DPDK

DATA PLANE DEVELOPMENT KIT

Programmer's Guide

Release 16.11.0

November 13, 2016

CHAPTER

ONE

INTRODUCTION

This document provides software architecture information, development environment information and optimization guidelines.

For programming examples and for instructions on compiling and running each sample application, see the *DPDK Sample Applications User Guide* for details.

For general information on compiling and running applications, see the *DPDK Getting Started Guide*.

1.1 Documentation Roadmap

The following is a list of DPDK documents in the suggested reading order:

- **Release Notes** (this document): Provides release-specific information, including supported features, limitations, fixed issues, known issues and so on. Also, provides the answers to frequently asked questions in FAQ format.
- **Getting Started Guide** : Describes how to install and configure the DPDK software; designed to get users up and running quickly with the software.
- **FreeBSD* Getting Started Guide** : A document describing the use of the DPDK with FreeBSD* has been added in DPDK Release 1.6.0. Refer to this guide for installation and configuration instructions to get started using the DPDK with FreeBSD*.
- **Programmer's Guide** (this document): Describes:
  - The software architecture and how to use it (through examples), specifically in a Linux* application (linuxapp) environment
  - The content of the DPDK, the build system (including the commands that can be used in the root DPDK Makefile to build the development kit and an application) and guidelines for porting an application
  - Optimizations used in the software and those that should be considered for new development

  A glossary of terms is also provided.
- **API Reference** : Provides detailed information about DPDK functions, data structures and other programming constructs.
- **Sample Applications User Guide**: Describes a set of sample applications. Each chapter describes a sample application that showcases specific functionality and provides instructions on how to compile, run and use the sample application.

1.2 Related Publications

The following documents provide information that is relevant to the development of applications using the DPDK:

- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide

Part 1: Architecture Overview

CHAPTER

TWO

OVERVIEW

This section gives a global overview of the architecture of Data Plane Development Kit (DPDK).

The main goal of the DPDK is to provide a simple, complete framework for fast packet processing in data plane applications. Users may use the code to understand some of the techniques employed, to build upon for prototyping or to add their own protocol stacks. Alternative ecosystem options that use the DPDK are available.

The framework creates a set of libraries for specific environments through the creation of an Environment Abstraction Layer (EAL), which may be specific to a mode of the Intel® architecture (32-bit or 64-bit), Linux* user space compilers or a specific platform. These environments are created through the use of make files and configuration files. Once the EAL library is created, the user may link with the library to create their own applications. Other libraries, outside of EAL, including the Hash, Longest Prefix Match (LPM) and rings libraries are also provided. Sample applications are provided to help show the user how to use various features of the DPDK.

The DPDK implements a run to completion model for packet processing, where all resources must be allocated prior to calling Data Plane applications, running as execution units on logical processing cores. The model does not support a scheduler and all devices are accessed by polling. The primary reason for not using interrupts is the performance overhead imposed by interrupt processing.

In addition to the run-to-completion model, a pipeline model may also be used by passing packets or messages between cores via the rings. This allows work to be performed in stages and may allow more efficient use of code on cores.

2.1 Development Environment

The DPDK project installation requires Linux and the associated toolchain, such as one or more compilers, assembler, make utility, editor and various libraries to create the DPDK components and libraries.

Once these libraries are created for the specific environment and architecture, they may then be used to create the user's data plane application.

When creating applications for the Linux user space, the glibc library is used. For DPDK applications, two environmental variables (RTE_SDK and RTE_TARGET) must be configured before compiling the applications. The following are examples of how the variables can be set:

```
export RTE_SDK=/home/user/DPDK
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started Guide* for information on setting up the development environment.

2.2 Environment Abstraction Layer

The Environment Abstraction Layer (EAL) provides a generic interface that hides the environment specifics from the applications and libraries. The services provided by the EAL are:

- DPDK loading and launching
- Support for multi-process and multi-thread execution types
- Core affinity/assignment procedures
- System memory allocation/de-allocation
- Atomic/lock operations
- Time reference
- PCI bus access
- Trace and debug functions
- CPU feature identification
- Interrupt handling
- Alarm operations
- Memory management (malloc)

The EAL is fully described in *Environment Abstraction Layer*.

2.3 Core Components

The *core components* are a set of libraries that provide all the elements needed for high-performance packet processing applications.

2.3.1 Ring Manager (librte_ring)

The ring structure provides a lockless multi-producer, multi-consumer FIFO API in a finite size table. It has some advantages over lockless queues; easier to implement, adapted to bulk operations and faster. A ring is used by the *Memory Pool Manager (librte_mempool)* and may be used as a general communication mechanism between cores and/or execution blocks connected together on a logical core.

This ring buffer and its usage are fully described in *Ring Library*.

2.3.2 Memory Pool Manager (librte_mempool)

The Memory Pool Manager is responsible for allocating pools of objects in memory. A pool is identified by name and uses a ring to store free objects. It provides some other optional

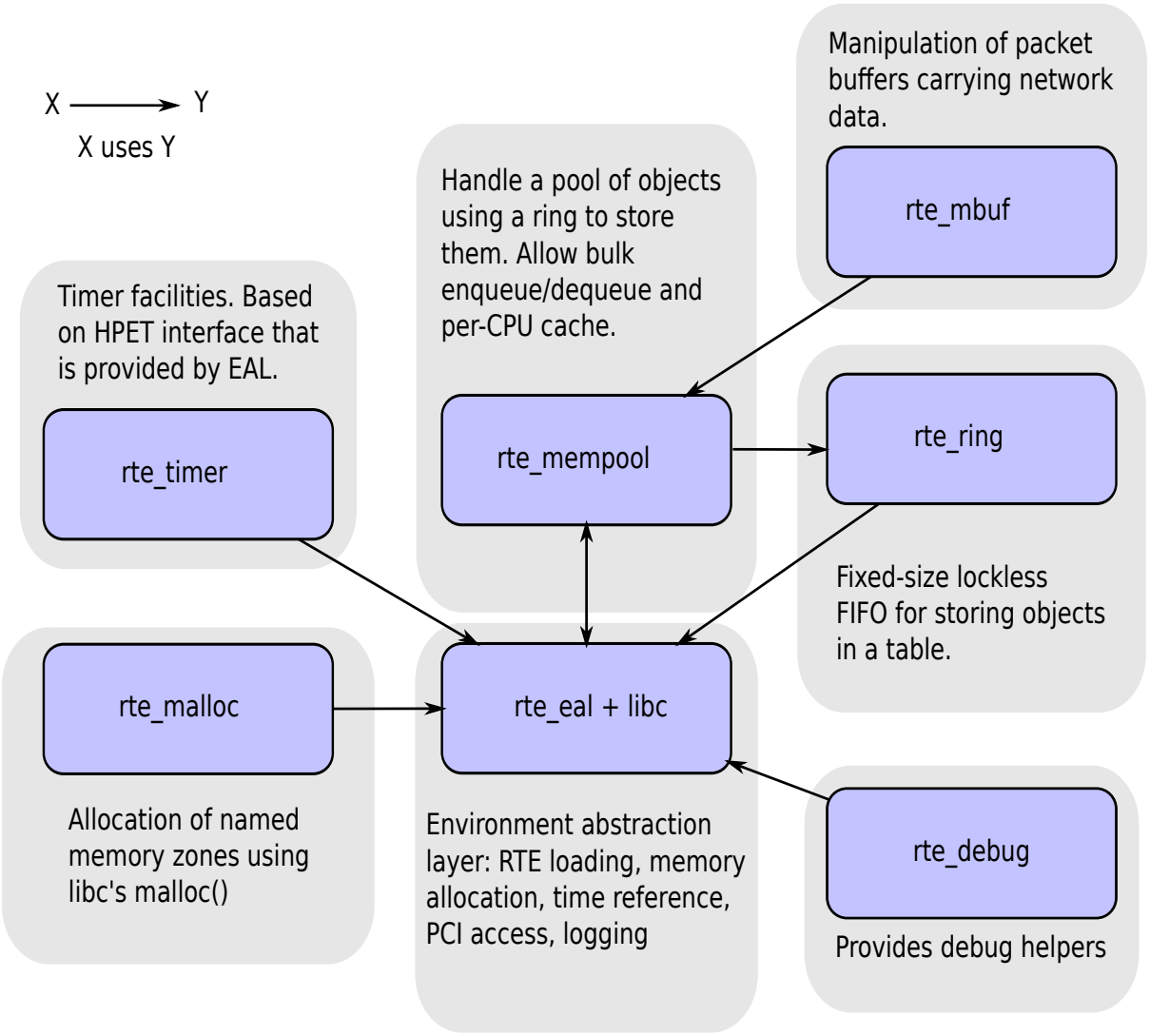


Fig. 2.1: Core Components Architecture

services, such as a per-core object cache and an alignment helper to ensure that objects are padded to spread them equally on all RAM channels.

This memory pool allocator is described in *Mempool Library*.

2.3.3 Network Packet Buffer Management (librte_mbuf)

The mbuf library provides the facility to create and destroy buffers that may be used by the DPDK application to store message buffers. The message buffers are created at startup time and stored in a mempool, using the DPDK mempool library.

This library provides an API to allocate/free mbufs, manipulate control message buffers (ctrlmbuf) which are generic message buffers, and packet buffers (pktmbuf) which are used to carry network packets.

Network Packet Buffer Management is described in *Mbuf Library*.

2.3.4 Timer Manager (librte_timer)

This library provides a timer service to DPDK execution units, providing the ability to execute a function asynchronously. It can be periodic function calls, or just a one-shot call. It uses the timer interface provided by the Environment Abstraction Layer (EAL) to get a precise time reference and can be initiated on a per-core basis as required.

The library documentation is available in *Timer Library*.

2.4 Ethernet* Poll Mode Driver Architecture

The DPDK includes Poll Mode Drivers (PMDs) for 1 GbE, 10 GbE and 40GbE, and para virtualized virtio Ethernet controllers which are designed to work without asynchronous, interrupt-based signaling mechanisms.

See *Poll Mode Driver*.

2.5 Packet Forwarding Algorithm Support

The DPDK includes Hash (librte_hash) and Longest Prefix Match (LPM,librte_lpm) libraries to support the corresponding packet forwarding algorithms.

See *Hash Library* and *LPM Library* for more information.

2.6 librte_net

The librte_net library is a collection of IP protocol definitions and convenience macros. It is based on code from the FreeBSD* IP stack and contains protocol numbers (for use in IP headers), IP-related macros, IPv4/IPv6 header structures and TCP, UDP and SCTP header structures.

CHAPTER

THREE

ENVIRONMENT ABSTRACTION LAYER

The Environment Abstraction Layer (EAL) is responsible for gaining access to low-level resources such as hardware and memory space. It provides a generic interface that hides the environment specifics from the applications and libraries. It is the responsibility of the initialization routine to decide how to allocate these resources (that is, memory space, PCI devices, timers, consoles, and so on).

Typical services expected from the EAL are:

- DPDK Loading and Launching: The DPDK and its application are linked as a single application and must be loaded by some means.
- Core Affinity/Assignment Procedures: The EAL provides mechanisms for assigning execution units to specific cores as well as creating execution instances.
- System Memory Reservation: The EAL facilitates the reservation of different memory zones, for example, physical memory areas for device interactions.
- PCI Address Abstraction: The EAL provides an interface to access PCI address space.
- Trace and Debug Functions: Logs, dump_stack, panic and so on.
- Utility Functions: Spinlocks and atomic counters that are not provided in libc.
- CPU Feature Identification: Determine at runtime if a particular feature, for example, Intel® AVX is supported. Determine if the current CPU supports the feature set that the binary was compiled for.
- Interrupt Handling: Interfaces to register/unregister callbacks to specific interrupt sources.
- Alarm Functions: Interfaces to set/remove callbacks to be run at a specific time.

3.1 EAL in a Linux-userland Execution Environment

In a Linux user space environment, the DPDK application runs as a user-space application using the pthread library. PCI information about devices and address space is discovered through the /sys kernel interface and through kernel modules such as uio_pci_generic, or igb_uio. Refer to the UIO: User-space drivers documentation in the Linux kernel. This memory is mmap'd in the application.

The EAL performs physical memory allocation using mmap() in hugetlbfs (using huge page sizes to increase performance). This memory is exposed to DPDK service layers such as the *Mempool Library*.

At this point, the DPDK services layer will be initialized, then through pthread setaffinity calls, each execution unit will be assigned to a specific logical core to run as a user-level thread.

The time reference is provided by the CPU Time-Stamp Counter (TSC) or by the HPET kernel API through a mmap() call.

3.1.1 Initialization and Core Launching

Part of the initialization is done by the start function of glibc. A check is also performed at initialization time to ensure that the micro architecture type chosen in the config file is supported by the CPU. Then, the main() function is called. The core initialization and launch is done in rte_eal_init() (see the API documentation). It consist of calls to the pthread library (more specifically, pthread_self(), pthread_create(), and pthread_setaffinity_np()).

Note: Initialization of objects, such as memory zones, rings, memory pools, lpm tables and hash tables, should be done as part of the overall application initialization on the master lcore. The creation and initialization functions for these objects are not multi-thread safe. However, once initialized, the objects themselves can safely be used in multiple threads simultaneously.

3.1.2 Multi-process Support

The Linuxapp EAL allows a multi-process as well as a multi-threaded (pthread) deployment model. See chapter *Multi-process Support* for more details.

3.1.3 Memory Mapping Discovery and Memory Reservation

The allocation of large contiguous physical memory is done using the hugetlbfs kernel filesystem. The EAL provides an API to reserve named memory zones in this contiguous memory. The physical address of the reserved memory for that memory zone is also returned to the user by the memory zone reservation API.

Note: Memory reservations done using the APIs provided by rte_malloc are also backed by pages from the hugetlbfs filesystem.

3.1.4 Xen Dom0 support without hugetbls

The existing memory management implementation is based on the Linux kernel hugepage mechanism. However, Xen Dom0 does not support hugepages, so a new Linux kernel module rte_dom0_mm is added to workaround this limitation.

The EAL uses IOCTL interface to notify the Linux kernel module rte_dom0_mm to allocate memory of specified size, and get all memory segments information from the module, and the EAL uses MMAP interface to map the allocated memory. For each memory segment, the physical addresses are contiguous within it but actual hardware addresses are contiguous within 2MB.

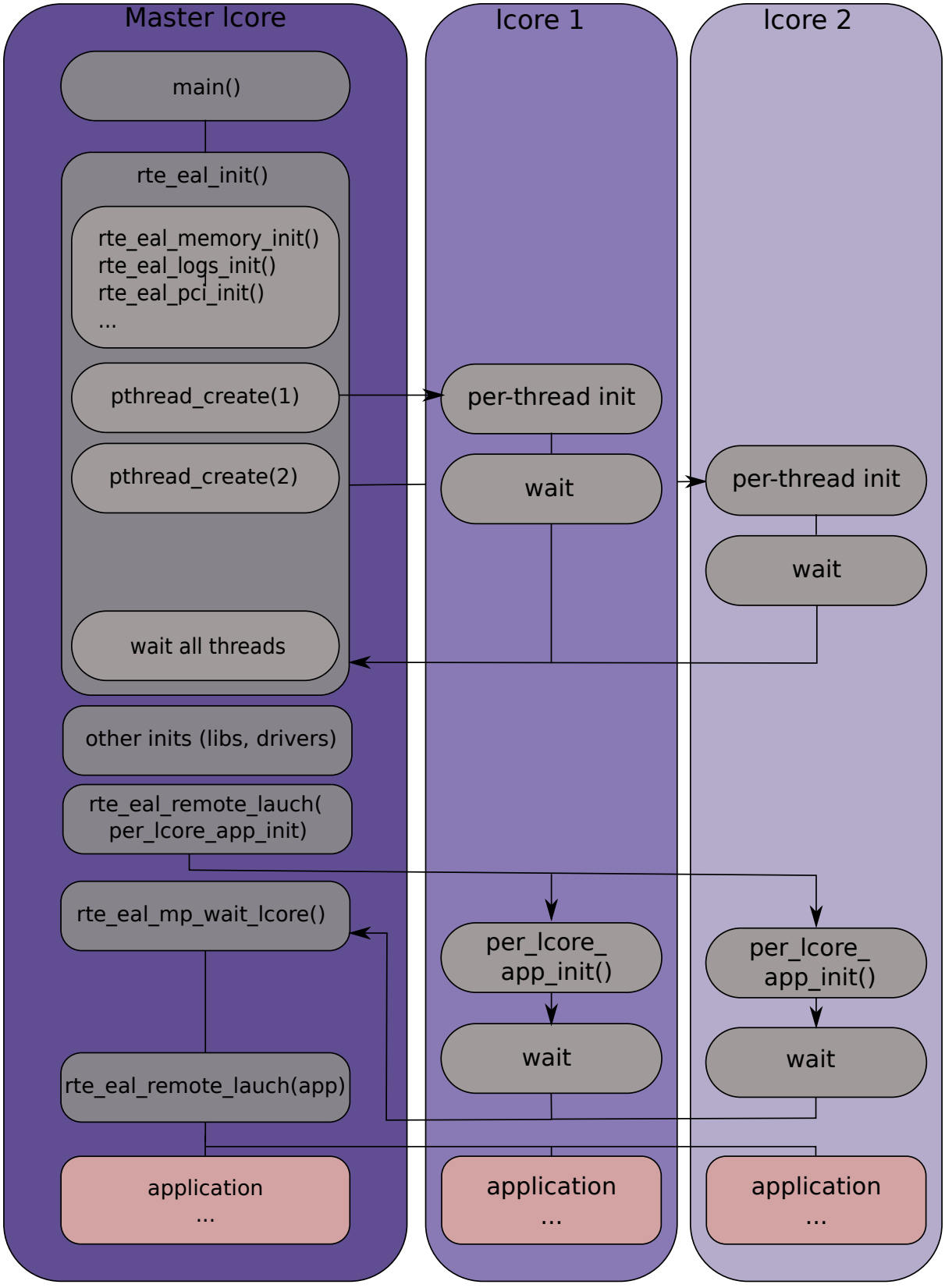


Fig. 3.1: EAL Initialization in a Linux Application Environment

3.1.5 PCI Access

The EAL uses the /sys/bus/pci utilities provided by the kernel to scan the content on the PCI bus. To access PCI memory, a kernel module called uio_pci_generic provides a /dev/uioX device file and resource files in /sys that can be mmap'd to obtain access to PCI address space from the application. The DPDK-specific igb_uio module can also be used for this. Both drivers use the uio kernel feature (userland driver).

3.1.6 Per-lcore and Shared Variables

Note: lcore refers to a logical execution unit of the processor, sometimes called a hardware *thread*.

Shared variables are the default behavior. Per-lcore variables are implemented using *Thread Local Storage* (TLS) to provide per-thread local storage.

3.1.7 Logs

A logging API is provided by EAL. By default, in a Linux application, logs are sent to syslog and also to the console. However, the log function can be overridden by the user to use a different logging mechanism.

Trace and Debug Functions

There are some debug functions to dump the stack in glibc. The rte_panic() function can voluntarily provoke a SIG_ABORT, which can trigger the generation of a core file, readable by gdb.

3.1.8 CPU Feature Identification

The EAL can query the CPU at runtime (using the rte_cpu_get_feature() function) to determine which CPU features are available.

3.1.9 User Space Interrupt Event

- User Space Interrupt and Alarm Handling in Host Thread

The EAL creates a host thread to poll the UIO device file descriptors to detect the interrupts. Callbacks can be registered or unregistered by the EAL functions for a specific interrupt event and are called in the host thread asynchronously. The EAL also allows timed callbacks to be used in the same way as for NIC interrupts.

Note: In DPDK PMD, the only interrupts handled by the dedicated host thread are those for link status change, i.e. link up and link down notification.

- RX Interrupt Event

The receive and transmit routines provided by each PMD don't limit themselves to execute in polling thread mode. To ease the idle polling with tiny throughput, it's useful to pause the polling and wait until the wake-up event happens. The RX interrupt is the first choice to be such kind of wake-up event, but probably won't be the only one.

EAL provides the event APIs for this event-driven thread mode. Taking linuxapp as an example, the implementation relies on epoll. Each thread can monitor an epoll instance in which all the wake-up events' file descriptors are added. The event file descriptors are created and mapped to the interrupt vectors according to the UIO/VFIO spec. From bsdapp's perspective, kqueue is the alternative way, but not implemented yet.

EAL initializes the mapping between event file descriptors and interrupt vectors, while each device initializes the mapping between interrupt vectors and queues. In this way, EAL actually is unaware of the interrupt cause on the specific vector. The eth_dev driver takes responsibility to program the latter mapping.

Note: Per queue RX interrupt event is only allowed in VFIO which supports multiple MSI-X vector. In UIO, the RX interrupt together with other interrupt causes shares the same vector. In this case, when RX interrupt and LSC(link status change) interrupt are both enabled(intr_conf.lsc == 1 && intr_conf.rxq == 1), only the former is capable.

The RX interrupt are controlled/enabled/disabled by ethdev APIs - 'rte_eth_dev_rx_intr_*'. They return failure if the PMD hasn't support them yet. The intr_conf.rxq flag is used to turn on the capability of RX interrupt per device.

3.1.10 Blacklisting

The EAL PCI device blacklist functionality can be used to mark certain NIC ports as blacklisted, so they are ignored by the DPDK. The ports to be blacklisted are identified using the PCIe* description (Domain:Bus:Device.Function).

3.1.11 Misc Functions

Locks and atomic operations are per-architecture (i686 and x86_64).

3.2 Memory Segments and Memory Zones (memzone)

The mapping of physical memory is provided by this feature in the EAL. As physical memory can have gaps, the memory is described in a table of descriptors, and each descriptor (called rte_memseg) describes a contiguous portion of memory.

On top of this, the memzone allocator's role is to reserve contiguous portions of physical memory. These zones are identified by a unique name when the memory is reserved.

The rte_memzone descriptors are also located in the configuration structure. This structure is accessed using rte_eal_get_configuration(). The lookup (by name) of a memory zone returns a descriptor containing the physical address of the memory zone.

Memory zones can be reserved with specific start address alignment by supplying the align parameter (by default, they are aligned to cache line size). The alignment value should be a

power of two and not less than the cache line size (64 bytes). Memory zones can also be reserved from either 2 MB or 1 GB hugepages, provided that both are available on the system.

3.3 Multiple pthread

DPDK usually pins one pthread per core to avoid the overhead of task switching. This allows for significant performance gains, but lacks flexibility and is not always efficient.

Power management helps to improve the CPU efficiency by limiting the CPU runtime frequency. However, alternately it is possible to utilize the idle cycles available to take advantage of the full capability of the CPU.

By taking advantage of cgroup, the CPU utilization quota can be simply assigned. This gives another way to improve the CPU efficiency, however, there is a prerequisite; DPDK must handle the context switching between multiple pthreads per core.

For further flexibility, it is useful to set pthread affinity not only to a CPU but to a CPU set.

3.3.1 EAL pthread and lcore Affinity

The term "lcore" refers to an EAL thread, which is really a Linux/FreeBSD pthread. "EAL pthreads" are created and managed by EAL and execute the tasks issued by *remote_launch*. In each EAL pthread, there is a TLS (Thread Local Storage) called *_lcore_id* for unique identification. As EAL pthreads usually bind 1:1 to the physical CPU, the *_lcore_id* is typically equal to the CPU ID.

When using multiple pthreads, however, the binding is no longer always 1:1 between an EAL pthread and a specified physical CPU. The EAL pthread may have affinity to a CPU set, and as such the *_lcore_id* will not be the same as the CPU ID. For this reason, there is an EAL long option '–lcores' defined to assign the CPU affinity of lcores. For a specified lcore ID or ID group, the option allows setting the CPU set for that EAL pthread.

The format pattern: –lcores='<lcore_set>[@cpu_set][,<lcore_set>[@cpu_set],...]'

'lcore_set' and 'cpu_set' can be a single number, range or a group.

A number is a "digit([0-9]+)"; a range is "<number>-<number>"; a group is "(<number|range>[,<number|range>,...])".

If a '@cpu_set' value is not supplied, the value of 'cpu_set' will default to the value of 'lcore_set'.

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

Using this option, for each given lcore ID, the associated CPUs can be assigned. It's also compatible with the pattern of corelist('-l') option.

3.3.2 non-EAL pthread support

It is possible to use the DPDK execution context with any user pthread (aka. Non-EAL pthreads). In a non-EAL pthread, the *_lcore_id* is always LCORE_ID_ANY which identifies that it is not an EAL thread with a valid, unique, *_lcore_id*. Some libraries will use an alternative unique ID (e.g. TID), some will not be impacted at all, and some will work but with limitations (e.g. timer and mempool libraries).

All these impacts are mentioned in *Known Issues* section.

3.3.3 Public Thread API

There are two public APIs `rte_thread_set_affinity()` and `rte_pthread_get_affinity()` introduced for threads. When they're used in any pthread context, the Thread Local Storage(TLS) will be set/get.

Those TLS include *_cpuset* and *_socket_id*:

- *_cpuset* stores the CPUs bitmap to which the pthread is affinitized.
- *_socket_id* stores the NUMA node of the CPU set. If the CPUs in CPU set belong to different NUMA node, the *_socket_id* will be set to SOCKET_ID_ANY.

3.3.4 Known Issues

- rte_mempool

  The rte_mempool uses a per-lcore cache inside the mempool. For non-EAL pthreads, `rte_lcore_id()` will not return a valid number. So for now, when rte_mempool is used with non-EAL pthreads, the put/get operations will bypass the default mempool cache and there is a performance penalty because of this bypass. Only user-owned external caches can be used in a non-EAL context in conjunction with `rte_mempool_generic_put()` and `rte_mempool_generic_get()` that accept an explicit cache parameter.

- rte_ring

  rte_ring supports multi-producer enqueue and multi-consumer dequeue. However, it is non-preemptive, this has a knock on effect of making rte_mempool non-preemptable.

  Note: The "non-preemptive" constraint means:

  - a pthread doing multi-producers enqueues on a given ring must not be preempted by another pthread doing a multi-producer enqueue on the same ring.
  - a pthread doing multi-consumers dequeues on a given ring must not be preempted by another pthread doing a multi-consumer dequeue on the same ring.

  Bypassing this constraint may cause the 2nd pthread to spin until the 1st one is scheduled again. Moreover, if the 1st pthread is preempted by a context that has an higher priority, it may even cause a dead lock.

  This does not mean it cannot be used, simply, there is a need to narrow down the situation when it is used by multi-pthread on the same core.

  1. It CAN be used for any single-producer or single-consumer situation.

2. It MAY be used by multi-producer/consumer pthread whose scheduling policy are all SCHED_OTHER(cfs). User SHOULD be aware of the performance penalty before using it.
3. It MUST not be used by multi-producer/consumer pthreads, whose scheduling policies are SCHED_FIFO or SCHED_RR.

`RTE_RING_PAUSE_REP_COUNT` is defined for rte_ring to reduce contention. It's mainly for case 2, a yield is issued after number of times pause repeat.

It adds a sched_yield() syscall if the thread spins for too long while waiting on the other thread to finish its operations on the ring. This gives the preempted thread a chance to proceed and finish with the ring enqueue/dequeue operation.

- rte_timer

  Running `rte_timer_manager()` on a non-EAL pthread is not allowed. However, resetting/stopping the timer from a non-EAL pthread is allowed.

- rte_log

  In non-EAL pthreads, there is no per thread loglevel and logtype, global loglevels are used.

- misc

  The debug statistics of rte_ring, rte_mempool and rte_timer are not supported in a non-EAL pthread.

3.3.5 cgroup control

The following is a simple example of cgroup control usage, there are two pthreads(t0 and t1) doing packet I/O on the same core ($CPU). We expect only 50% of CPU spend on packet IO.

