cache Access

The modern-day computer systems prefill the packets in the L3 cache (and not main memory) on the Rx (DMA) side. This reduces the number of cycles spent in bringing the packets from memory to CPU. VPP uses a buffer-allocation strategy i.e. the number of packets, size of each buffer, etc to avoid sending packets back to main memory during the entire process.

6.2 Transport Layer Development Kit (TLDK)

TLDK [23] is a set of C libraries that provides L4 functions - TCP and UDP processing.

Input Parameters

- Supported hardware offloads.
- L3/L2 addresses (IP/MAC).
- MTU of the links.

This helps in filling up all the relevant headers and mbuf metadata (single v/s. multisegmented (4.2)).

6.2.1 Features

- TLDK implements the main DPDK concepts such as bulk-packet processing, non-blocking API, no context or mode switch, cache and memory alignment.
- TLDK is built over DPDK and is compatible with vectorized packet processing (6.1). TLDK can not be used with BSD sockets (Linux Socket API).
- **Pull vs. Push** Network stacks are generally push-based systems - packets are pushed to the application even when they do not need it. TLDK builds a pull-based system in which applicatiion requests for packets when it needs them.

Figure 6.4 shows the movement of packets in the DPDK-VPP-TLDK-Application stack. Note that all components are loaded in the same address space. So multiple-memory copies are not required. VPP layer is optional. Data packets move to the application layer. Control packets (like ICMP, ARP requests etc.) are handled below the application layer.

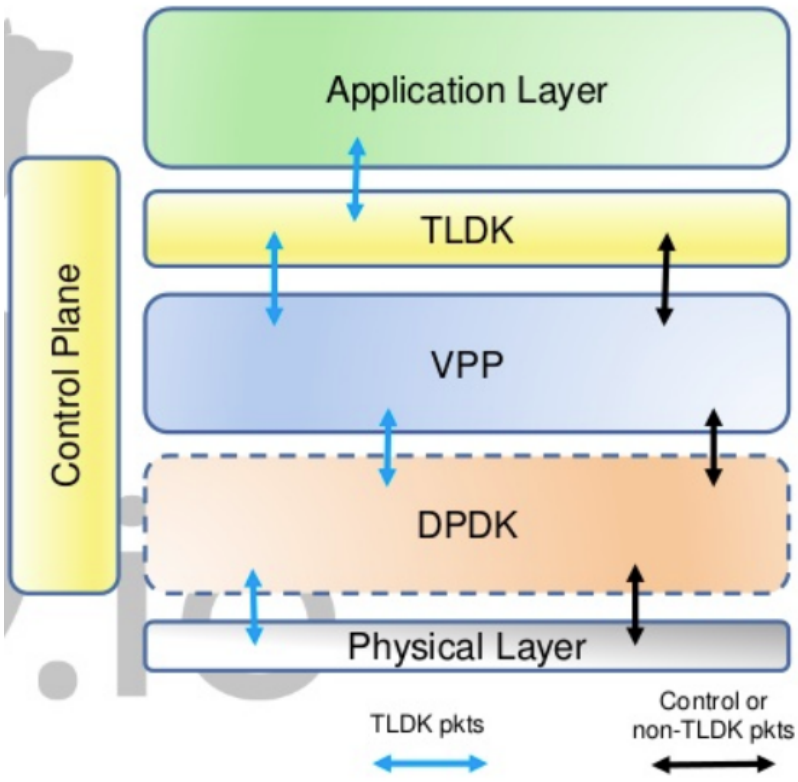


Figure 6.4: Packet movement in DPDK [7]

Chapter 7

Related Work and Comparisons

This report is concluded by describing three software techniques for fast packet processing besides DPDK.

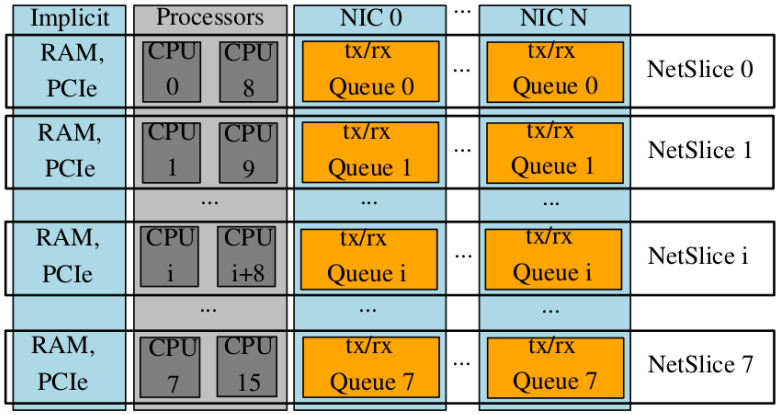


Figure 7.1: Netslices [8]

7.1 Netslices

The packet capture libraries (3.3.3) provide raw packets into the user space using raw sockets. However the path taken by packet can traverse across different cores. This is a performance bottleneck due to slow accesses to non local memory in NUMA RAMs and cache misses in the local per core cache. This implies that raw sockets do not scale with the number of cores.