```
mkdir /sys/fs/cgroup/cpu/pkt_io
mkdir /sys/fs/cgroup/cpuset/pkt_io

echo $cpu > /sys/fs/cgroup/cpuset/cpuset.cpus

echo $t0 > /sys/fs/cgroup/cpu/pkt_io/tasks
echo $t0 > /sys/fs/cgroup/cpuset/pkt_io/tasks

echo $t1 > /sys/fs/cgroup/cpu/pkt_io/tasks
echo $t1 > /sys/fs/cgroup/cpuset/pkt_io/tasks

cd /sys/fs/cgroup/cpu/pkt_io
echo 100000 > pkt_io/cpu.cfs_period_us
echo  50000 > pkt_io/cpu.cfs_quota_us
```

3.4 Malloc

The EAL provides a malloc API to allocate any-sized memory.

The objective of this API is to provide malloc-like functions to allow allocation from hugepage memory and to facilitate application porting. The *DPDK API Reference* manual describes the available functions.

Typically, these kinds of allocations should not be done in data plane processing because they are slower than pool-based allocation and make use of locks within the allocation and free paths. However, they can be used in configuration code.

Refer to the rte_malloc() function description in the *DPDK API Reference* manual for more information.

3.4.1 Cookies

When CONFIG_RTE_MALLOC_DEBUG is enabled, the allocated memory contains overwrite protection fields to help identify buffer overflows.

3.4.2 Alignment and NUMA Constraints

The rte_malloc() takes an align argument that can be used to request a memory area that is aligned on a multiple of this value (which must be a power of two).

On systems with NUMA support, a call to the rte_malloc() function will return memory that has been allocated on the NUMA socket of the core which made the call. A set of APIs is also provided, to allow memory to be explicitly allocated on a NUMA socket directly, or by allocated on the NUMA socket where another core is located, in the case where the memory is to be used by a logical core other than on the one doing the memory allocation.

3.4.3 Use Cases

This API is meant to be used by an application that requires malloc-like functions at initialization time.

For allocating/freeing data at runtime, in the fast-path of an application, the memory pool library should be used instead.

3.4.4 Internal Implementation

Data Structures

There are two data structure types used internally in the malloc library:

- struct malloc_heap - used to track free space on a per-socket basis
- struct malloc_elem - the basic element of allocation and free-space tracking inside the library.

Structure: malloc_heap

The malloc_heap structure is used to manage free space on a per-socket basis. Internally, there is one heap structure per NUMA node, which allows us to allocate memory to a thread based on the NUMA node on which this thread runs. While this does not guarantee that the memory will be used on that NUMA node, it is no worse than a scheme where the memory is always allocated on a fixed or random node.

The key fields of the heap structure and their function are described below (see also diagram above):

- lock - the lock field is needed to synchronize access to the heap. Given that the free space in the heap is tracked using a linked list, we need a lock to prevent two threads manipulating the list at the same time.
- free_head - this points to the first element in the list of free nodes for this malloc heap.

Note: The malloc_heap structure does not keep track of in-use blocks of memory, since these are never touched except when they are to be freed again - at which point the pointer to the block is an input to the free() function.

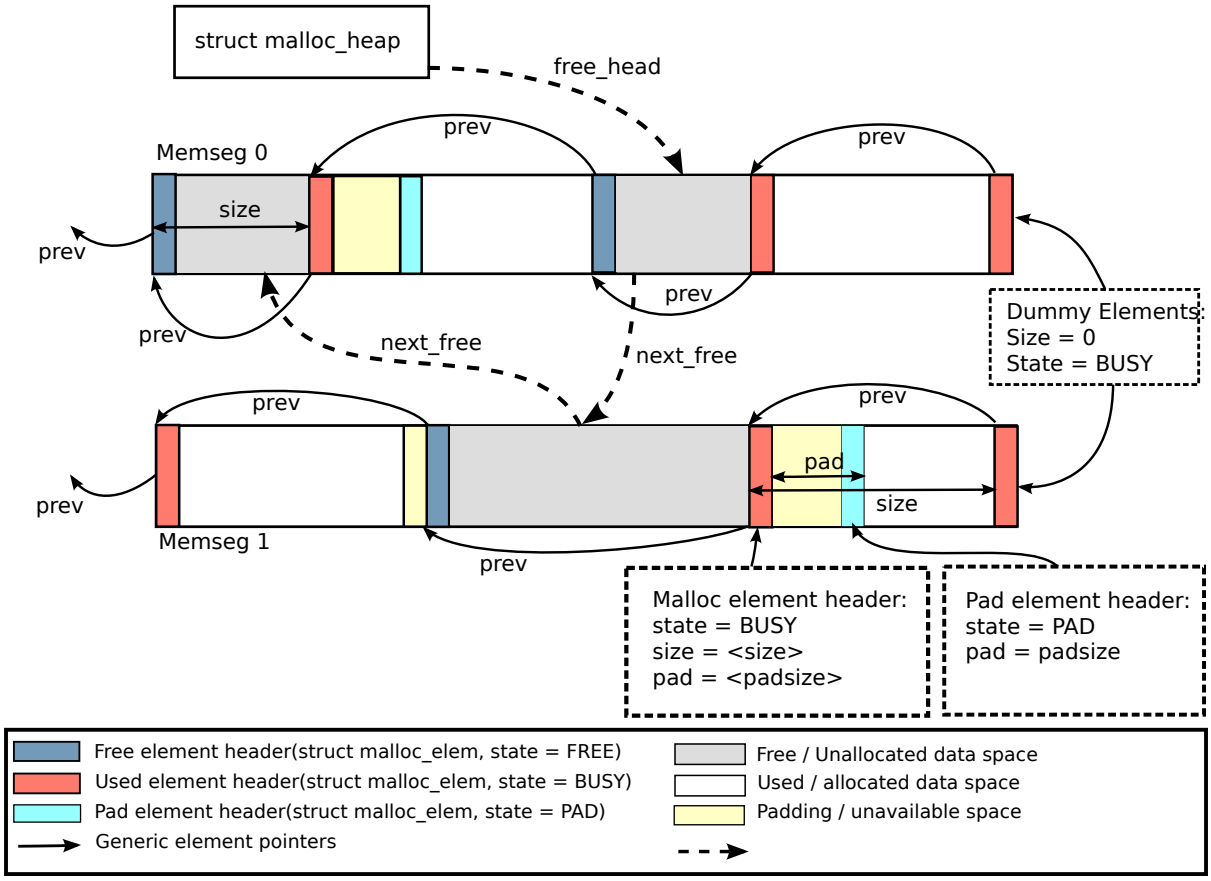


Fig. 3.2: Example of a malloc heap and malloc elements within the malloc library

Structure: malloc_elem

The malloc_elem structure is used as a generic header structure for various blocks of memory. It is used in three different ways - all shown in the diagram above:

1. As a header on a block of free or allocated memory - normal case
2. As a padding header inside a block of memory
3. As an end-of-memseg marker

The most important fields in the structure and how they are used are described below.

Note: If the usage of a particular field in one of the above three usages is not described, the field can be assumed to have an undefined value in that situation, for example, for padding headers only the "state" and "pad" fields have valid values.

- heap - this pointer is a reference back to the heap structure from which this block was allocated. It is used for normal memory blocks when they are being freed, to add the newly-freed block to the heap's free-list.
- prev - this pointer points to the header element/block in the memseg immediately behind the current one. When freeing a block, this pointer is used to reference the previous block to check if that block is also free. If so, then the two free blocks are merged to form a single larger block.
- next_free - this pointer is used to chain the free-list of unallocated memory blocks together. It is only used in normal memory blocks; on `malloc()` to find a suitable free block to allocate and on `free()` to add the newly freed element to the free-list.
- state - This field can have one of three values: `FREE`, `BUSY` or `PAD`. The former two are to indicate the allocation state of a normal memory block and the latter is to indicate that the element structure is a dummy structure at the end of the start-of-block padding, i.e. where the start of the data within a block is not at the start of the block itself, due to alignment constraints. In that case, the pad header is used to locate the actual malloc element header for the block. For the end-of-memseg structure, this is always a `BUSY` value, which ensures that no element, on being freed, searches beyond the end of the memseg for other blocks to merge with into a larger free area.
- pad - this holds the length of the padding present at the start of the block. In the case of a normal block header, it is added to the address of the end of the header to give the address of the start of the data area, i.e. the value passed back to the application on a malloc. Within a dummy header inside the padding, this same value is stored, and is subtracted from the address of the dummy header to yield the address of the actual block header.
- size - the size of the data block, including the header itself. For end-of-memseg structures, this size is given as zero, though it is never actually checked. For normal blocks which are being freed, this size value is used in place of a "next" pointer to identify the location of the next block of memory that in the case of being `FREE`, the two free blocks can be merged into one.

Memory Allocation

On EAL initialization, all memsegs are setup as part of the malloc heap. This setup involves placing a dummy structure at the end with `BUSY` state, which may contain a sentinel value if `CONFIG_RTE_MALLOC_DEBUG` is enabled, and a proper *element header* with `FREE` at the start for each memseg. The `FREE` element is then added to the `free_list` for the malloc heap.

When an application makes a call to a malloc-like function, the malloc function will first index the `lcore_config` structure for the calling thread, and determine the NUMA node of that thread. The NUMA node is used to index the array of `malloc_heap` structures which is passed as a parameter to the `heap_alloc()` function, along with the requested size, type, alignment and boundary parameters.

The `heap_alloc()` function will scan the free_list of the heap, and attempt to find a free block suitable for storing data of the requested size, with the requested alignment and boundary

constraints.

When a suitable free element has been identified, the pointer to be returned to the user is calculated. The cache-line of memory immediately preceding this pointer is filled with a struct malloc_elem header. Because of alignment and boundary constraints, there could be free space at the start and/or end of the element, resulting in the following behavior:

1. Check for trailing space. If the trailing space is big enough, i.e. > 128 bytes, then the free element is split. If it is not, then we just ignore it (wasted space).
2. Check for space at the start of the element. If the space at the start is small, i.e. <=128 bytes, then a pad header is used, and the remaining space is wasted. If, however, the remaining space is greater, then the free element is split.

The advantage of allocating the memory from the end of the existing element is that no adjustment of the free list needs to take place - the existing element on the free list just has its size pointer adjusted, and the following element has its "prev" pointer redirected to the newly created element.

Freeing Memory

To free an area of memory, the pointer to the start of the data area is passed to the free function. The size of the `malloc_elem` structure is subtracted from this pointer to get the element header for the block. If this header is of type `PAD` then the pad length is further subtracted from the pointer to get the proper element header for the entire block.

From this element header, we get pointers to the heap from which the block was allocated and to where it must be freed, as well as the pointer to the previous element, and via the size field, we can calculate the pointer to the next element. These next and previous elements are then checked to see if they are also `FREE`, and if so, they are merged with the current element. This means that we can never have two `FREE` memory blocks adjacent to one another, as they are always merged into a single block.

CHAPTER

FOUR

RING LIBRARY

The ring allows the management of queues. Instead of having a linked list of infinite size, the rte_ring has the following properties:

- FIFO
- Maximum size is fixed, the pointers are stored in a table
- Lockless implementation
- Multi-consumer or single-consumer dequeue
- Multi-producer or single-producer enqueue
- Bulk dequeue - Dequeues the specified count of objects if successful; otherwise fails
- Bulk enqueue - Enqueues the specified count of objects if successful; otherwise fails
- Burst dequeue - Dequeue the maximum available objects if the specified count cannot be fulfilled
- Burst enqueue - Enqueue the maximum available objects if the specified count cannot be fulfilled

The advantages of this data structure over a linked list queue are as follows:

- Faster; only requires a single Compare-And-Swap instruction of sizeof(void *) instead of several double-Compare-And-Swap instructions.
- Simpler than a full lockless queue.
- Adapted to bulk enqueue/dequeue operations. As pointers are stored in a table, a dequeue of several objects will not produce as many cache misses as in a linked queue. Also, a bulk dequeue of many objects does not cost more than a dequeue of a simple object.

The disadvantages:

- Size is fixed
- Having many rings costs more in terms of memory than a linked list queue. An empty ring contains at least N pointers.

A simplified representation of a Ring is shown in with consumer and producer head and tail pointers to objects stored in the data structure.

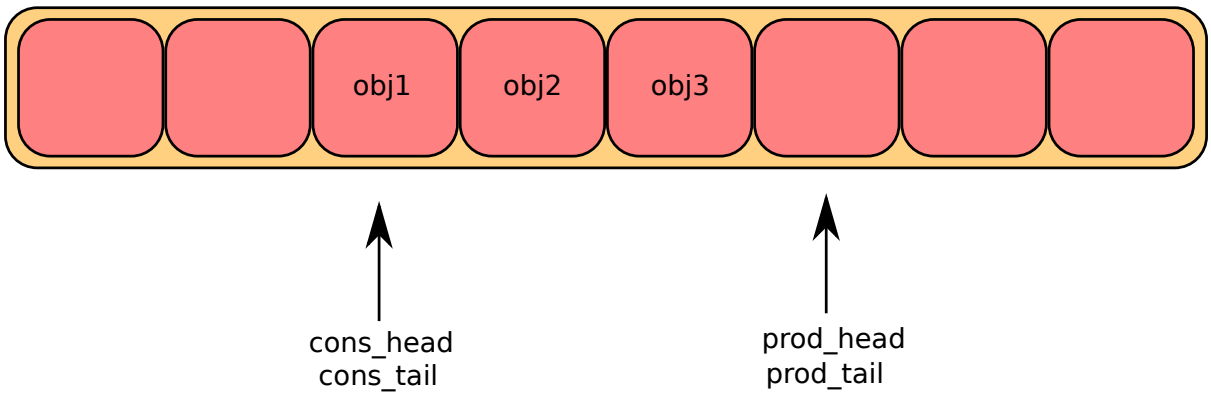


Fig. 4.1: Ring Structure

4.1 References for Ring Implementation in FreeBSD*

The following code was added in FreeBSD 8.0, and is used in some network device drivers (at least in Intel drivers):

- bufring.h in FreeBSD
- bufring.c in FreeBSD

4.2 Lockless Ring Buffer in Linux*

The following is a link describing the Linux Lockless Ring Buffer Design.

4.3 Additional Features

4.3.1 Name

A ring is identified by a unique name. It is not possible to create two rings with the same name (rte_ring_create() returns NULL if this is attempted).

4.3.2 Water Marking

The ring can have a high water mark (threshold). Once an enqueue operation reaches the high water mark, the producer is notified, if the water mark is configured.

This mechanism can be used, for example, to exert a back pressure on I/O to inform the LAN to PAUSE.

4.3.3 Debug

When debug is enabled (CONFIG_RTE_LIBRTE_RING_DEBUG is set), the library stores some per-ring statistic counters about the number of enqueues/dequeues. These statistics are per-core to avoid concurrent accesses or atomic operations.

4.4 Use Cases

Use cases for the Ring library include:

- Communication between applications in the DPDK
- Used by memory pool allocator

4.5 Anatomy of a Ring Buffer

This section explains how a ring buffer operates. The ring structure is composed of two head and tail couples; one is used by producers and one is used by the consumers. The figures of the following sections refer to them as prod_head, prod_tail, cons_head and cons_tail.

Each figure represents a simplified state of the ring, which is a circular buffer. The content of the function local variables is represented on the top of the figure, and the content of ring structure is represented on the bottom of the figure.

4.5.1 Single Producer Enqueue

This section explains what occurs when a producer adds an object to the ring. In this example, only the producer head and tail (prod_head and prod_tail) are modified, and there is only one producer.

The initial state is to have a prod_head and prod_tail pointing at the same location.

Enqueue First Step

First, *ring->prod_head* and ring->cons_tail are copied in local variables. The prod_next local variable points to the next element of the table, or several elements after in case of bulk enqueue.

If there is not enough room in the ring (this is detected by checking cons_tail), it returns an error.

Enqueue Second Step

The second step is to modify *ring->prod_head* in ring structure to point to the same location as prod_next.

A pointer to the added object is copied in the ring (obj4).

Enqueue Last Step

Once the object is added in the ring, ring->prod_tail in the ring structure is modified to point to the same location as *ring->prod_head*. The enqueue operation is finished.

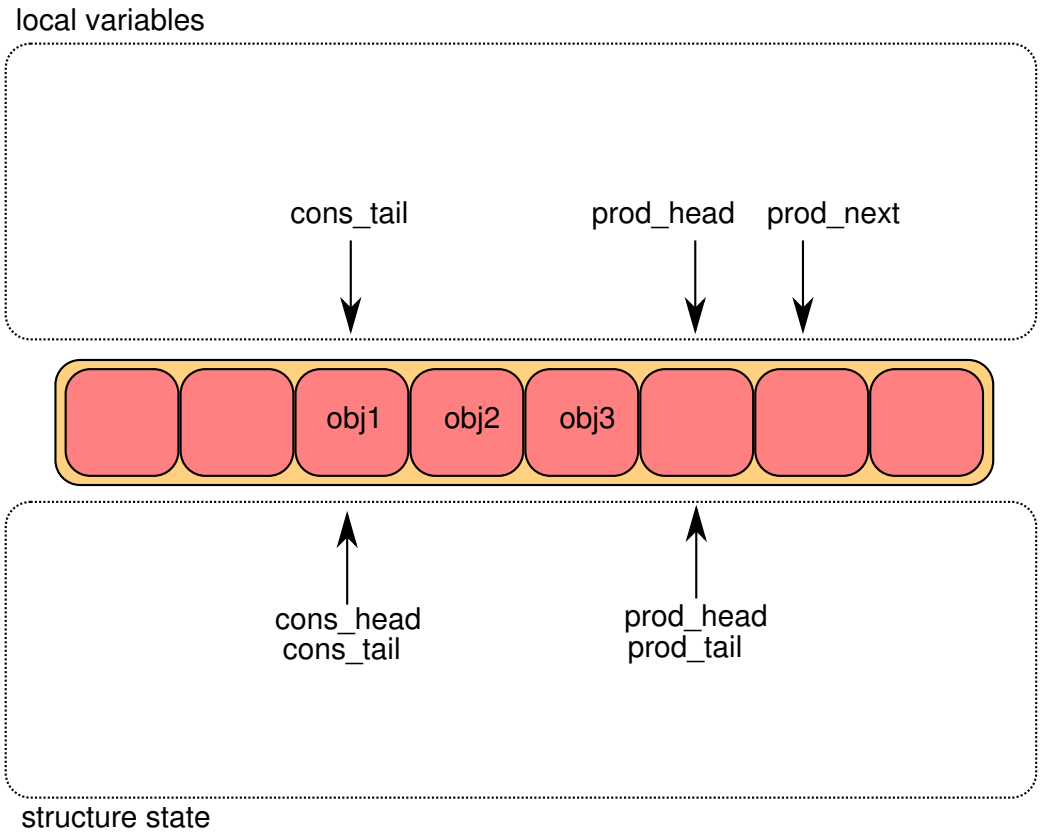


Fig. 4.2: Enqueue first step

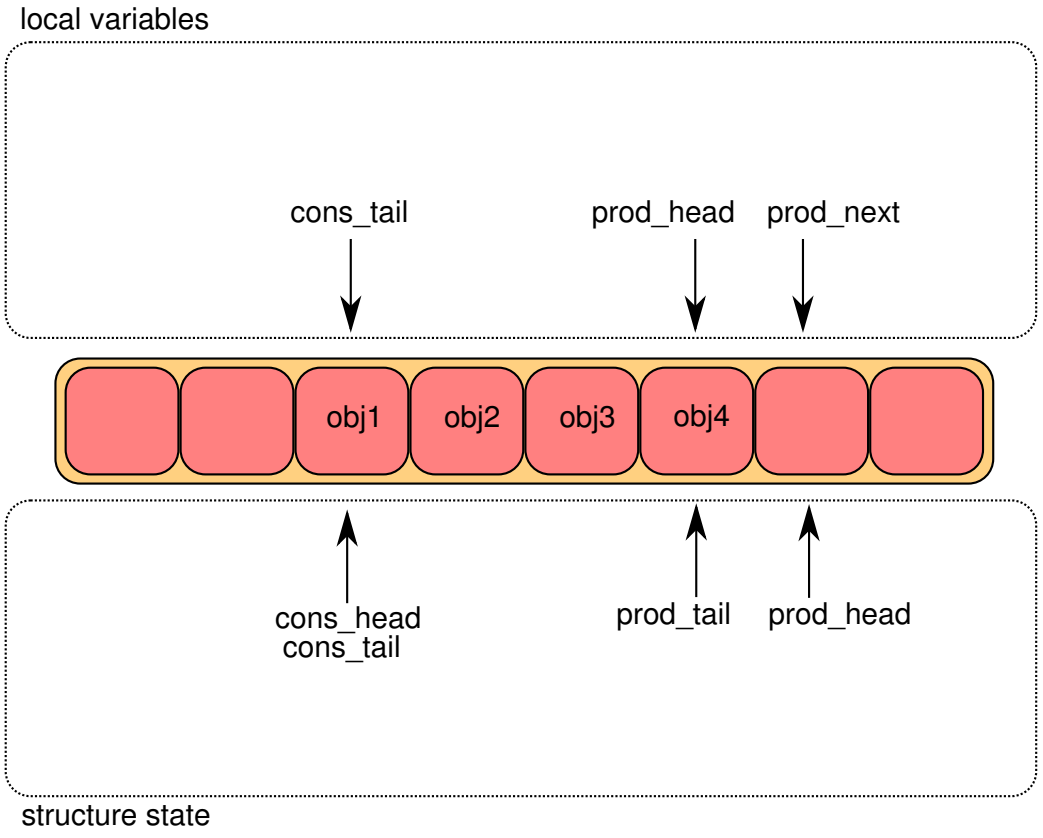


Fig. 4.3: Enqueue second step

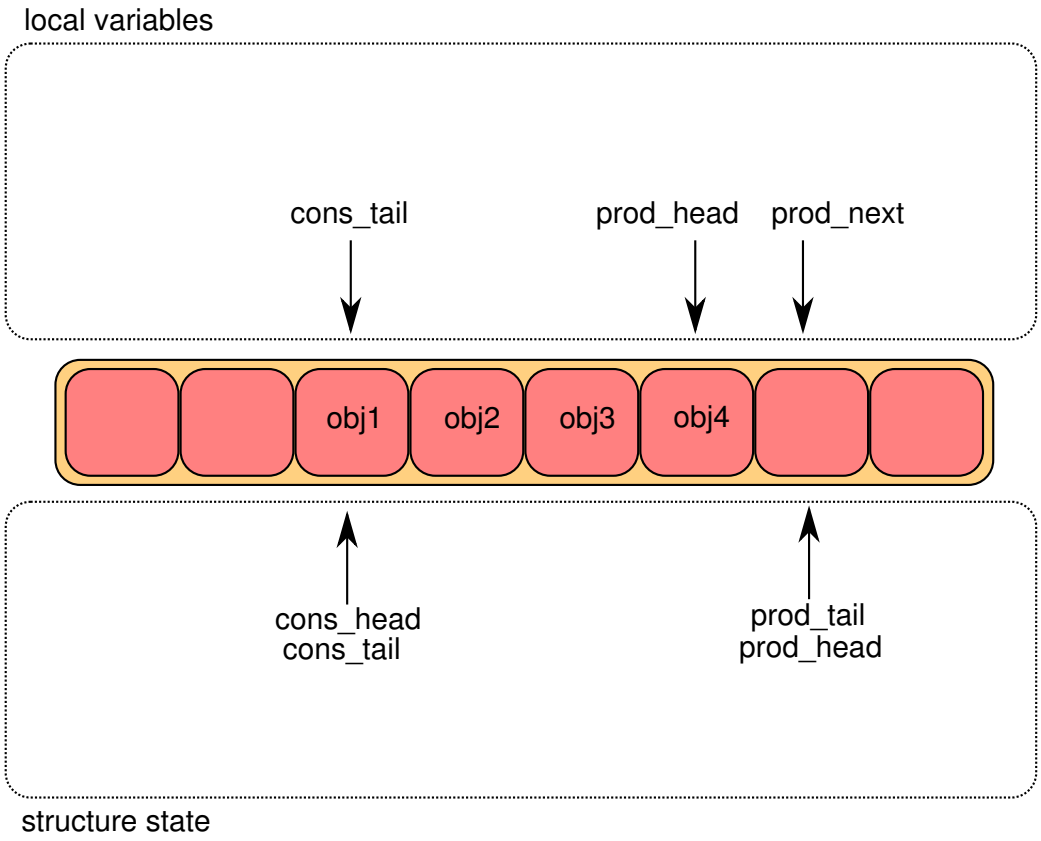


Fig. 4.4: Enqueue last step

4.5.2 Single Consumer Dequeue

This section explains what occurs when a consumer dequeues an object from the ring. In this example, only the consumer head and tail (cons_head and cons_tail) are modified and there is only one consumer.

The initial state is to have a cons_head and cons_tail pointing at the same location.

Dequeue First Step

First, ring->cons_head and ring->prod_tail are copied in local variables. The cons_next local variable points to the next element of the table, or several elements after in the case of bulk dequeue.

If there are not enough objects in the ring (this is detected by checking prod_tail), it returns an error.

Dequeue Second Step

The second step is to modify ring->cons_head in the ring structure to point to the same location as cons_next.

The pointer to the dequeued object (obj1) is copied in the pointer given by the user.

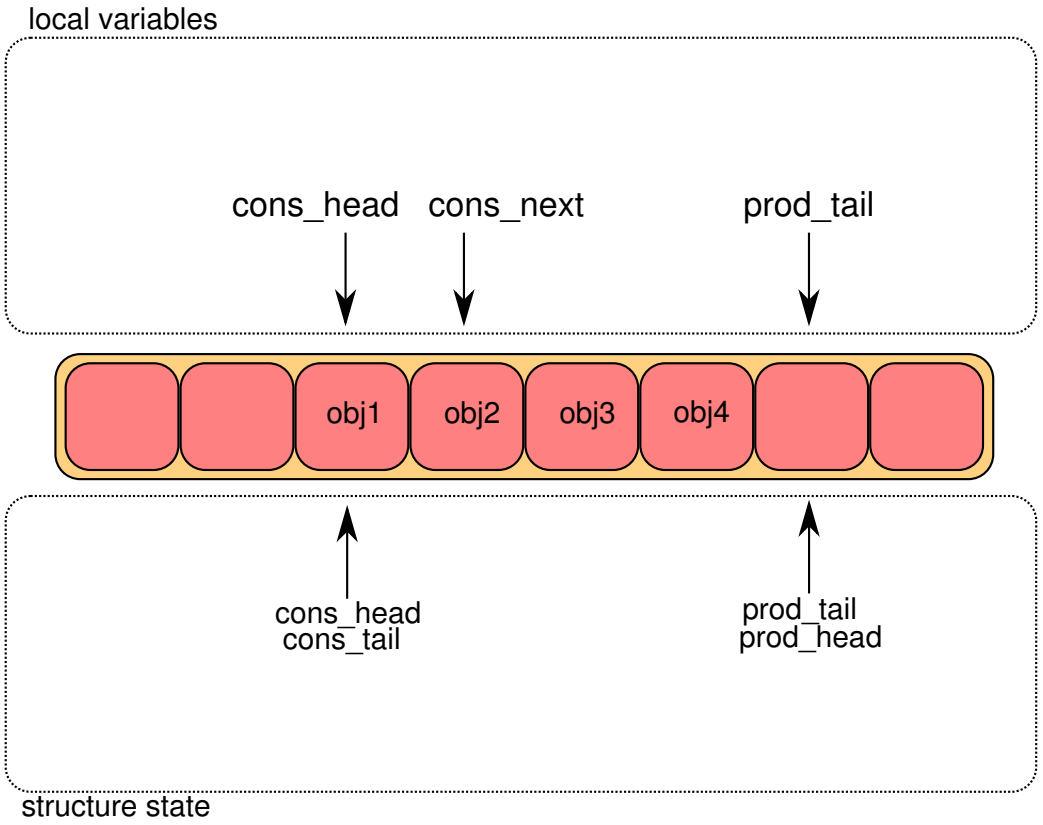


Fig. 4.5: Dequeue last step

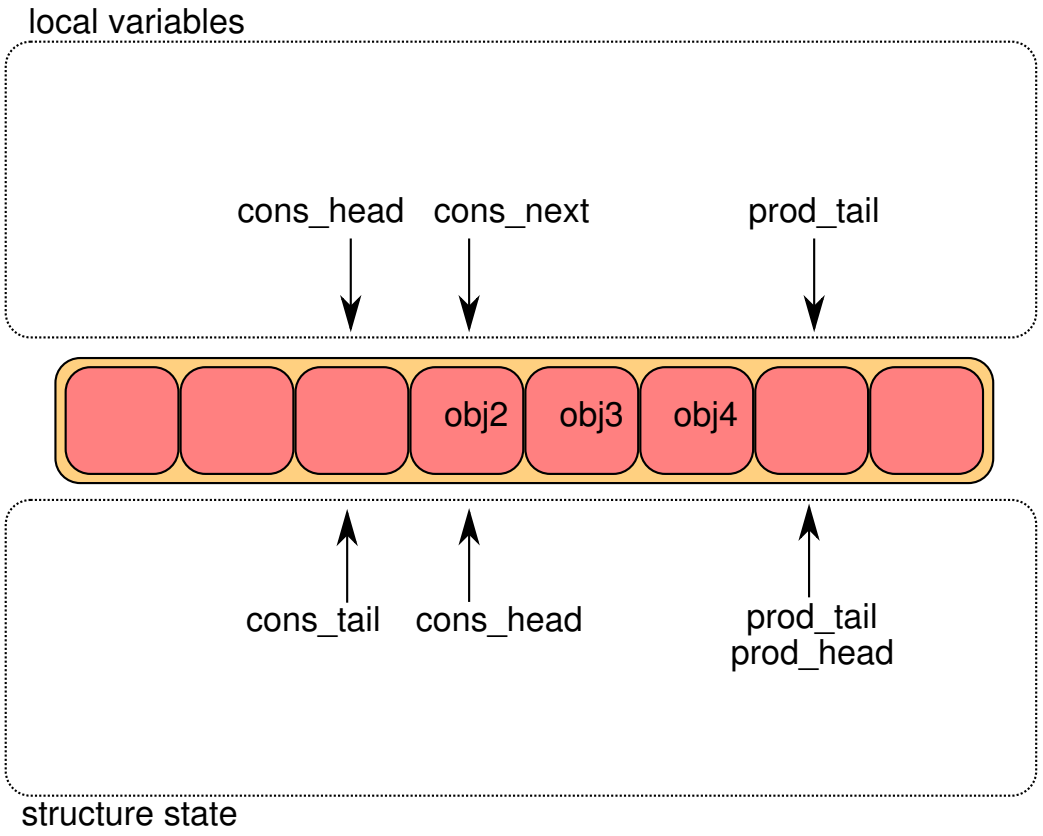


Fig. 4.6: Dequeue second step

Dequeue Last Step

Finally, ring->cons_tail in the ring structure is modified to point to the same location as ring->cons_head. The dequeue operation is finished.

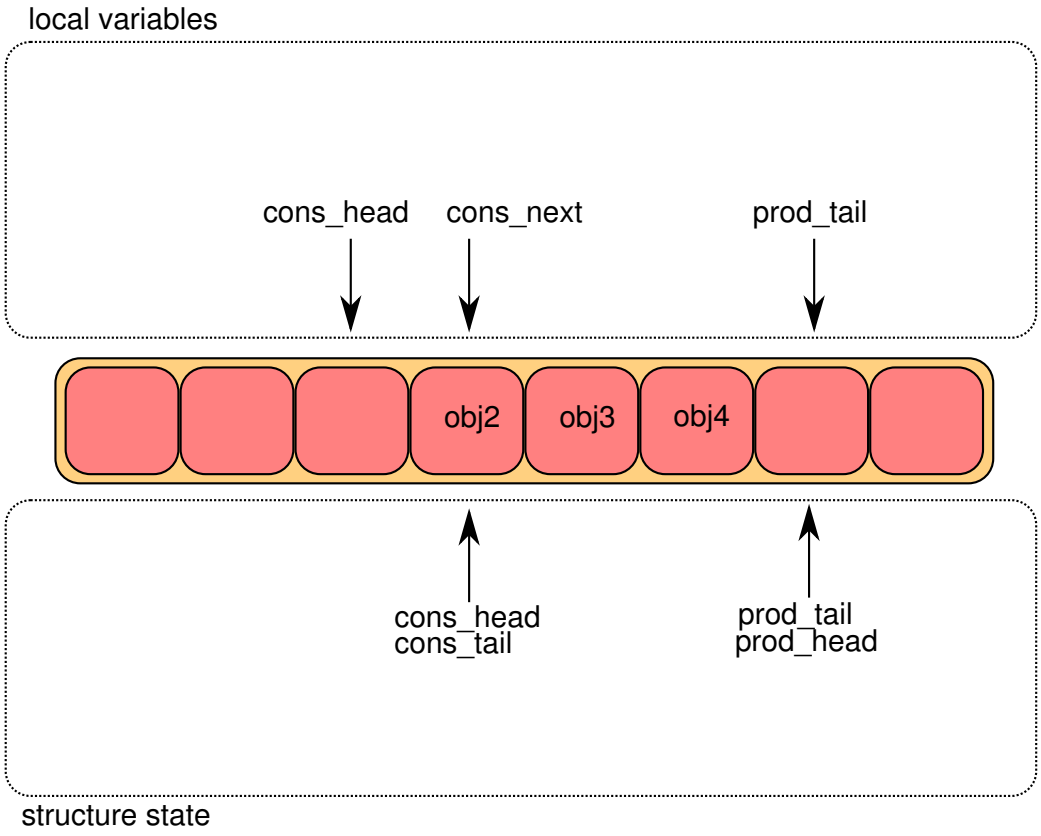


Fig. 4.7: Dequeue last step

4.5.3 Multiple Producers Enqueue

This section explains what occurs when two producers concurrently add an object to the ring. In this example, only the producer head and tail (prod_head and prod_tail) are modified.

The initial state is to have a prod_head and prod_tail pointing at the same location.

Multiple Producers Enqueue First Step

On both cores, *ring->prod_head* and ring->cons_tail are copied in local variables. The prod_next local variable points to the next element of the table, or several elements after in the case of bulk enqueue.

If there is not enough room in the ring (this is detected by checking cons_tail), it returns an error.

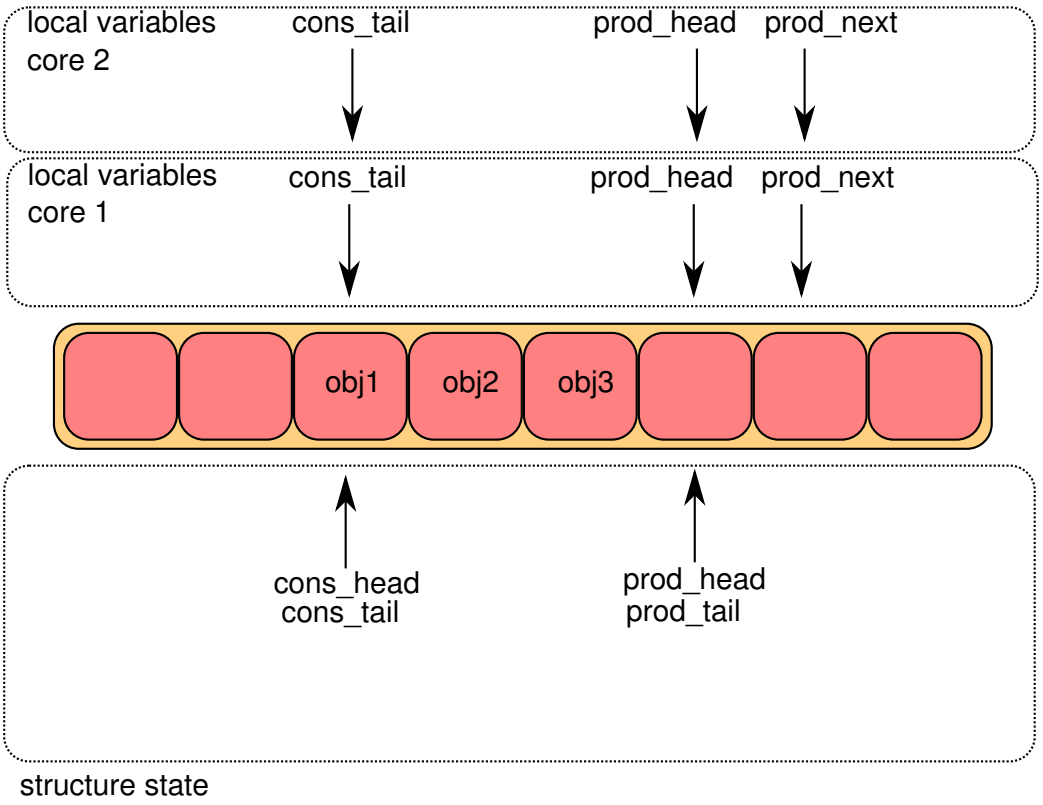


Fig. 4.8: Multiple producer enqueue first step

Multiple Producers Enqueue Second Step

The second step is to modify ring->prod_head in the ring structure to point to the same location as prod_next. This operation is done using a Compare And Swap (CAS) instruction, which does the following operations atomically:

- If ring->prod_head is different to local variable prod_head, the CAS operation fails, and the code restarts at first step.
- Otherwise, ring->prod_head is set to local prod_next, the CAS operation is successful, and processing continues.

In the figure, the operation succeeded on core 1, and step one restarted on core 2.

Multiple Producers Enqueue Third Step

The CAS operation is retried on core 2 with success.

The core 1 updates one element of the ring(obj4), and the core 2 updates another one (obj5).

Multiple Producers Enqueue Fourth Step

Each core now wants to update ring->prod_tail. A core can only update it if ring->prod_tail is equal to the prod_head local variable. This is only true on core 1. The operation is finished on core 1.

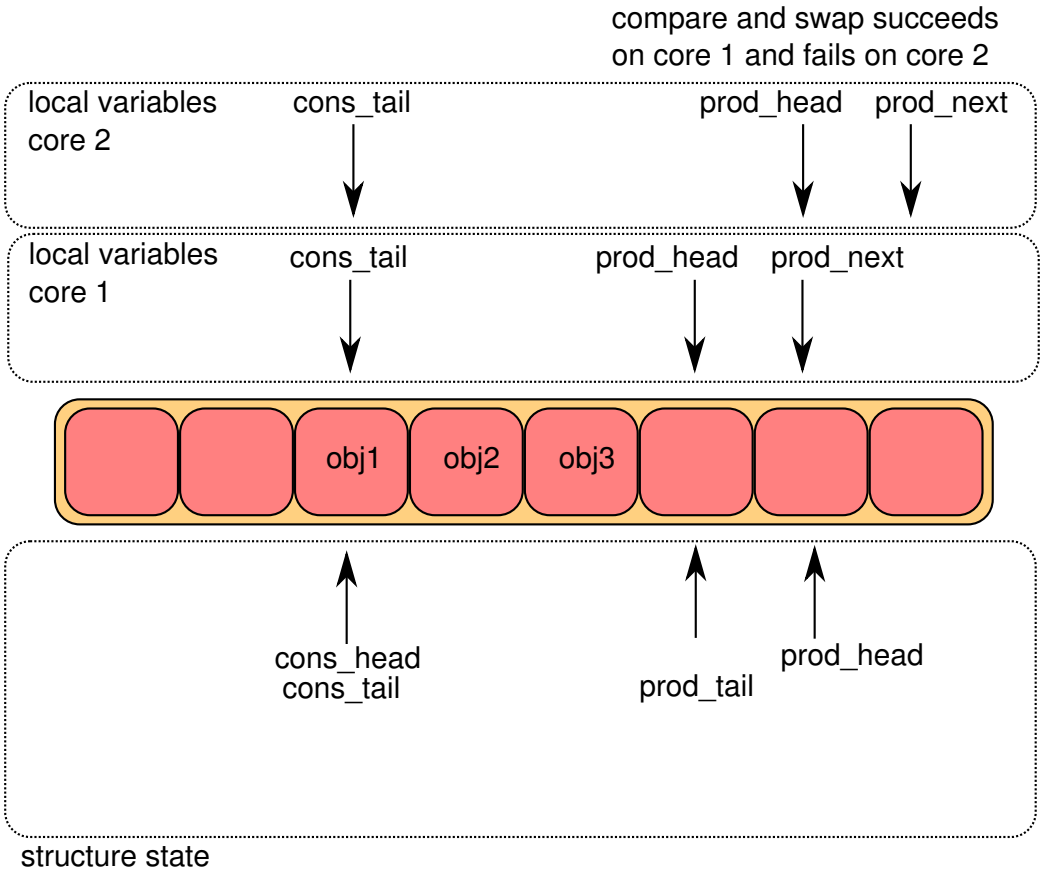


Fig. 4.9: Multiple producer enqueue second step

Multiple Producers Enqueue Last Step

Once ring->prod_tail is updated by core 1, core 2 is allowed to update it too. The operation is also finished on core 2.

4.5.4 Modulo 32-bit Indexes

In the preceding figures, the prod_head, prod_tail, cons_head and cons_tail indexes are represented by arrows. In the actual implementation, these values are not between 0 and size(ring)-1 as would be assumed. The indexes are between 0 and 2^32 -1, and we mask their value when we access the pointer table (the ring itself). 32-bit modulo also implies that operations on indexes (such as, add/subtract) will automatically do 2^32 modulo if the result overflows the 32-bit number range.

The following are two examples that help to explain how indexes are used in a ring.

Note: To simplify the explanation, operations with modulo 16-bit are used instead of modulo 32-bit. In addition, the four indexes are defined as unsigned 16-bit integers, as opposed to unsigned 32-bit integers in the more realistic case.

This ring contains 11000 entries.

This ring contains 12536 entries.

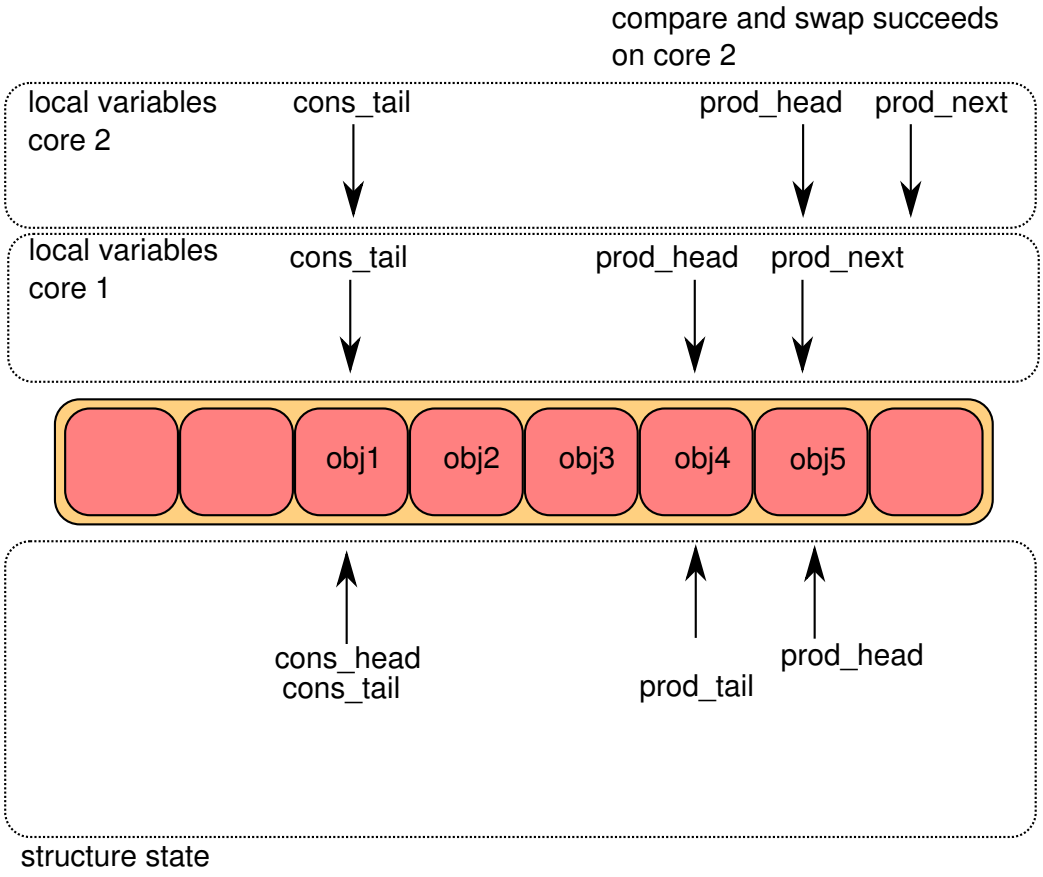


Fig. 4.10: Multiple producer enqueue third step

Note: For ease of understanding, we use modulo 65536 operations in the above examples. In real execution cases, this is redundant for low efficiency, but is done automatically when the result overflows.

The code always maintains a distance between producer and consumer between 0 and size(ring)-1. Thanks to this property, we can do subtractions between 2 index values in a modulo-32bit base: that's why the overflow of the indexes is not a problem.

At any time, entries and free_entries are between 0 and size(ring)-1, even if only the first term of subtraction has overflowed:

```
uint32_t entries = (prod_tail - cons_head);
uint32_t free_entries = (mask + cons_tail -prod_head);
```

4.6 References

- bufring.h in FreeBSD (version 8)
- bufring.c in FreeBSD (version 8)
- Linux Lockless Ring Buffer Design

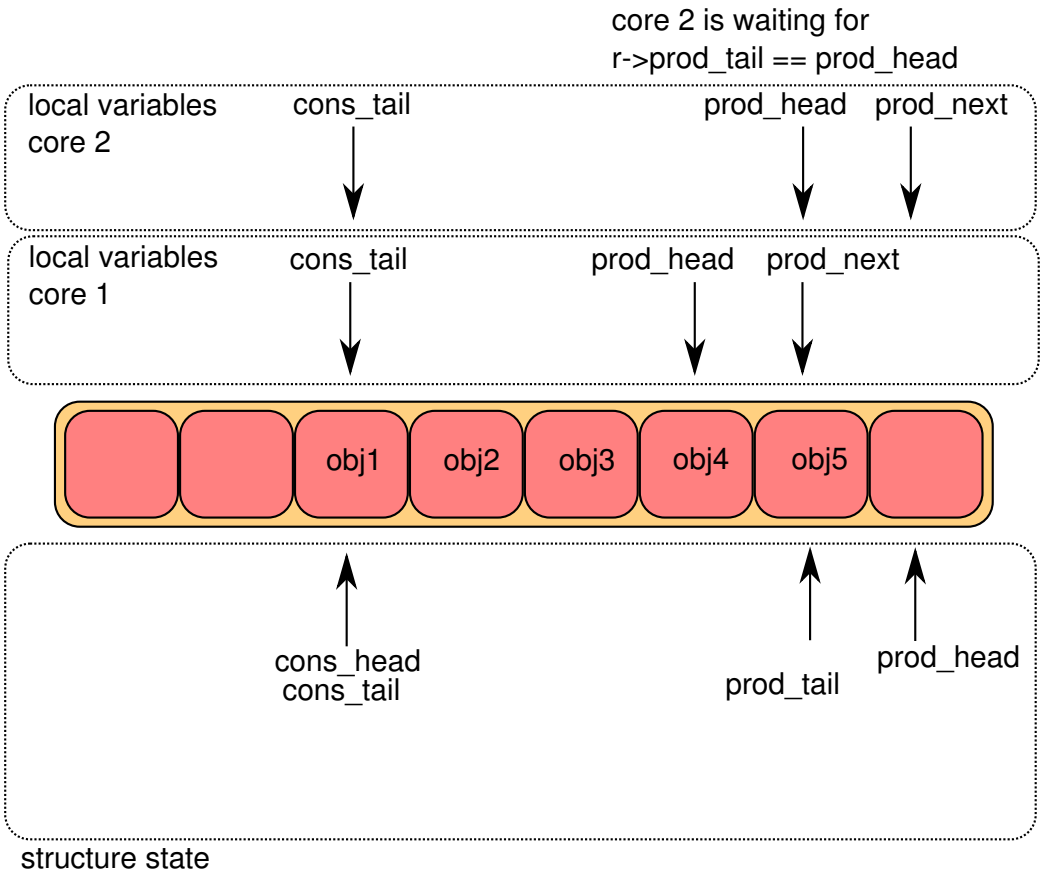


Fig. 4.11: Multiple producer enqueue fourth step

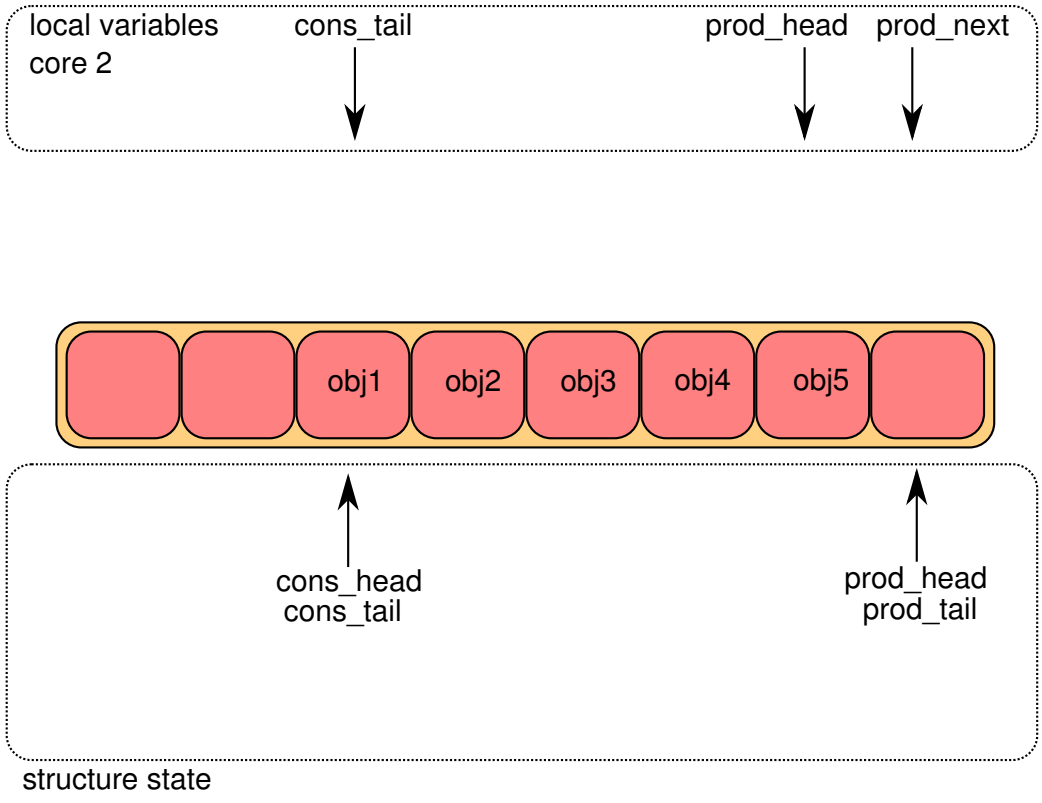


Fig. 4.12: Multiple producer enqueue last step

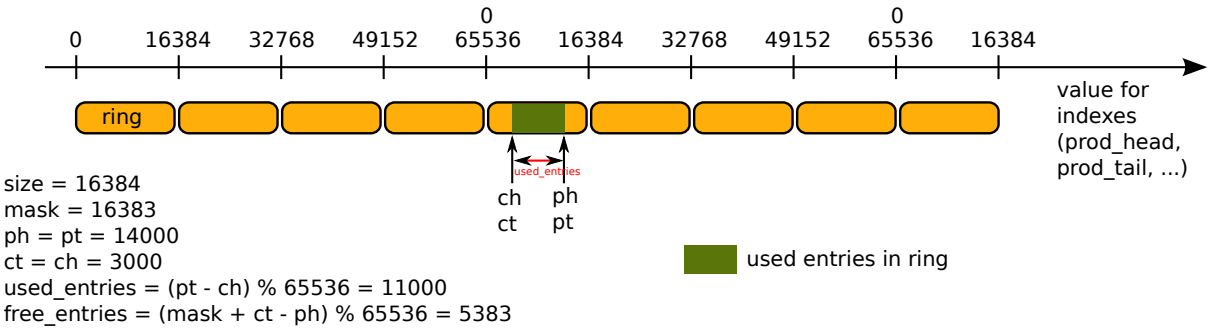


Fig. 4.13: Modulo 32-bit indexes - Example 1

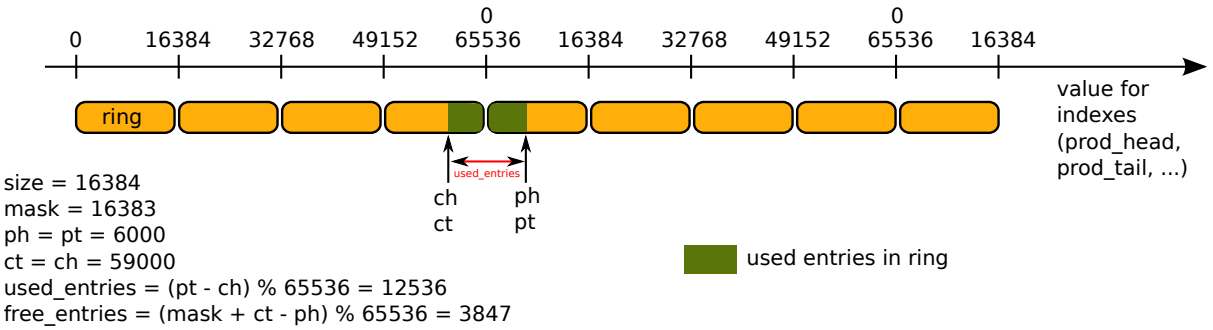


Fig. 4.14: Modulo 32-bit indexes - Example 2

CHAPTER

FIVE

MEMPOOL LIBRARY

A memory pool is an allocator of a fixed-sized object. In the DPDK, it is identified by name and uses a mempool handler to store free objects. The default mempool handler is ring based. It provides some other optional services such as a per-core object cache and an alignment helper to ensure that objects are padded to spread them equally on all DRAM or DDR3 channels.

This library is used by the *Mbuf Library*.

5.1 Cookies

In debug mode (CONFIG_RTE_LIBRTE_MEMPOOL_DEBUG is enabled), cookies are added at the beginning and end of allocated blocks. The allocated objects then contain overwrite protection fields to help debugging buffer overflows.

5.2 Stats

In debug mode (CONFIG_RTE_LIBRTE_MEMPOOL_DEBUG is enabled), statistics about get from/put in the pool are stored in the mempool structure. Statistics are per-lcore to avoid concurrent access to statistics counters.

5.3 Memory Alignment Constraints

Depending on hardware memory configuration, performance can be greatly improved by adding a specific padding between objects. The objective is to ensure that the beginning of each object starts on a different channel and rank in memory so that all channels are equally loaded.

This is particularly true for packet buffers when doing L3 forwarding or flow classification. Only the first 64 bytes are accessed, so performance can be increased by spreading the start addresses of objects among the different channels.

The number of ranks on any DIMM is the number of independent sets of DRAMs that can be accessed for the full data bit-width of the DIMM. The ranks cannot be accessed simultaneously since they share the same data path. The physical layout of the DRAM chips on the DIMM itself does not necessarily relate to the number of ranks.

When running an application, the EAL command line options provide the ability to add the number of memory channels and ranks.

Note: The command line must always have the number of memory channels specified for the processor.

Examples of alignment for different DIMM architectures are shown in Fig. 5.1 and Fig. 5.2.

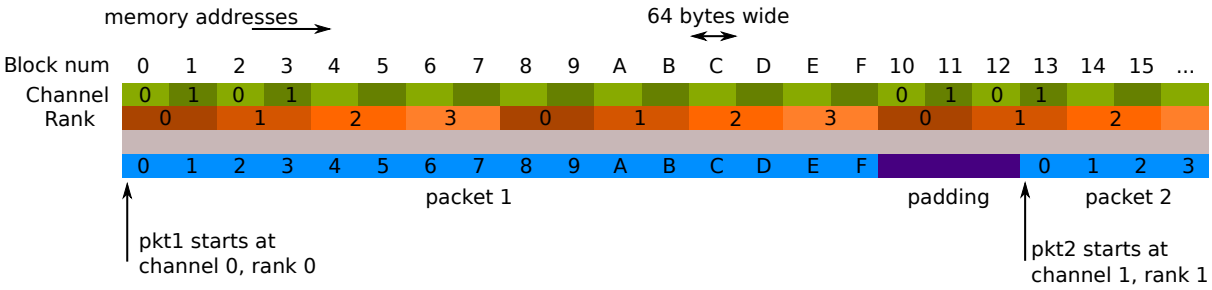


Fig. 5.1: Two Channels and Quad-ranked DIMM Example

In this case, the assumption is that a packet is 16 blocks of 64 bytes, which is not true.

The Intel® 5520 chipset has three channels, so in most cases, no padding is required between objects (except for objects whose size are n x 3 x 64 bytes blocks).

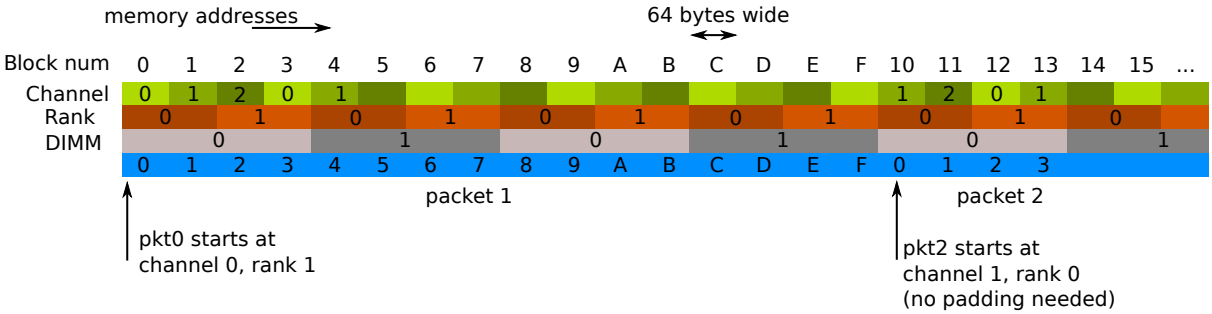


Fig. 5.2: Three Channels and Two Dual-ranked DIMM Example

When creating a new pool, the user can specify to use this feature or not.

5.4 Local Cache

In terms of CPU usage, the cost of multiple cores accessing a memory pool's ring of free buffers may be high since each access requires a compare-and-set (CAS) operation. To avoid having too many access requests to the memory pool's ring, the memory pool allocator can maintain a per-core cache and do bulk requests to the memory pool's ring, via the cache with many fewer locks on the actual memory pool structure. In this way, each core has full access to its own cache (with locks) of free objects and only when the cache fills does the core need to shuffle some of the free objects back to the pools ring or obtain more objects when the cache is empty.

While this may mean a number of buffers may sit idle on some core's cache, the speed at which a core can access its own cache for a specific memory pool without locks provides performance gains.

The cache is composed of a small, per-core table of pointers and its length (used as a stack). This internal cache can be enabled or disabled at creation of the pool.

The maximum size of the cache is static and is defined at compilation time (CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE).

Fig. 5.3 shows a cache in operation.

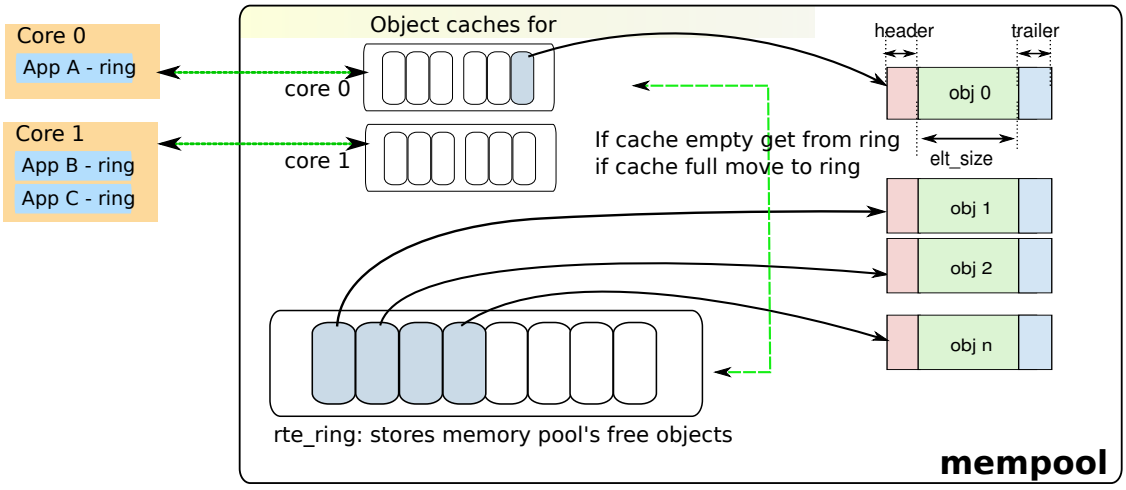


Fig. 5.3: A mempool in Memory with its Associated Ring

Alternatively to the internal default per-lcore local cache, an application can create and manage external caches through the `rte_mempool_cache_create()`, `rte_mempool_cache_free()` and `rte_mempool_cache_flush()` calls. These user-owned caches can be explicitly passed to `rte_mempool_generic_put()` and `rte_mempool_generic_get()`. The `rte_mempool_default_cache()` call returns the default internal cache if any. In contrast to the default caches, user-owned caches can be used by non-EAL threads too.

5.5 Mempool Handlers

This allows external memory subsystems, such as external hardware memory management systems and software based memory allocators, to be used with DPDK.

There are two aspects to a mempool handler.

- Adding the code for your new mempool operations (ops). This is achieved by adding a new mempool ops code, and using the `MEMPOOL_REGISTER_OPS` macro.
- Using the new API to call `rte_mempool_create_empty()` and `rte_mempool_set_ops_byname()` to create a new mempool and specifying which ops to use.

Several different mempool handlers may be used in the same application. A new mempool can be created by using the `rte_mempool_create_empty()` function, then using `rte_mempool_set_ops_byname()` to point the mempool to the relevant mempool handler callback (ops) structure.

Legacy applications may continue to use the old `rte_mempool_create()` API call, which uses a ring based mempool handler by default. These applications will need to be modified to

use a new mempool handler.

For applications that use `rte_pktmbuf_create()`, there is a config setting (`RTE_MBUF_DEFAULT_MEMPOOL_OPS`) that allows the application to make use of an alternative mempool handler.

5.6 Use Cases

All allocations that require a high level of performance should use a pool-based memory allocator. Below are some examples:

- *Mbuf Library*
- *Environment Abstraction Layer* , for logging service
- Any application that needs to allocate fixed-sized objects in the data plane and that will be continuously utilized by the system.

CHAPTER

SIX

MBUF LIBRARY

The mbuf library provides the ability to allocate and free buffers (mbufs) that may be used by the DPDK application to store message buffers. The message buffers are stored in a mempool, using the *Mempool Library*.

A rte_mbuf struct can carry network packet buffers or generic control buffers (indicated by the CTRL_MBUF_FLAG). This can be extended to other types. The rte_mbuf header structure is kept as small as possible and currently uses just two cache lines, with the most frequently used fields being on the first of the two cache lines.

6.1 Design of Packet Buffers

For the storage of the packet data (including protocol headers), two approaches were considered:

1. Embed metadata within a single memory buffer the structure followed by a fixed size area for the packet data.
2. Use separate memory buffers for the metadata structure and for the packet data.

The advantage of the first method is that it only needs one operation to allocate/free the whole memory representation of a packet. On the other hand, the second method is more flexible and allows the complete separation of the allocation of metadata structures from the allocation of packet data buffers.

The first method was chosen for the DPDK. The metadata contains control information such as message type, length, offset to the start of the data and a pointer for additional mbuf structures allowing buffer chaining.

Message buffers that are used to carry network packets can handle buffer chaining where multiple buffers are required to hold the complete packet. This is the case for jumbo frames that are composed of many mbufs linked together through their next field.

For a newly allocated mbuf, the area at which the data begins in the message buffer is RTE_PKTMBUF_HEADROOM bytes after the beginning of the buffer, which is cache aligned. Message buffers may be used to carry control information, packets, events, and so on between different entities in the system. Message buffers may also use their buffer pointers to point to other message buffer data sections or other structures.

Fig. 6.1 and Fig. 6.2 show some of these scenarios.

The Buffer Manager implements a fairly standard set of buffer access functions to manipulate network packets.

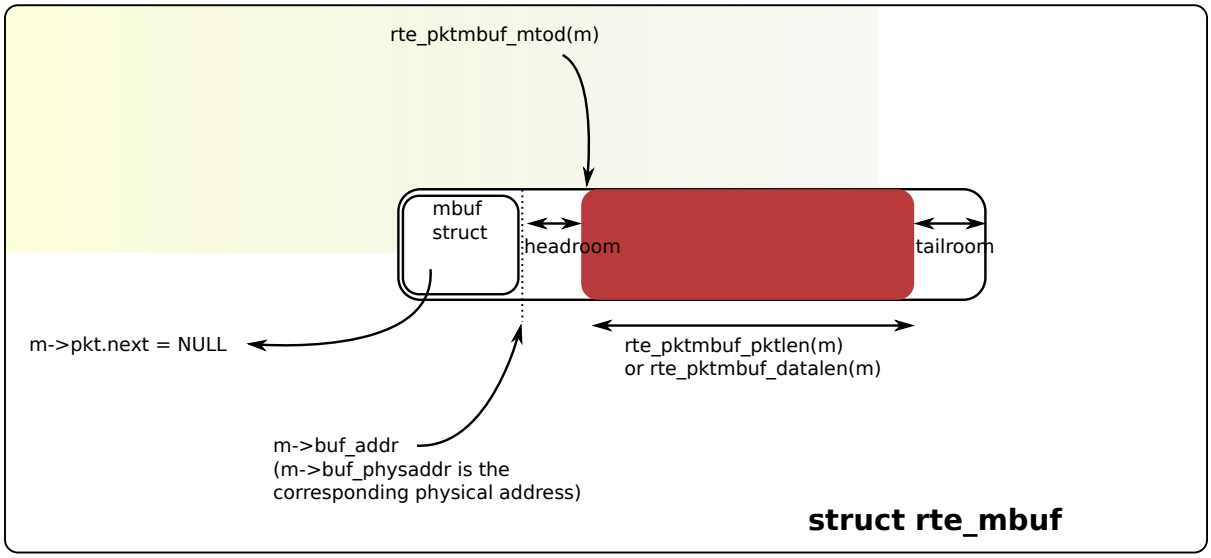


Fig. 6.1: An mbuf with One Segment

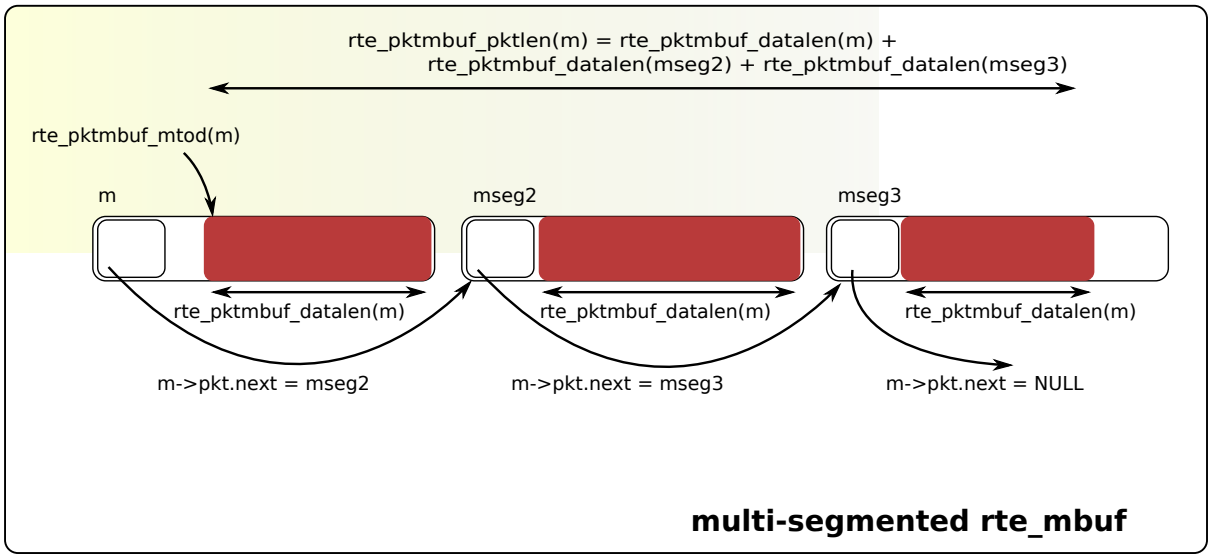


Fig. 6.2: An mbuf with Three Segments

6.2 Buffers Stored in Memory Pools

The Buffer Manager uses the *Mempool Library* to allocate buffers. Therefore, it ensures that the packet header is interleaved optimally across the channels and ranks for L3 processing. An mbuf contains a field indicating the pool that it originated from. When calling rte_ctrlmbuf_free(m) or rte_pktmbuf_free(m), the mbuf returns to its original pool.

6.3 Constructors

Packet and control mbuf constructors are provided by the API. The rte_pktmbuf_init() and rte_ctrlmbuf_init() functions initialize some fields in the mbuf structure that are not modified by the user once created (mbuf type, origin pool, buffer start address, and so on). This function is given as a callback function to the rte_mempool_create() function at pool creation time.

6.4 Allocating and Freeing mbufs

Allocating a new mbuf requires the user to specify the mempool from which the mbuf should be taken. For any newly-allocated mbuf, it contains one segment, with a length of 0. The offset to data is initialized to have some bytes of headroom in the buffer (RTE_PKTMBUF_HEADROOM).

Freeing a mbuf means returning it into its original mempool. The content of an mbuf is not modified when it is stored in a pool (as a free mbuf). Fields initialized by the constructor do not need to be re-initialized at mbuf allocation.

When freeing a packet mbuf that contains several segments, all of them are freed and returned to their original mempool.

6.5 Manipulating mbufs

This library provides some functions for manipulating the data in a packet mbuf. For instance:

- Get data length
- Get a pointer to the start of data
- Prepend data before data
- Append data after data
- Remove data at the beginning of the buffer (rte_pktmbuf_adj())
- Remove data at the end of the buffer (rte_pktmbuf_trim()) Refer to the *DPDK API Reference* for details.

6.6 Meta Information

Some information is retrieved by the network driver and stored in an mbuf to make processing easier. For instance, the VLAN, the RSS hash result (see *Poll Mode Driver*) and a flag indicating that the checksum was computed by hardware.

An mbuf also contains the input port (where it comes from), and the number of segment mbufs in the chain.

For chained buffers, only the first mbuf of the chain stores this meta information.

For instance, this is the case on RX side for the IEEE1588 packet timestamp mechanism, the VLAN tagging and the IP checksum computation.

On TX side, it is also possible for an application to delegate some processing to the hardware if it supports it. For instance, the PKT_TX_IP_CKSUM flag allows to offload the computation of the IPv4 checksum.

The following examples explain how to configure different TX offloads on a vxlan-encapsulated tcp packet: `out_eth/out_ip/out_udp/vxlan/in_eth/in_ip/in_tcp/payload`

- calculate checksum of out_ip:

```
mb->l2_len = len(out_eth)
mb->l3_len = len(out_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM
set out_ip checksum to 0 in the packet
```

  This is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM.

- calculate checksum of out_ip and out_udp:

```
 mb->l2_len = len(out_eth)
 mb->l3_len = len(out_ip)
 mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM | PKT_TX_UDP_CKSUM
 set out_ip checksum to 0 in the packet
 set out_udp checksum to pseudo header using rte_ipv4_phdr_cksum()

This is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM
and DEV_TX_OFFLOAD_UDP_CKSUM.
```

- calculate checksum of in_ip:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM
set in_ip checksum to 0 in the packet
```

  This is similar to case 1), but l2_len is different. It is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM. Note that it can only work if outer L4 checksum is 0.

- calculate checksum of in_ip and in_tcp:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM | PKT_TX_TCP_CKSUM
set in_ip checksum to 0 in the packet
set in_tcp checksum to pseudo header using rte_ipv4_phdr_cksum()
```

  This is similar to case 2), but l2_len is different. It is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM and DEV_TX_OFFLOAD_TCP_CKSUM. Note that it can only work if outer L4 checksum is 0.

- segment inner TCP:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->l4_len = len(in_tcp)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CKSUM | PKT_TX_TCP_CKSUM |
  PKT_TX_TCP_SEG;
```

```
set in_ip checksum to 0 in the packet
set in_tcp checksum to pseudo header without including the IP
  payload length using rte_ipv4_phdr_cksum()
```

This is supported on hardware advertising DEV_TX_OFFLOAD_TCP_TSO. Note that it can only work if outer L4 checksum is 0.

- calculate checksum of out_ip, in_ip, in_tcp:

```
mb->outer_l2_len = len(out_eth)
mb->outer_l3_len = len(out_ip)
mb->l2_len = len(out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags |= PKT_TX_OUTER_IPV4 | PKT_TX_OUTER_IP_CKSUM  | \
  PKT_TX_IP_CKSUM |  PKT_TX_TCP_CKSUM;
set out_ip checksum to 0 in the packet
set in_ip checksum to 0 in the packet
set in_tcp checksum to pseudo header using rte_ipv4_phdr_cksum()

This is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM,
DEV_TX_OFFLOAD_UDP_CKSUM and DEV_TX_OFFLOAD_OUTER_IPV4_CKSUM.
```

The list of flags and their precise meaning is described in the mbuf API documentation (rte_mbuf.h). Also refer to the testpmd source code (specifically the csumonly.c file) for details.

6.7 Direct and Indirect Buffers

A direct buffer is a buffer that is completely separate and self-contained. An indirect buffer behaves like a direct buffer but for the fact that the buffer pointer and data offset in it refer to data in another direct buffer. This is useful in situations where packets need to be duplicated or fragmented, since indirect buffers provide the means to reuse the same packet data across multiple buffers.

A buffer becomes indirect when it is "attached" to a direct buffer using the rte_pktmbuf_attach() function. Each buffer has a reference counter field and whenever an indirect buffer is attached to the direct buffer, the reference counter on the direct buffer is incremented. Similarly, whenever the indirect buffer is detached, the reference counter on the direct buffer is decremented. If the resulting reference counter is equal to 0, the direct buffer is freed since it is no longer in use.

There are a few things to remember when dealing with indirect buffers. First of all, it is not possible to attach an indirect buffer to another indirect buffer. Secondly, for a buffer to become indirect, its reference counter must be equal to 1, that is, it must not be already referenced by another indirect buffer. Finally, it is not possible to reattach an indirect buffer to the direct buffer (unless it is detached first).

While the attach/detach operations can be invoked directly using the recommended rte_pktmbuf_attach() and rte_pktmbuf_detach() functions, it is suggested to use the higher-level rte_pktmbuf_clone() function, which takes care of the correct initialization of an indirect buffer and can clone buffers with multiple segments.

Since indirect buffers are not supposed to actually hold any data, the memory pool for indirect buffers should be configured to indicate the reduced memory consumption. Examples of the initialization of a memory pool for indirect buffers (as well as use case examples for indirect buffers) can be found in several of the sample applications, for example, the IPv4 Multicast sample application.

6.8 Debug

In debug mode (CONFIG_RTE_MBUF_DEBUG is enabled), the functions of the mbuf library perform sanity checks before any operation (such as, buffer corruption, bad type, and so on).

6.9 Use Cases

All networking application should use mbufs to transport network packets.

CHAPTER

SEVEN

POLL MODE DRIVER

The DPDK includes 1 Gigabit, 10 Gigabit and 40 Gigabit and para virtualized virtio Poll Mode Drivers.

A Poll Mode Driver (PMD) consists of APIs, provided through the BSD driver running in user space, to configure the devices and their respective queues. In addition, a PMD accesses the RX and TX descriptors directly without any interrupts (with the exception of Link Status Change interrupts) to quickly receive, process and deliver packets in the user's application. This section describes the requirements of the PMDs, their global design principles and proposes a high-level architecture and a generic external API for the Ethernet PMDs.

7.1 Requirements and Assumptions

The DPDK environment for packet processing applications allows for two models, run-to-completion and pipe-line:

- In the *run-to-completion* model, a specific port's RX descriptor ring is polled for packets through an API. Packets are then processed on the same core and placed on a port's TX descriptor ring through an API for transmission.
- In the *pipe-line* model, one core polls one or more port's RX descriptor ring through an API. Packets are received and passed to another core via a ring. The other core continues to process the packet which then may be placed on a port's TX descriptor ring through an API for transmission.