The key idea in Netslices [8] is the spatial partitioning of physical resources like CPU across different NIC queues. Figure 7.1 shows how the resources are

partitioned. A minimum of 2 cores are provided to a pair of Tx/Rx per NIC ring to match the line speeds. If the memory is NUMA-aware, local memory of the cores is implicitly reserved for handling packet.

The problem of multiple memory copies - from NIC to kernel and from kernel to user space still remains. The solution scales with the number of cores on the system.

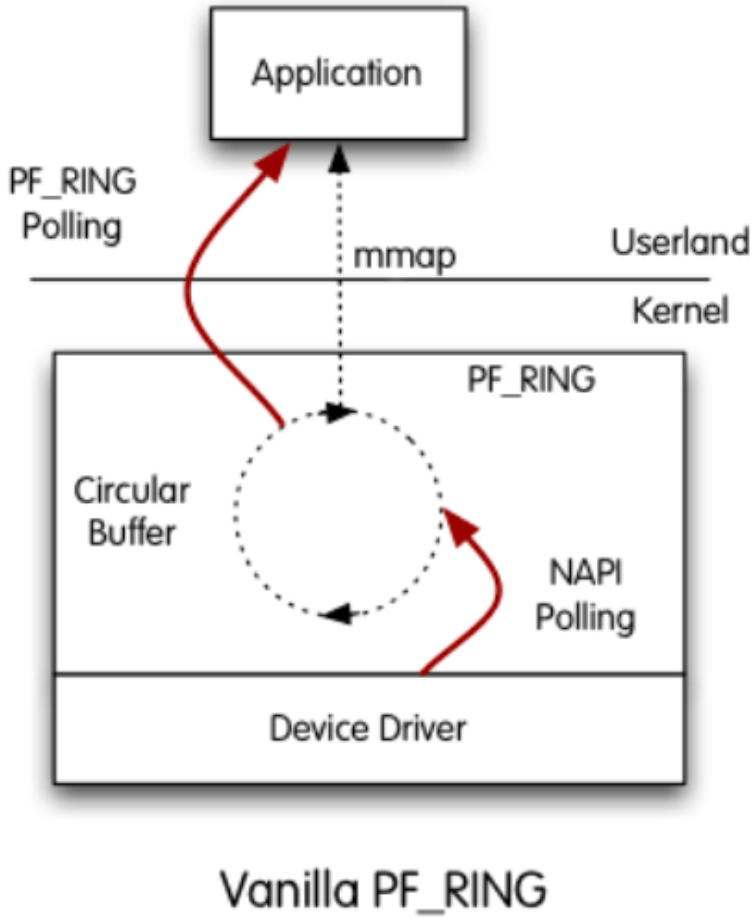


Figure 7.2: Vanilla PF_RING [9]

7.2 PF_RING and PF_RING ZC

PF_RING (figure 7.2) is a new socket which captures packets with the help of New API for polling in Linux. The packets are polled and copied in the kernel PF_RING buffer. The user space application then retrieves these packets by another polling mechanism between kernel and user space. PF_RING socket preallocates memory buffers for handling packets. This removes the allocation/deallocation overhead as-

sociated with ***sk_buff*** in Linux stack (refer 1.2.2). However the problem of multiple meory copies still remains.

PF_RING ZC (Zero Copy) is an enhanced library which gives the userpace process direct access to NIC inteface. The packets are copied directly into memory buffers provided by the user space applications. This technique can also scale linearly with the number of cores as the tuple (queue, application) can be independently handled (refer [24]).

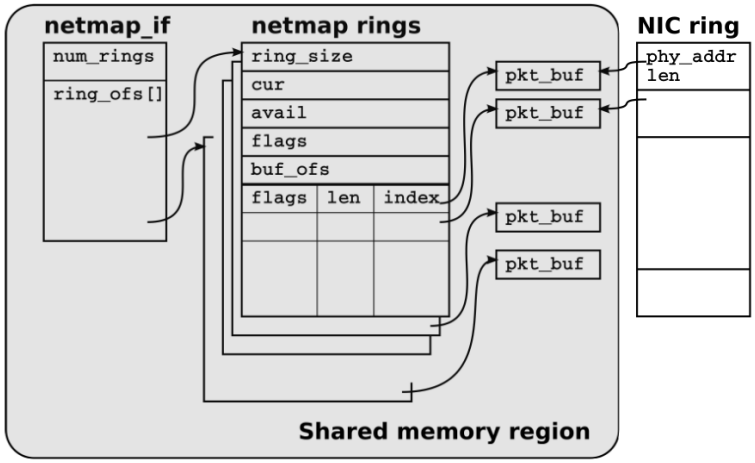


Figure 7.3: Netmap Rings [10]

7.3 netmap

Netmap [10] and DPDK try to solve the similar problems with Linux stack i.e. multiple memory copies, interrupt processing overheads and overhead associated with allocation and deallocation of buffers (section 1.2.2). However, netmap solves this problem with the help of Linux kernel system calls - *ioctl*, *mmap*, *epoll* and *select* among others. But the packet processing occurs in user space. DPDK, as seen earlier, does not take any kernel support for packet processing except at the initialization.