In a synchronous run-to-completion model, each logical core assigned to the DPDK executes a packet processing loop that includes the following steps:

- Retrieve input packets through the PMD receive API
- Process each received packet one at a time, up to its forwarding
- Send pending output packets through the PMD transmit API

Conversely, in an asynchronous pipe-line model, some logical cores may be dedicated to the retrieval of received packets and other logical cores to the processing of previously received packets. Received packets are exchanged between logical cores through rings. The loop for packet retrieval includes the following steps:

- Retrieve input packets through the PMD receive API
- Provide received packets to processing lcores through packet queues

The loop for packet processing includes the following steps:

- Retrieve the received packet from the packet queue
- Process the received packet, up to its retransmission if forwarded

To avoid any unnecessary interrupt processing overhead, the execution environment must not use any asynchronous notification mechanisms. Whenever needed and appropriate, asynchronous communication should be introduced as much as possible through the use of rings.

Avoiding lock contention is a key issue in a multi-core environment. To address this issue, PMDs are designed to work with per-core private resources as much as possible. For example, a PMD maintains a separate transmit queue per-core, per-port. In the same way, every receive queue of a port is assigned to and polled by a single logical core (lcore).

To comply with Non-Uniform Memory Access (NUMA), memory management is designed to assign to each logical core a private buffer pool in local memory to minimize remote memory access. The configuration of packet buffer pools should take into account the underlying physical memory architecture in terms of DIMMS, channels and ranks. The application must ensure that appropriate parameters are given at memory pool creation time. See *Mempool Library*.

7.2 Design Principles

The API and architecture of the Ethernet* PMDs are designed with the following guidelines in mind.

PMDs must help global policy-oriented decisions to be enforced at the upper application level. Conversely, NIC PMD functions should not impede the benefits expected by upper-level global policies, or worse prevent such policies from being applied.

For instance, both the receive and transmit functions of a PMD have a maximum number of packets/descriptors to poll. This allows a run-to-completion processing stack to statically fix or to dynamically adapt its overall behavior through different global loop policies, such as:

- Receive, process immediately and transmit packets one at a time in a piecemeal fashion.
- Receive as many packets as possible, then process all received packets, transmitting them immediately.
- Receive a given maximum number of packets, process the received packets, accumulate them and finally send all accumulated packets to transmit.

To achieve optimal performance, overall software design choices and pure software optimization techniques must be considered and balanced against available low-level hardware-based optimization features (CPU cache properties, bus speed, NIC PCI bandwidth, and so on). The case of packet transmission is an example of this software/hardware tradeoff issue when optimizing burst-oriented network packet processing engines. In the initial case, the PMD could export only an rte_eth_tx_one function to transmit one packet at a time on a given queue. On top of that, one can easily build an rte_eth_tx_burst function that loops invoking the rte_eth_tx_one function to transmit several packets at a time. However, an rte_eth_tx_burst function is effectively implemented by the PMD to minimize the driver-level transmit cost per packet through the following optimizations:

- Share among multiple packets the un-amortized cost of invoking the rte_eth_tx_one function.
- Enable the rte_eth_tx_burst function to take advantage of burst-oriented hardware features (prefetch data in cache, use of NIC head/tail registers) to minimize the number of

CPU cycles per packet, for example by avoiding unnecessary read memory accesses to ring transmit descriptors, or by systematically using arrays of pointers that exactly fit cache line boundaries and sizes.

- Apply burst-oriented software optimization techniques to remove operations that would otherwise be unavoidable, such as ring index wrap back management.

Burst-oriented functions are also introduced via the API for services that are intensively used by the PMD. This applies in particular to buffer allocators used to populate NIC rings, which provide functions to allocate/free several buffers at a time. For example, an mbuf_multiple_alloc function returning an array of pointers to rte_mbuf buffers which speeds up the receive poll function of the PMD when replenishing multiple descriptors of the receive ring.

7.3 Logical Cores, Memory and NIC Queues Relationships

The DPDK supports NUMA allowing for better performance when a processor's logical cores and interfaces utilize its local memory. Therefore, mbuf allocation associated with local PCIe* interfaces should be allocated from memory pools created in the local memory. The buffers should, if possible, remain on the local processor to obtain the best performance results and RX and TX buffer descriptors should be populated with mbufs allocated from a mempool allocated from local memory.

The run-to-completion model also performs better if packet or data manipulation is in local memory instead of a remote processors memory. This is also true for the pipe-line model provided all logical cores used are located on the same processor.

Multiple logical cores should never share receive or transmit queues for interfaces since this would require global locks and hinder performance.

7.4 Device Identification and Configuration

7.4.1 Device Identification

Each NIC port is uniquely designated by its (bus/bridge, device, function) PCI identifiers assigned by the PCI probing/enumeration function executed at DPDK initialization. Based on their PCI identifier, NIC ports are assigned two other identifiers:

- A port index used to designate the NIC port in all functions exported by the PMD API.
- A port name used to designate the port in console messages, for administration or debugging purposes. For ease of use, the port name includes the port index.

7.4.2 Device Configuration

The configuration of each NIC port includes the following operations:

- Allocate PCI resources
- Reset the hardware (issue a Global Reset) to a well-known default state
- Set up the PHY and the link
- Initialize statistics counters

The PMD API must also export functions to start/stop the all-multicast feature of a port and functions to set/unset the port in promiscuous mode.

Some hardware offload features must be individually configured at port initialization through specific configuration parameters. This is the case for the Receive Side Scaling (RSS) and Data Center Bridging (DCB) features for example.

7.4.3 On-the-Fly Configuration

All device features that can be started or stopped "on the fly" (that is, without stopping the device) do not require the PMD API to export dedicated functions for this purpose.

All that is required is the mapping address of the device PCI registers to implement the configuration of these features in specific functions outside of the drivers.

For this purpose, the PMD API exports a function that provides all the information associated with a device that can be used to set up a given device feature outside of the driver. This includes the PCI vendor identifier, the PCI device identifier, the mapping address of the PCI device registers, and the name of the driver.

The main advantage of this approach is that it gives complete freedom on the choice of the API used to configure, to start, and to stop such features.

As an example, refer to the configuration of the IEEE1588 feature for the Intel® 82576 Gigabit Ethernet Controller and the Intel® 82599 10 Gigabit Ethernet Controller controllers in the testpmd application.

Other features such as the L3/L4 5-Tuple packet filtering feature of a port can be configured in the same way. Ethernet* flow control (pause frame) can be configured on the individual port. Refer to the testpmd source code for details. Also, L4 (UDP/TCP/ SCTP) checksum offload by the NIC can be enabled for an individual packet as long as the packet mbuf is set up correctly. See *Hardware Offload* for details.

7.4.4 Configuration of Transmit and Receive Queues

Each transmit queue is independently configured with the following information:

- The number of descriptors of the transmit ring
- The socket identifier used to identify the appropriate DMA memory zone from which to allocate the transmit ring in NUMA architectures
- The values of the Prefetch, Host and Write-Back threshold registers of the transmit queue
- The *minimum* transmit packets to free threshold (tx_free_thresh). When the number of descriptors used to transmit packets exceeds this threshold, the network adaptor should be checked to see if it has written back descriptors. A value of 0 can be passed during the TX queue configuration to indicate the default value should be used. The default value for tx_free_thresh is 32. This ensures that the PMD does not search for completed descriptors until at least 32 have been processed by the NIC for this queue.
- The *minimum* RS bit threshold. The minimum number of transmit descriptors to use before setting the Report Status (RS) bit in the transmit descriptor. Note that this parameter may only be valid for Intel 10 GbE network adapters. The RS bit is set on the last descriptor used to transmit a packet if the number of descriptors used since the last RS bit

setting, up to the first descriptor used to transmit the packet, exceeds the transmit RS bit threshold (tx_rs_thresh). In short, this parameter controls which transmit descriptors are written back to host memory by the network adapter. A value of 0 can be passed during the TX queue configuration to indicate that the default value should be used. The default value for tx_rs_thresh is 32. This ensures that at least 32 descriptors are used before the network adapter writes back the most recently used descriptor. This saves upstream PCIe* bandwidth resulting from TX descriptor write-backs. It is important to note that the TX Write-back threshold (TX wthresh) should be set to 0 when tx_rs_thresh is greater than 1. Refer to the Intel® 82599 10 Gigabit Ethernet Controller Datasheet for more details.

The following constraints must be satisfied for tx_free_thresh and tx_rs_thresh:

- tx_rs_thresh must be greater than 0.
- tx_rs_thresh must be less than the size of the ring minus 2.
- tx_rs_thresh must be less than or equal to tx_free_thresh.
- tx_free_thresh must be greater than 0.
- tx_free_thresh must be less than the size of the ring minus 3.
- For optimal performance, TX wthresh should be set to 0 when tx_rs_thresh is greater than 1.

One descriptor in the TX ring is used as a sentinel to avoid a hardware race condition, hence the maximum threshold constraints.

Note: When configuring for DCB operation, at port initialization, both the number of transmit queues and the number of receive queues must be set to 128.

7.4.5 Hardware Offload

Depending on driver capabilities advertised by `rte_eth_dev_info_get()`, the PMD may support hardware offloading feature like checksumming, TCP segmentation or VLAN insertion.

The support of these offload features implies the addition of dedicated status bit(s) and value field(s) into the rte_mbuf data structure, along with their appropriate handling by the receive/transmit functions exported by each PMD. The list of flags and their precise meaning is described in the mbuf API documentation and in the in *Mbuf Library*, section "Meta Information".

7.5 Poll Mode Driver API

7.5.1 Generalities

By default, all functions exported by a PMD are lock-free functions that are assumed not to be invoked in parallel on different logical cores to work on the same target object. For instance, a PMD receive function cannot be invoked in parallel on two logical cores to poll the same RX queue of the same port. Of course, this function can be invoked in parallel by different logical cores on different RX queues. It is the responsibility of the upper-level application to enforce this rule.

If needed, parallel accesses by multiple logical cores to shared queues can be explicitly protected by dedicated inline lock-aware functions built on top of their corresponding lock-free functions of the PMD API.

7.5.2 Generic Packet Representation

A packet is represented by an rte_mbuf structure, which is a generic metadata structure containing all necessary housekeeping information. This includes fields and status bits corresponding to offload hardware features, such as checksum computation of IP headers or VLAN tags.

The rte_mbuf data structure includes specific fields to represent, in a generic way, the offload features provided by network controllers. For an input packet, most fields of the rte_mbuf structure are filled in by the PMD receive function with the information contained in the receive descriptor. Conversely, for output packets, most fields of rte_mbuf structures are used by the PMD transmit function to initialize transmit descriptors.

The mbuf structure is fully described in the *Mbuf Library* chapter.

7.5.3 Ethernet Device API

The Ethernet device API exported by the Ethernet PMDs is described in the *DPDK API Reference*.

7.5.4 Extended Statistics API

The extended statistics API allows each individual PMD to expose a unique set of statistics. Accessing these from application programs is done via two functions:

- `rte_eth_xstats_get`: Fills in an array of `struct rte_eth_xstat` with extended statistics.
- `rte_eth_xstats_get_names`: Fills in an array of `struct rte_eth_xstat_name` with extended statistic name lookup information.

Each `struct rte_eth_xstat` contains an identifier and value pair, and each `struct rte_eth_xstat_name` contains a string. Each identifier within the `struct rte_eth_xstat` lookup array must have a corresponding entry in the `struct rte_eth_xstat_name` lookup array. Within the latter the index of the entry is the identifier the string is associated with. These identifiers, as well as the number of extended statistic exposed, must remain constant during runtime. Note that extended statistic identifiers are driver-specific, and hence might not be the same for different ports.

A naming scheme exists for the strings exposed to clients of the API. This is to allow scraping of the API for statistics of interest. The naming scheme uses strings split by a single underscore _. The scheme is as follows:

- direction
- detail 1
- detail 2
- detail n

- unit

Examples of common statistics xstats strings, formatted to comply to the scheme proposed above:

- `rx_bytes`
- `rx_crc_errors`
- `tx_multicast_packets`

The scheme, although quite simple, allows flexibility in presenting and reading information from the statistic strings. The following example illustrates the naming scheme:`rx_packets`. In this example, the string is split into two components. The first component `rx` indicates that the statistic is associated with the receive side of the NIC. The second component `packets` indicates that the unit of measure is packets.

A more complicated example: `tx_size_128_to_255_packets`. In this example, `tx` indicates transmission, `size` is the first detail, `128` etc are more details, and `packets` indicates that this is a packet counter.

Some additions in the metadata scheme are as follows:

- If the first part does not match `rx` or `tx`, the statistic does not have an affinity with either receive of transmit.
- If the first letter of the second part is `q` and this `q` is followed by a number, this statistic is part of a specific queue.

An example where queue numbers are used is as follows: `tx_q7_bytes` which indicates this statistic applies to queue number 7, and represents the number of transmitted bytes on that queue.

CHAPTER

EIGHT

CRYPTOGRAPHY DEVICE LIBRARY

The cryptodev library provides a Crypto device framework for management and provisioning of hardware and software Crypto poll mode drivers, defining generic APIs which support a number of different Crypto operations. The framework currently only supports cipher, authentication, chained cipher/authentication and AEAD symmetric Crypto operations.

8.1 Design Principles

The cryptodev library follows the same basic principles as those used in DPDKs Ethernet Device framework. The Crypto framework provides a generic Crypto device framework which supports both physical (hardware) and virtual (software) Crypto devices as well as a generic Crypto API which allows Crypto devices to be managed and configured and supports Crypto operations to be provisioned on Crypto poll mode driver.

8.2 Device Management

8.2.1 Device Creation

Physical Crypto devices are discovered during the PCI probe/enumeration of the EAL function which is executed at DPDK initialization, based on their PCI device identifier, each unique PCI BDF (bus/bridge, device, function). Specific physical Crypto devices, like other physical devices in DPDK can be white-listed or black-listed using the EAL command line options.

Virtual devices can be created by two mechanisms, either using the EAL command line options or from within the application using an EAL API directly.

From the command line using the –vdev EAL option

```
--vdev  'cryptodev_aesni_mb_pmd0,max_nb_queue_pairs=2,max_nb_sessions=1024,socket_id=0'
```

Our using the rte_eal_vdev_init API within the application code.

```
rte_eal_vdev_init("cryptodev_aesni_mb_pmd",
                  "max_nb_queue_pairs=2,max_nb_sessions=1024,socket_id=0")
```

All virtual Crypto devices support the following initialization parameters:

- `max_nb_queue_pairs` - maximum number of queue pairs supported by the device.
- `max_nb_sessions` - maximum number of sessions supported by the device
- `socket_id` - socket on which to allocate the device resources on.

8.2.2 Device Identification

Each device, whether virtual or physical is uniquely designated by two identifiers:

- A unique device index used to designate the Crypto device in all functions exported by the cryptodev API.
- A device name used to designate the Crypto device in console messages, for administration or debugging purposes. For ease of use, the port name includes the port index.

8.2.3 Device Configuration

The configuration of each Crypto device includes the following operations:

- Allocation of resources, including hardware resources if a physical device.
- Resetting the device into a well-known default state.
- Initialization of statistics counters.

The rte_cryptodev_configure API is used to configure a Crypto device.

```
int rte_cryptodev_configure(uint8_t dev_id,
                            struct rte_cryptodev_config *config)
```

The `rte_cryptodev_config` structure is used to pass the configuration parameters. In contains parameter for socket selection, number of queue pairs and the session mempool configuration.

```
struct rte_cryptodev_config {
    int socket_id;
    /**< Socket to allocate resources on */
    uint16_t nb_queue_pairs;
    /**< Number of queue pairs to configure on device */

    struct {
        uint32_t nb_objs;
        uint32_t cache_size;
    } session_mp;
    /**< Session mempool configuration */
};
```

8.2.4 Configuration of Queue Pairs

Each Crypto devices queue pair is individually configured through the `rte_cryptodev_queue_pair_setup` API. Each queue pairs resources may be allocated on a specified socket.

```
int rte_cryptodev_queue_pair_setup(uint8_t dev_id, uint16_t queue_pair_id,
            const struct rte_cryptodev_qp_conf *qp_conf,
            int socket_id)

struct rte_cryptodev_qp_conf {
    uint32_t nb_descriptors; /**< Number of descriptors per queue pair */
};
```

8.2.5 Logical Cores, Memory and Queues Pair Relationships

The Crypto device Library as the Poll Mode Driver library support NUMA for when a processor's logical cores and interfaces utilize its local memory. Therefore Crypto operations, and in the case of symmetric Crypto operations, the session and the mbuf being operated on, should be allocated from memory pools created in the local memory. The buffers should, if possible, remain on the local processor to obtain the best performance results and buffer descriptors should be populated with mbufs allocated from a mempool allocated from local memory.

The run-to-completion model also performs better, especially in the case of virtual Crypto devices, if the Crypto operation and session and data buffer is in local memory instead of a remote processor's memory. This is also true for the pipe-line model provided all logical cores used are located on the same processor.

Multiple logical cores should never share the same queue pair for enqueuing operations or dequeuing operations on the same Crypto device since this would require global locks and hinder performance. It is however possible to use a different logical core to dequeue an operation on a queue pair from the logical core which it was enqueued on. This means that a crypto burst enqueue/dequeue APIs are a logical place to transition from one logical core to another in a packet processing pipeline.

8.3 Device Features and Capabilities

Crypto devices define their functionality through two mechanisms, global device features and algorithm capabilities. Global devices features identify device wide level features which are applicable to the whole device such as the device having hardware acceleration or supporting symmetric Crypto operations,

The capabilities mechanism defines the individual algorithms/functions which the device supports, such as a specific symmetric Crypto cipher or authentication operation.

8.3.1 Device Features

Currently the following Crypto device features are defined:

- Symmetric Crypto operations
- Asymmetric Crypto operations
- Chaining of symmetric Crypto operations
- SSE accelerated SIMD vector operations
- AVX accelerated SIMD vector operations
- AVX2 accelerated SIMD vector operations
- AESNI accelerated instructions
- Hardware off-load processing

8.3.2 Device Operation Capabilities

Crypto capabilities which identify particular algorithm which the Crypto PMD supports are defined by the operation type, the operation transform, the transform identifier and then the particulars of the transform. For the full scope of the Crypto capability see the definition of the structure in the *DPDK API Reference*.

```
struct rte_cryptodev_capabilities;
```

Each Crypto poll mode driver defines its own private array of capabilities for the operations it supports. Below is an example of the capabilities for a PMD which supports the authentication algorithm SHA1_HMAC and the cipher algorithm AES_CBC.

```
static const struct rte_cryptodev_capabilities pmd_capabilities[] = {
    {    /* SHA1 HMAC */
        .op = RTE_CRYPTO_OP_TYPE_SYMMETRIC,
        .sym = {
            .xform_type = RTE_CRYPTO_SYM_XFORM_AUTH,
            .auth = {
                .algo = RTE_CRYPTO_AUTH_SHA1_HMAC,
                .block_size = 64,
                .key_size = {
                    .min = 64,
                    .max = 64,
                    .increment = 0
                },
                .digest_size = {
                    .min = 12,
                    .max = 12,
                    .increment = 0
                },
                .aad_size = { 0 }
            }
        }
    },
    {    /* AES CBC */
        .op = RTE_CRYPTO_OP_TYPE_SYMMETRIC,
        .sym = {
            .xform_type = RTE_CRYPTO_SYM_XFORM_CIPHER,
            .cipher = {
                .algo = RTE_CRYPTO_CIPHER_AES_CBC,
                .block_size = 16,
                .key_size = {
                    .min = 16,
                    .max = 32,
                    .increment = 8
                },
                .iv_size = {
                    .min = 16,
                    .max = 16,
                    .increment = 0
                }
            }
        }
    }
}
```

8.3.3 Capabilities Discovery

Discovering the features and capabilities of a Crypto device poll mode driver is achieved through the `rte_cryptodev_info_get` function.

```
void rte_cryptodev_info_get(uint8_t dev_id,
                            struct rte_cryptodev_info *dev_info);
```

This allows the user to query a specific Crypto PMD and get all the device features and capabilities. The `rte_cryptodev_info` structure contains all the relevant information for the device.

```
struct rte_cryptodev_info {
    const char *driver_name;
    enum rte_cryptodev_type dev_type;
    struct rte_pci_device *pci_dev;

    uint64_t feature_flags;

    const struct rte_cryptodev_capabilities *capabilities;

    unsigned max_nb_queue_pairs;

    struct {
        unsigned max_nb_sessions;
    } sym;
};
```

8.4 Operation Processing

Scheduling of Crypto operations on DPDK's application data path is performed using a burst oriented asynchronous API set. A queue pair on a Crypto device accepts a burst of Crypto operations using enqueue burst API. On physical Crypto devices the enqueue burst API will place the operations to be processed on the devices hardware input queue, for virtual devices the processing of the Crypto operations is usually completed during the enqueue call to the Crypto device. The dequeue burst API will retrieve any processed operations available from the queue pair on the Crypto device, from physical devices this is usually directly from the devices processed queue, and for virtual device's from a `rte_ring` where processed operations are place after being processed on the enqueue call.

8.4.1 Enqueue / Dequeue Burst APIs

The burst enqueue API uses a Crypto device identifier and a queue pair identifier to specify the Crypto device queue pair to schedule the processing on. The `nb_ops` parameter is the number of operations to process which are supplied in the `ops` array of `rte_crypto_op` structures. The enqueue function returns the number of operations it actually enqueued for processing, a return value equal to `nb_ops` means that all packets have been enqueued.

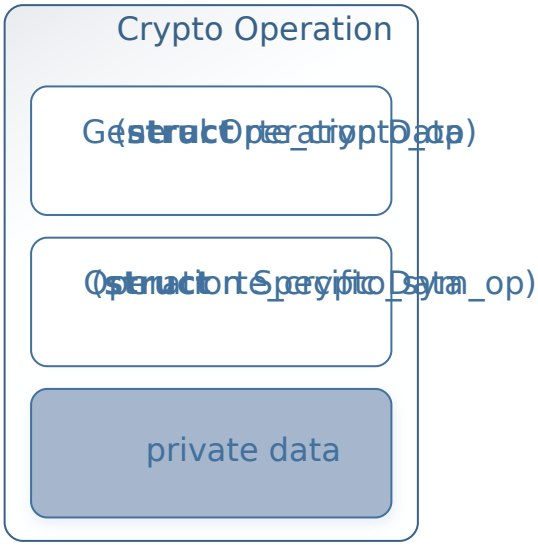
```
uint16_t rte_cryptodev_enqueue_burst(uint8_t dev_id, uint16_t qp_id,
                                     struct rte_crypto_op **ops, uint16_t nb_ops)
```

The dequeue API uses the same format as the enqueue API of processed but the `nb_ops` and `ops` parameters are now used to specify the max processed operations the user wishes to retrieve and the location in which to store them. The API call returns the actual number of processed operations returned, this can never be larger than `nb_ops`.

```
uint16_t rte_cryptodev_dequeue_burst(uint8_t dev_id, uint16_t qp_id,
                                     struct rte_crypto_op **ops, uint16_t nb_ops)
```

8.4.2 Operation Representation

An Crypto operation is represented by an rte_crypto_op structure, which is a generic metadata container for all necessary information required for the Crypto operation to be processed on a particular Crypto device poll mode driver.



The operation structure includes the operation type and the operation status, a reference to the operation specific data, which can vary in size and content depending on the operation being provisioned. It also contains the source mempool for the operation, if it allocate from a mempool. Finally an opaque pointer for user specific data is provided.

If Crypto operations are allocated from a Crypto operation mempool, see next section, there is also the ability to allocate private memory with the operation for applications purposes.

Application software is responsible for specifying all the operation specific fields in the `rte_crypto_op` structure which are then used by the Crypto PMD to process the requested operation.

8.4.3 Operation Management and Allocation

The cryptodev library provides an API set for managing Crypto operations which utilize the Mempool Library to allocate operation buffers. Therefore, it ensures that the crytpo operation is interleaved optimally across the channels and ranks for optimal processing. A `rte_crypto_op` contains a field indicating the pool that it originated from. When calling `rte_crypto_op_free(op)`, the operation returns to its original pool.

```
extern struct rte_mempool *
rte_crypto_op_pool_create(const char *name, enum rte_crypto_op_type type,
                          unsigned nb_elts, unsigned cache_size, uint16_t priv_size,
                          int socket_id);
```

During pool creation `rte_crypto_op_init()` is called as a constructor to initialize each Crypto operation which subsequently calls `__rte_crypto_op_reset()` to configure any operation type specific fields based on the type parameter.

`rte_crypto_op_alloc()` and `rte_crypto_op_bulk_alloc()` are used to allocate Crypto operations of a specific type from a given Crypto operation mempool. `__rte_crypto_op_reset()` is called on each operation before being returned to allocate to a user so the operation is always in a good known state before use by the application.

```
struct rte_crypto_op *rte_crypto_op_alloc(struct rte_mempool *mempool,
                                          enum rte_crypto_op_type type)

unsigned rte_crypto_op_bulk_alloc(struct rte_mempool *mempool,
                                  enum rte_crypto_op_type type,
                                  struct rte_crypto_op **ops, uint16_t nb_ops)
```

`rte_crypto_op_free()` is called by the application to return an operation to its allocating pool.

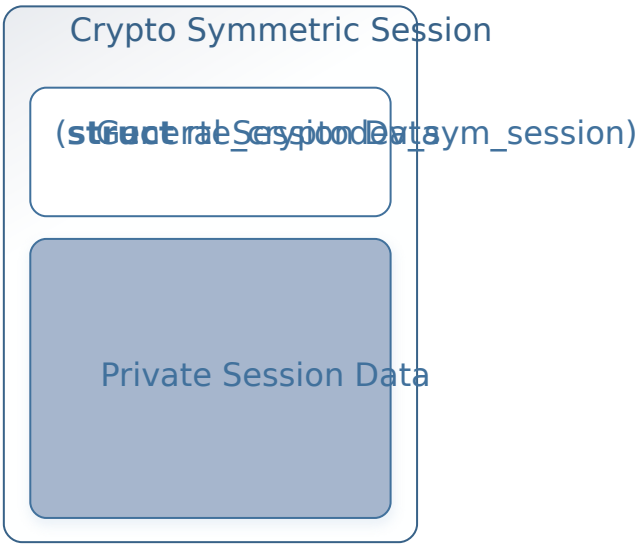
```
void rte_crypto_op_free(struct rte_crypto_op *op)
```

8.5 Symmetric Cryptography Support

The cryptodev library currently provides support for the following symmetric Crypto operations; cipher, authentication, including chaining of these operations, as well as also supporting AEAD operations.

8.5.1 Session and Session Management

Session are used in symmetric cryptographic processing to store the immutable data defined in a cryptographic transform which is used in the operation processing of a packet flow. Sessions are used to manage information such as expand cipher keys and HMAC IPADs and OPADs, which need to be calculated for a particular Crypto operation, but are immutable on a packet to packet basis for a flow. Crypto sessions cache this immutable data in a optimal way for the underlying PMD and this allows further acceleration of the offload of Crypto workloads.



The Crypto device framework provides a set of session pool management APIs for the creation and freeing of the sessions, utilizing the Mempool Library.

The framework also provides hooks so the PMDs can pass the amount of memory required for that PMDs private session parameters, as well as initialization functions for the configuration of the session parameters and freeing function so the PMD can managed the memory on destruction of a session.

Note: Sessions created on a particular device can only be used on Crypto devices of the same type, and if you try to use a session on a device different to that on which it was created then the Crypto operation will fail.

`rte_cryptodev_sym_session_create()` is used to create a symmetric session on Crypto device. A symmetric transform chain is used to specify the particular operation and its parameters. See the section below for details on transforms.

```
struct rte_cryptodev_sym_session * rte_cryptodev_sym_session_create(
        uint8_t dev_id, struct rte_crypto_sym_xform *xform);
```

Note: For AEAD operations the algorithm selected for authentication and ciphering must aligned, eg AES_GCM.

8.5.2 Transforms and Transform Chaining

Symmetric Crypto transforms (`rte_crypto_sym_xform`) are the mechanism used to specify the details of the Crypto operation. For chaining of symmetric operations such as cipher encrypt and authentication generate, the next pointer allows transform to be chained together. Crypto devices which support chaining must publish the chaining of symmetric Crypto operations feature flag.

Currently there are two transforms types cipher and authentication, to specify an AEAD operation it is required to chain a cipher and an authentication transform together. Also it is important to note that the order in which the transforms are passed indicates the order of the chaining.

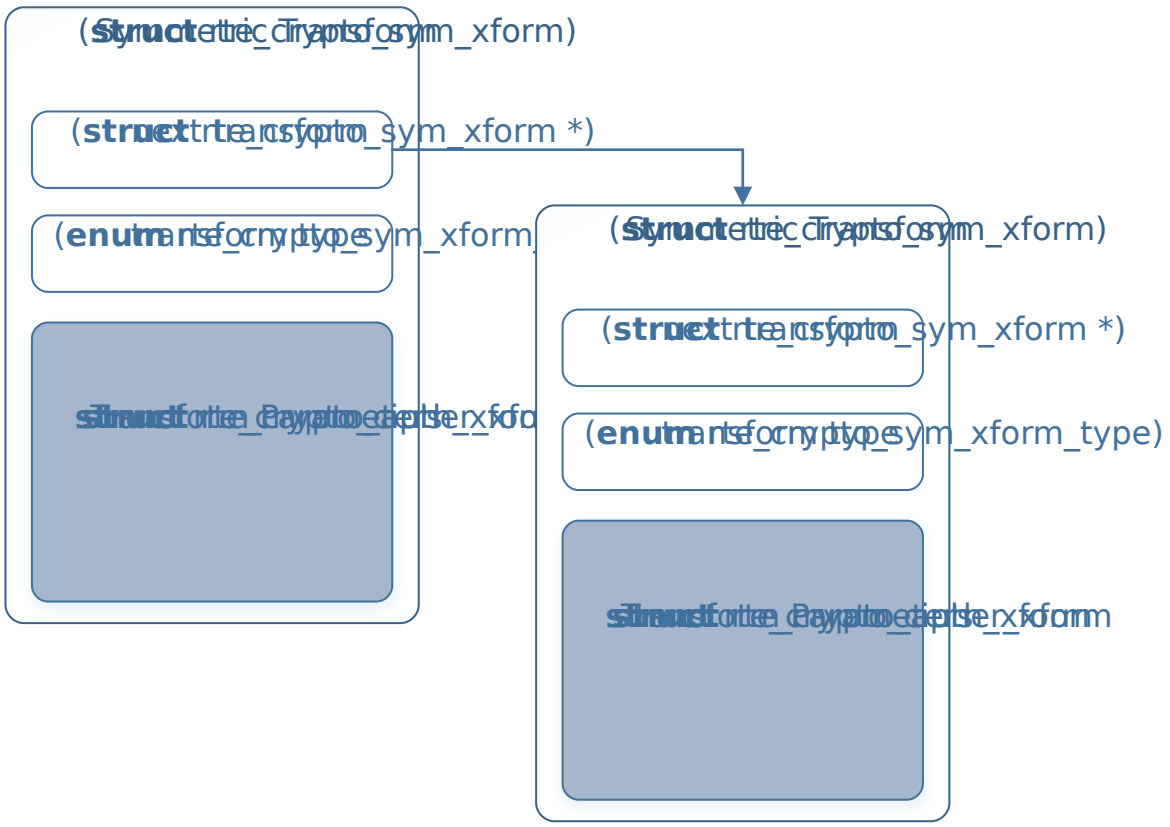
```
struct rte_crypto_sym_xform {
    struct rte_crypto_sym_xform *next;
    /**< next xform in chain */
    enum rte_crypto_sym_xform_type type;
    /**< xform type */
    union {
        struct rte_crypto_auth_xform auth;
        /**< Authentication / hash xform */
        struct rte_crypto_cipher_xform cipher;
        /**< Cipher xform */
    };
};
```

The API does not place a limit on the number of transforms that can be chained together but this will be limited by the underlying Crypto device poll mode driver which is processing the operation.

8.5.3 Symmetric Operations

The symmetric Crypto operation structure contains all the mutable data relating to performing symmetric cryptographic processing on a referenced mbuf data buffer. It is used for either cipher, authentication, AEAD and chained operations.

As a minimum the symmetric operation must have a source data buffer (`m_src`), the session type (session-based/less), a valid session (or transform chain if in session-less mode) and



the minimum authentication/ cipher parameters required depending on the type of operation specified in the session or the transform chain.

```
struct rte_crypto_sym_op {
    struct rte_mbuf *m_src;
    struct rte_mbuf *m_dst;

    enum rte_crypto_sym_op_sess_type type;

    union {
        struct rte_cryptodev_sym_session *session;
        /**< Handle for the initialised session context */
        struct rte_crypto_sym_xform *xform;
        /**< Session-less API Crypto operation parameters */
    };

    struct {
        struct {
            uint32_t offset;
            uint32_t length;
        } data;    /**< Data offsets and length for ciphering */

        struct {
            uint8_t *data;
            phys_addr_t phys_addr;
            uint16_t length;
        } iv;      /**< Initialisation vector parameters */
    } cipher;

    struct {
        struct {
            uint32_t offset;
            uint32_t length;
```

```
        } data;    /**< Data offsets and length for authentication */

        struct {
            uint8_t *data;
            phys_addr_t phys_addr;
            uint16_t length;
        } digest; /**< Digest parameters */

        struct {
            uint8_t *data;
            phys_addr_t phys_addr;
            uint16_t length;
        } aad;     /**< Additional authentication parameters */
    } auth;
}
```

8.6 Asymmetric Cryptography

Asymmetric functionality is currently not supported by the cryptodev API.

8.6.1 Crypto Device API

The cryptodev Library API is described in the *DPDK API Reference* document.

CHAPTER

NINE

LINK BONDING POLL MODE DRIVER LIBRARY

In addition to Poll Mode Drivers (PMDs) for physical and virtual hardware, DPDK also includes a pure-software library that allows physical PMD's to be bonded together to create a single logical PMD.

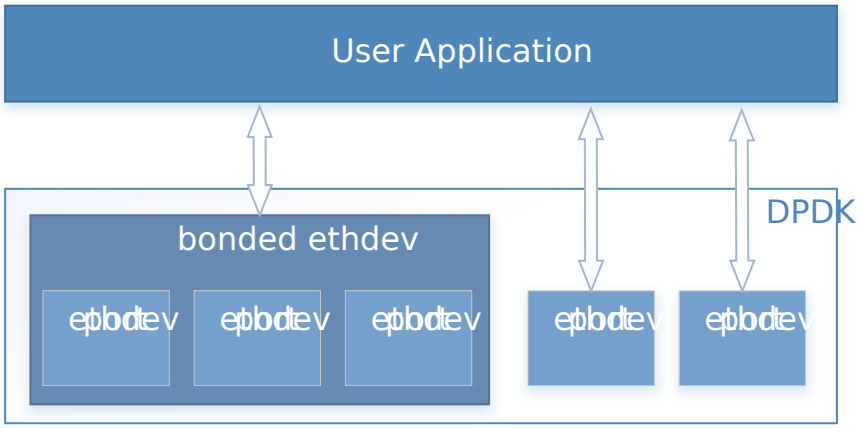


Fig. 9.1: Bonded PMDs

The Link Bonding PMD library(librte_pmd_bond) supports bonding of groups of `rte_eth_dev` ports of the same speed and duplex to provide similar the capabilities to that found in Linux bonding driver to allow the aggregation of multiple (slave) NICs into a single logical interface between a server and a switch. The new bonded PMD will then process these interfaces based on the mode of operation specified to provide support for features such as redundant links, fault tolerance and/or load balancing.

The librte_pmd_bond library exports a C API which provides an API for the creation of bonded devices as well as the configuration and management of the bonded device and its slave devices.

Note: The Link Bonding PMD Library is enabled by default in the build configuration files, the library can be disabled by setting `CONFIG_RTE_LIBRTE_PMD_BOND=n` and recompiling the DPDK.

9.1 Link Bonding Modes Overview

Currently the Link Bonding PMD library supports following modes of operation:

- **Round-Robin (Mode 0):**

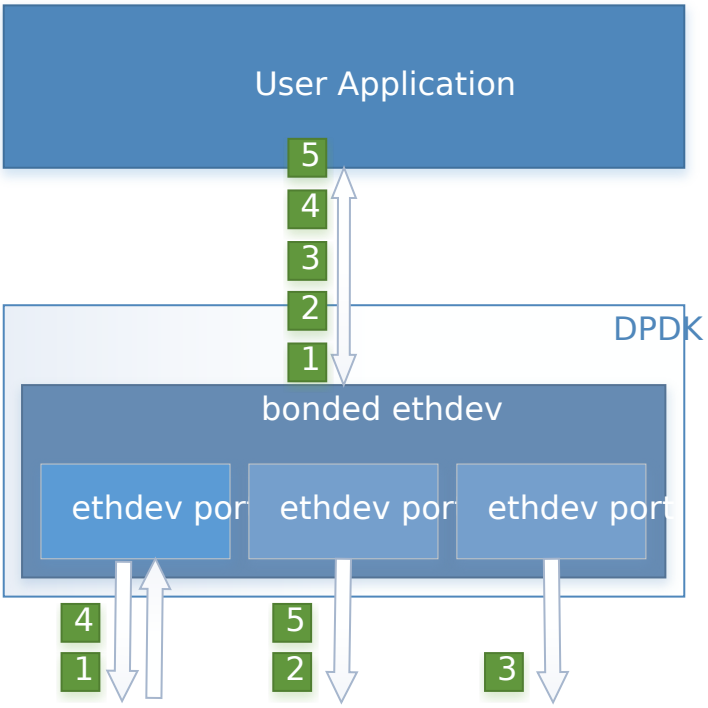


Fig. 9.2: Round-Robin (Mode 0)

This mode provides load balancing and fault tolerance by transmission of packets in sequential order from the first available slave device through the last. Packets are bulk dequeued from devices then serviced in a round-robin manner. This mode does not guarantee in order reception of packets and down stream should be able to handle out of order packets.

- **Active Backup (Mode 1):**
- **Balance XOR (Mode 2):**

Note: The coloring differences of the packets are used to identify different flow classification calculated by the selected transmit policy

---

- **Broadcast (Mode 3):**
- **Link Aggregation 802.3AD (Mode 4):**
- **Transmit Load Balancing (Mode 5):**

9.2 Implementation Details

The librte_pmd_bond bonded device are compatible with the Ethernet device API exported by the Ethernet PMDs described in the *DPDK API Reference*.

The Link Bonding Library supports the creation of bonded devices at application startup time during EAL initialization using the `--vdev` option as well as programmatically via the C API `rte_eth_bond_create` function.

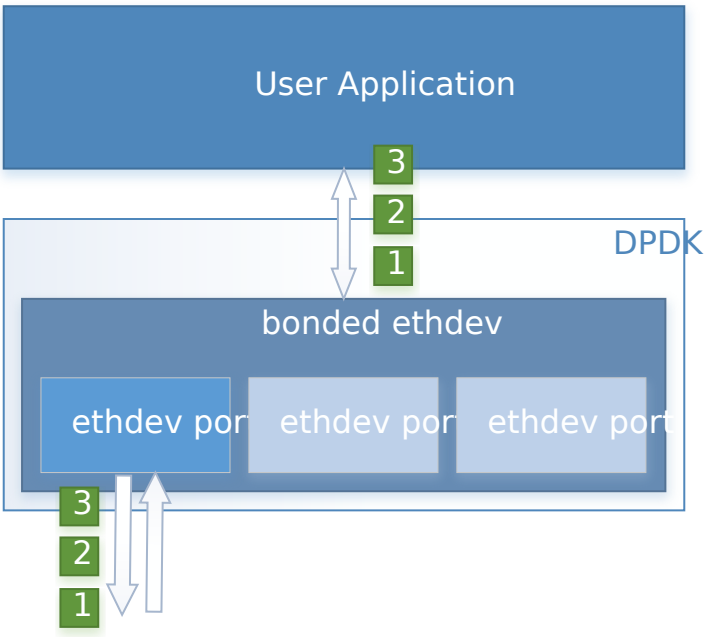


Fig. 9.3: Active Backup (Mode 1)

In this mode only one slave in the bond is active at any time, a different slave becomes active if, and only if, the primary active slave fails, thereby providing fault tolerance to slave failure. The single logical bonded interface's MAC address is externally visible on only one NIC (port) to avoid confusing the network switch.

Bonded devices support the dynamical addition and removal of slave devices using the `rte_eth_bond_slave_add` / `rte_eth_bond_slave_remove` APIs.

After a slave device is added to a bonded device slave is stopped using `rte_eth_dev_stop` and then reconfigured using `rte_eth_dev_configure` the RX and TX queues are also reconfigured using `rte_eth_tx_queue_setup` / `rte_eth_rx_queue_setup` with the parameters use to configure the bonding device. If RSS is enabled for bonding device, this mode is also enabled on new slave and configured as well.

Setting up multi-queue mode for bonding device to RSS, makes it fully RSS-capable, so all slaves are synchronized with its configuration. This mode is intended to provide RSS configuration on slaves transparent for client application implementation.

Bonding device stores its own version of RSS settings i.e. RETA, RSS hash function and RSS key, used to set up its slaves. That let to define the meaning of RSS configuration of bonding device as desired configuration of whole bonding (as one unit), without pointing any of slave inside. It is required to ensure consistency and made it more error-proof.

RSS hash function set for bonding device, is a maximal set of RSS hash functions supported by all bonded slaves. RETA size is a GCD of all its RETA's sizes, so it can be easily used as a pattern providing expected behavior, even if slave RETAs' sizes are different. If RSS Key is not set for bonded device, it's not changed on the slaves and default key for device is used.

All settings are managed through the bonding port API and always are propagated in one direction (from bonding to slaves).