7.3.1 Working Principles of *netmap*

Netmap uses preallocated buffers. These buffers are in a shared memory region between kernel and user space. NIC rings (that stores the pointers to the data

packets) are replicated in the form of netmap rings (Figure 7.3). These rings are used to communicate between the user space application and kernel. The actual NIC rings are protected from misbehaving applications by the kernel. This protection is the advantage of netmap over DPDK. In the case of DPDK PMDs, the NIC rings are directly mapped in the user space. However different processes using the same netmap ring are not protected from each other. The solution to this problem is to use different netmap rings for different processes on different cores. This also enables parallelism in netmap.

The netmap rings store relative offsets of packets from the base addresses enabling easy translation from kernel space to user space and vice versa. Hence netmap uses zero-copy technique.

The packets of batches can be processed together by using polling system calls. This reduces interrupt processing overheads.

7.4 Closure

	Zero-Copy	Batching	Parallelism
Netslices	No	Yes	Yes
PF_RING	Yes (ZC)	Yes	Yes
netmap	Yes	Yes	Yes (multiple netmap rings)
DPDK	Yes	Yes	Yes

Table 7.1: Comparison of Software Techniques

Packet processing in all the techniques discussed here operate in user-space. All the techniques discussed above can handle speeds upto 10 Gigabits/sec wire speeds (refer [11]).

DPDK is preferable over all the techniques and is widely used in industry for handling packet processing needs. DPDK was started by Intel, which is the leader in manufacturing of network hardware and processors. Now it is a part of the Linux foundation with support from big players in telecom industry like AT & T, ARM, and Ericcson among others [25]. DPDK's success is attributed to the application of latest technological advances (discussed in 2.4). DPDK provides the support on application level by using state-of-the-art algorithms optimized for multi-core

architectures (Chapter 5).

Bibliography

[1] https://people.cs.clemson.edu/~westall/853/notes/skbuff.pdf.

[2] https://www.slideshare.net/garyachy/dpdk-44585840.

[3] P. Emmerich, M. Pudelko, S. Bauer, S. Huber, T. Zwickl, and G. Carle, "User space network drivers," *arXiv preprint arXiv:1901.10664*, 2019.

[4] https://doc.dpdk.org/guides/prog_guide/.

[5] https://en.wikipedia.org/wiki/Compare-and-swap.

[6] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi, "High-speed software data plane via vectorized packet processing," *IEEE Communications Magazine*, vol. 56, no. 12, pp. 97–103, 2018.

[7] https://www.slideshare.net/blopeur/tldk-fdio-sept-2016.

[8] T. Marian, K. S. Lee, and H. Weatherspoon, "Netslices: scalable multi-core packet processing in user-space," in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, pp. 27–38, 2012.

[9] https://www.ntop.org/products/packet-capture/pf_ring/.

[10] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in *21st USENIX Security Symposium (USENIX Security 12)*, pp. 101–112, 2012.

[11] D. Cerović, V. Del Piccolo, A. Amamou, K. Haddadou, and G. Pujolle, "Fast packet processing: A survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3645–3676, 2018.

[12] L. L. Peterson and B. S. Davie, *Computer networks: a systems approach*. Elsevier, 2007.

[13] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 15–28, 2009.

[14] https://www.dpdk.org/blog/2019/08/21/memory-in-dpdk-part-1-general-concepts/

[15] https://www.dpdk.org/blog/2019/09/14/memory-in-dpdk-part-2-deep-dive-into-io/.

[16] `https://doc.dpdk.org/guides/nics/af_packet.html`.

[17] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *European Symposium on Algorithms*, pp. 121–133, Springer, 2001.

[18] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th annual international symposium on computer architecture*, pp. 289–300, 1993.

[19] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1543–1561, 2010.

[20] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[21] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proceedings. IEEE INFOCOM'98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No. 98*, vol. 3, pp. 1240–1247, IEEE, 1998.

[22] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 5–16, IEEE, 2015.

[23] `https://github.com/FDio/tldk`.

[24] L. Rizzo, L. Deri, and A. Cardigliano, "10 gbit/s line rate packet processing using commodity hardware: Survey and new proposals."

[25] `https://www.dpdk.org/ecosystem/#members`.