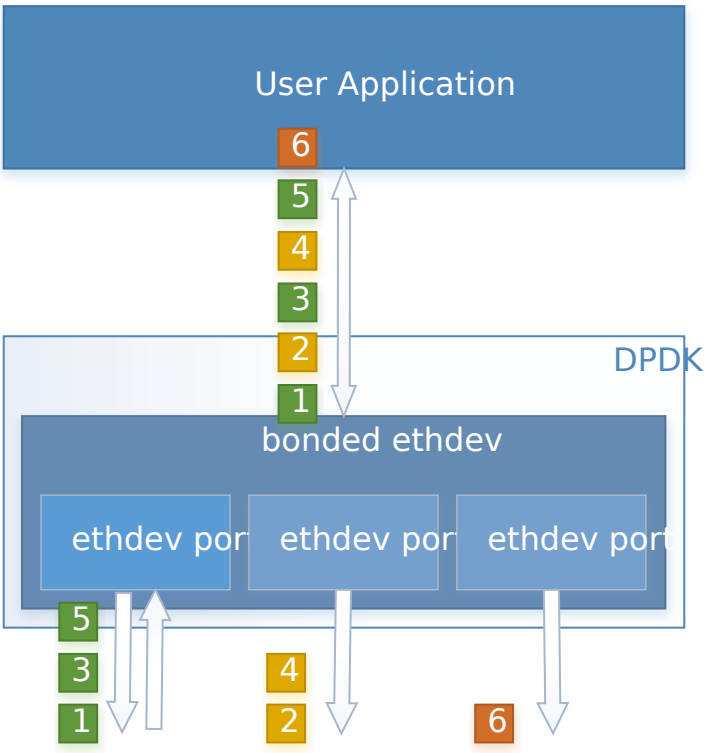


Fig. 9.4: Balance XOR (Mode 2)

This mode provides transmit load balancing (based on the selected transmission policy) and fault tolerance. The default policy (layer2) uses a simple calculation based on the packet flow source and destination MAC addresses as well as the number of active slaves available to the bonded device to classify the packet to a specific slave to transmit on. Alternate transmission policies supported are layer 2+3, this takes the IP source and destination addresses into the calculation of the transmit slave port and the final supported policy is layer 3+4, this uses IP source and destination addresses as well as the TCP/UDP source and destination port.

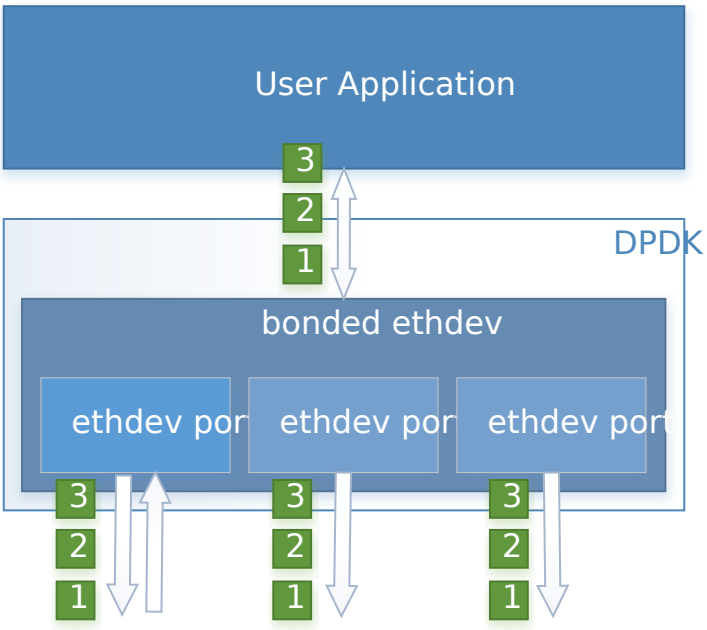


Fig. 9.5: Broadcast (Mode 3)

This mode provides fault tolerance by transmission of packets on all slave ports.

9.2.1 Link Status Change Interrupts / Polling

Link bonding devices support the registration of a link status change callback, using the `rte_eth_dev_callback_register` API, this will be called when the status of the bonding device changes. For example in the case of a bonding device which has 3 slaves, the link status will change to up when one slave becomes active or change to down when all slaves become inactive. There is no callback notification when a single slave changes state and the previous conditions are not met. If a user wishes to monitor individual slaves then they must register callbacks with that slave directly.

The link bonding library also supports devices which do not implement link status change interrupts, this is achieved by polling the devices link status at a defined period which is set using the `rte_eth_bond_link_monitoring_set` API, the default polling interval is 10ms. When a device is added as a slave to a bonding device it is determined using the `RTE_PCI_DRV_INTR_LSC` flag whether the device supports interrupts or whether the link status should be monitored by polling it.

9.2.2 Requirements / Limitations

The current implementation only supports devices that support the same speed and duplex to be added as a slaves to the same bonded device. The bonded device inherits these attributes from the first active slave added to the bonded device and then all further slaves added to the bonded device must support these parameters.

A bonding device must have a minimum of one slave before the bonding device itself can be started.

To use a bonding device dynamic RSS configuration feature effectively, it is also required, that all slaves should be RSS-capable and support, at least one common hash function available

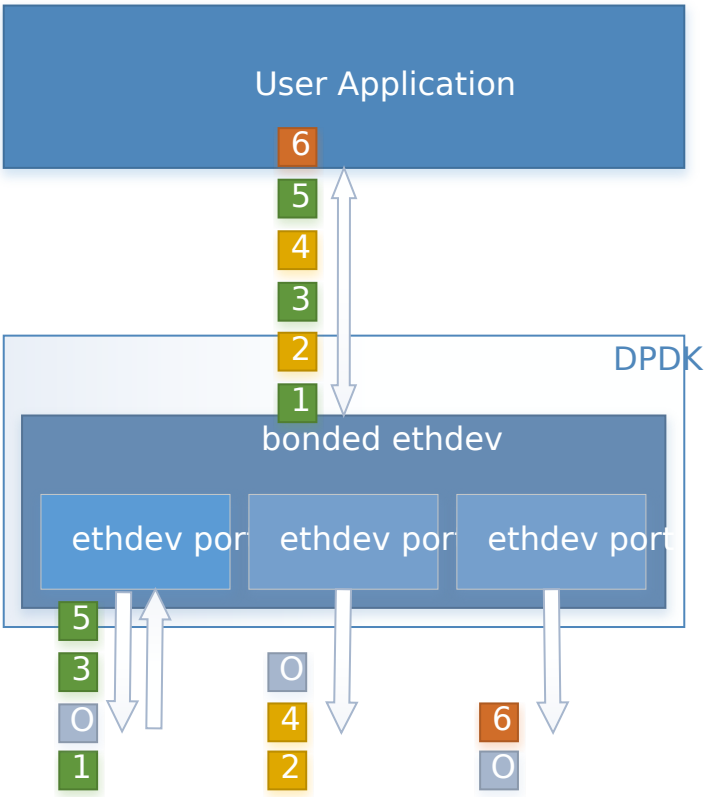


Fig. 9.6: Link Aggregation 802.3AD (Mode 4)

This mode provides dynamic link aggregation according to the 802.3ad specification. It negotiates and monitors aggregation groups that share the same speed and duplex settings using the selected balance transmit policy for balancing outgoing traffic.

DPDK implementation of this mode provide some additional requirements of the application.

1. It needs to call `rte_eth_tx_burst` and `rte_eth_rx_burst` with intervals period of less than 100ms.
2. Calls to `rte_eth_tx_burst` must have a buffer size of at least 2xN, where N is the number of slaves. This is a space required for LACP frames. Additionally LACP packets are included in the statistics, but they are not returned to the application.

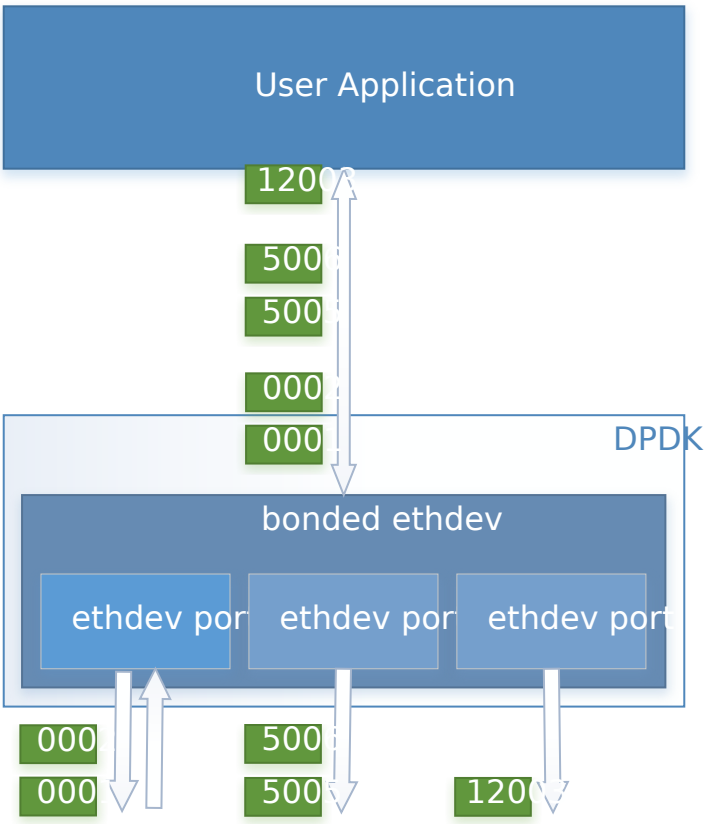


Fig. 9.7: Transmit Load Balancing (Mode 5)

This mode provides an adaptive transmit load balancing. It dynamically changes the transmitting slave, according to the computed load. Statistics are collected in 100ms intervals and scheduled every 10ms.

for each of them. Changing RSS key is only possible, when all slave devices support the same key size.

To prevent inconsistency on how slaves process packets, once a device is added to a bonding device, RSS configuration should be managed through the bonding device API, and not directly on the slave.

Like all other PMD, all functions exported by a PMD are lock-free functions that are assumed not to be invoked in parallel on different logical cores to work on the same target object.

It should also be noted that the PMD receive function should not be invoked directly on a slave devices after they have been to a bonded device since packets read directly from the slave device will no longer be available to the bonded device to read.

9.2.3 Configuration

Link bonding devices are created using the `rte_eth_bond_create` API which requires a unique device name, the bonding mode, and the socket Id to allocate the bonding device's resources on. The other configurable parameters for a bonded device are its slave devices, its primary slave, a user defined MAC address and transmission policy to use if the device is in balance XOR mode.

Slave Devices

Bonding devices support up to a maximum of `RTE_MAX_ETHPORTS` slave devices of the same speed and duplex. Ethernet devices can be added as a slave to a maximum of one bonded device. Slave devices are reconfigured with the configuration of the bonded device on being added to a bonded device.

The bonded also guarantees to return the MAC address of the slave device to its original value of removal of a slave from it.

Primary Slave

The primary slave is used to define the default port to use when a bonded device is in active backup mode. A different port will only be used if, and only if, the current primary port goes down. If the user does not specify a primary port it will default to being the first port added to the bonded device.

MAC Address

The bonded device can be configured with a user specified MAC address, this address will be inherited by the some/all slave devices depending on the operating mode. If the device is in active backup mode then only the primary device will have the user specified MAC, all other slaves will retain their original MAC address. In mode 0, 2, 3, 4 all slaves devices are configure with the bonded devices MAC address.

If a user defined MAC address is not defined then the bonded device will default to using the primary slaves MAC address.

Balance XOR Transmit Policies

There are 3 supported transmission policies for bonded device running in Balance XOR mode. Layer 2, Layer 2+3, Layer 3+4.

- **Layer 2:** Ethernet MAC address based balancing is the default transmission policy for Balance XOR bonding mode. It uses a simple XOR calculation on the source MAC address and destination MAC address of the packet and then calculate the modulus of this value to calculate the slave device to transmit the packet on.
- **Layer 2 + 3:** Ethernet MAC address & IP Address based balancing uses a combination of source/destination MAC addresses and the source/destination IP addresses of the data packet to decide which slave port the packet will be transmitted on.
- **Layer 3 + 4:** IP Address & UDP Port based balancing uses a combination of source/destination IP Address and the source/destination UDP ports of the packet of the data packet to decide which slave port the packet will be transmitted on.

All these policies support 802.1Q VLAN Ethernet packets, as well as IPv4, IPv6 and UDP protocols for load balancing.

9.3 Using Link Bonding Devices

The librte_pmd_bond library supports two modes of device creation, the libraries export full C API or using the EAL command line to statically configure link bonding devices at application startup. Using the EAL option it is possible to use link bonding functionality transparently without specific knowledge of the libraries API, this can be used, for example, to add bonding functionality, such as active backup, to an existing application which has no knowledge of the link bonding C API.

9.3.1 Using the Poll Mode Driver from an Application

Using the librte_pmd_bond libraries API it is possible to dynamically create and manage link bonding device from within any application. Link bonding devices are created using the `rte_eth_bond_create` API which requires a unique device name, the link bonding mode to initial the device in and finally the socket Id which to allocate the devices resources onto. After successful creation of a bonding device it must be configured using the generic Ethernet device configure API `rte_eth_dev_configure` and then the RX and TX queues which will be used must be setup using `rte_eth_tx_queue_setup` / `rte_eth_rx_queue_setup`.

Slave devices can be dynamically added and removed from a link bonding device using the `rte_eth_bond_slave_add` / `rte_eth_bond_slave_remove` APIs but at least one slave device must be added to the link bonding device before it can be started using `rte_eth_dev_start`.

The link status of a bonded device is dictated by that of its slaves, if all slave device link status are down or if all slaves are removed from the link bonding device then the link status of the bonding device will go down.

It is also possible to configure / query the configuration of the control parameters of a bonded device using the provided APIs `rte_eth_bond_mode_set/get`, `rte_eth_bond_primary_set/get`, `rte_eth_bond_mac_set/reset` and `rte_eth_bond_xmit_policy_set/get`.

9.3.2 Using Link Bonding Devices from the EAL Command Line

Link bonding devices can be created at application startup time using the `--vdev` EAL command line option. The device name must start with the net_bond prefix followed by numbers or letters. The name must be unique for each device. Each device can have multiple options arranged in a comma separated list. Multiple devices definitions can be arranged by calling the `--vdev` option multiple times.

Device names and bonding options must be separated by commas as shown below:

```
$RTE_TARGET/app/testpmd -c f -n 4 --vdev 'net_bond0,bond_opt0=..,bond opt1=..'--vdev 'net_bond1
```

Link Bonding EAL Options

There are multiple ways of definitions that can be assessed and combined as long as the following two rules are respected:

- A unique device name, in the format of net_bondX is provided, where X can be any combination of numbers and/or letters, and the name is no greater than 32 characters long.
- A least one slave device is provided with for each bonded device definition.
- The operation mode of the bonded device being created is provided.

The different options are:

- mode: Integer value defining the bonding mode of the device. Currently supports modes 0,1,2,3,4,5 (round-robin, active backup, balance, broadcast, link aggregation, transmit load balancing).

```
mode=2
```

- slave: Defines the PMD device which will be added as slave to the bonded device. This option can be selected multiple times, for each device to be added as a slave. Physical devices should be specified using their PCI address, in the format domain:bus:devid.function

```
slave=0000:0a:00.0,slave=0000:0a:00.1
```

- primary: Optional parameter which defines the primary slave port, is used in active backup mode to select the primary slave for data TX/RX if it is available. The primary port also is used to select the MAC address to use when it is not defined by the user. This defaults to the first slave added to the device if it is specified. The primary device must be a slave of the bonded device.

```
primary=0000:0a:00.0
```

- socket_id: Optional parameter used to select which socket on a NUMA device the bonded devices resources will be allocated on.

```
socket_id=0
```

- mac: Optional parameter to select a MAC address for link bonding device, this overrides the value of the primary slave device.

```
mac=00:1e:67:1d:fd:1d
```

- xmit_policy: Optional parameter which defines the transmission policy when the bonded device is in balance mode. If not user specified this defaults to l2 (layer 2) forwarding, the other transmission policies available are l23 (layer 2+3) and l34 (layer 3+4)

```
xmit_policy=l23
```

- lsc_poll_period_ms: Optional parameter which defines the polling interval in milli-seconds at which devices which don't support lsc interrupts are checked for a change in the devices link status

```
lsc_poll_period_ms=100
```

- up_delay: Optional parameter which adds a delay in milli-seconds to the propagation of a devices link status changing to up, by default this parameter is zero.

```
up_delay=10
```

- down_delay: Optional parameter which adds a delay in milli-seconds to the propagation of a devices link status changing to down, by default this parameter is zero.

```
down_delay=50
```

Examples of Usage

Create a bonded device in round robin mode with two slaves specified by their PCI address:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'net_bond0,mode=0, slave=0000:00a:00.01,slave=0000
```

Create a bonded device in round robin mode with two slaves specified by their PCI address and an overriding MAC address:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'net_bond0,mode=0, slave=0000:00a:00.01,slave=0000
```

Create a bonded device in active backup mode with two slaves specified, and a primary slave specified by their PCI addresses:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'net_bond0,mode=1, slave=0000:00a:00.01,slave=0000
```

Create a bonded device in balance mode with two slaves specified by their PCI addresses, and a transmission policy of layer 3 + 4 forwarding:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'net_bond0,mode=2, slave=0000:00a:00.01,slave=0000
```

CHAPTER

TEN

TIMER LIBRARY

The Timer library provides a timer service to DPDK execution units to enable execution of callback functions asynchronously. Features of the library are:

- Timers can be periodic (multi-shot) or single (one-shot).
- Timers can be loaded from one core and executed on another. It has to be specified in the call to rte_timer_reset().
- Timers provide high precision (depends on the call frequency to rte_timer_manage() that checks timer expiration for the local core).
- If not required in the application, timers can be disabled at compilation time by not calling the rte_timer_manage() to increase performance.

The timer library uses the rte_get_timer_cycles() function that uses the High Precision Event Timer (HPET) or the CPUs Time Stamp Counter (TSC) to provide a reliable time reference.

This library provides an interface to add, delete and restart a timer. The API is based on BSD callout() with a few differences. Refer to the callout manual.

10.1 Implementation Details

Timers are tracked on a per-lcore basis, with all pending timers for a core being maintained in order of timer expiry in a skiplist data structure. The skiplist used has ten levels and each entry in the table appears in each level with probability ¼^level. This means that all entries are present in level 0, 1 in every 4 entries is present at level 1, one in every 16 at level 2 and so on up to level 9. This means that adding and removing entries from the timer list for a core can be done in log(n) time, up to 4^10 entries, that is, approximately 1,000,000 timers per lcore.

A timer structure contains a special field called status, which is a union of a timer state (stopped, pending, running, config) and an owner (lcore id). Depending on the timer state, we know if a timer is present in a list or not:

- STOPPED: no owner, not in a list
- CONFIG: owned by a core, must not be modified by another core, maybe in a list or not, depending on previous state
- PENDING: owned by a core, present in a list
- RUNNING: owned by a core, must not be modified by another core, present in a list

Resetting or stopping a timer while it is in a CONFIG or RUNNING state is not allowed. When modifying the state of a timer, a Compare And Swap instruction should be used to guarantee that the status (state+owner) is modified atomically.

Inside the rte_timer_manage() function, the skiplist is used as a regular list by iterating along the level 0 list, which contains all timer entries, until an entry which has not yet expired has been encountered. To improve performance in the case where there are entries in the timer list but none of those timers have yet expired, the expiry time of the first list entry is maintained within the per-core timer list structure itself. On 64-bit platforms, this value can be checked without the need to take a lock on the overall structure. (Since expiry times are maintained as 64-bit values, a check on the value cannot be done on 32-bit platforms without using either a compare-and-swap (CAS) instruction or using a lock, so this additional check is skipped in favor of checking as normal once the lock has been taken.) On both 64-bit and 32-bit platforms, a call to rte_timer_manage() returns without taking a lock in the case where the timer list for the calling core is empty.

10.2 Use Cases

The timer library is used for periodic calls, such as garbage collectors, or some state machines (ARP, bridging, and so on).

10.3 References

- callout manual - The callout facility that provides timers with a mechanism to execute a function at a given time.
- HPET - Information about the High Precision Event Timer (HPET).

CHAPTER

ELEVEN

HASH LIBRARY

The DPDK provides a Hash Library for creating hash table for fast lookup. The hash table is a data structure optimized for searching through a set of entries that are each identified by a unique key. For increased performance the DPDK Hash requires that all the keys have the same number of bytes which is set at the hash creation time.

11.1 Hash API Overview

The main configuration parameters for the hash are:

- Total number of hash entries
- Size of the key in bytes

The hash also allows the configuration of some low-level implementation related parameters such as:

- Hash function to translate the key into a bucket index

The main methods exported by the hash are:

- Add entry with key: The key is provided as input. If a new entry is successfully added to the hash for the specified key, or there is already an entry in the hash for the specified key, then the position of the entry is returned. If the operation was not successful, for example due to lack of free entries in the hash, then a negative value is returned;
- Delete entry with key: The key is provided as input. If an entry with the specified key is found in the hash, then the entry is removed from the hash and the position where the entry was found in the hash is returned. If no entry with the specified key exists in the hash, then a negative value is returned
- Lookup for entry with key: The key is provided as input. If an entry with the specified key is found in the hash (lookup hit), then the position of the entry is returned, otherwise (lookup miss) a negative value is returned.

Apart from these method explained above, the API allows the user three more options:

- Add / lookup / delete with key and precomputed hash: Both the key and its precomputed hash are provided as input. This allows the user to perform these operations faster, as hash is already computed.
- Add / lookup with key and data: A pair of key-value is provided as input. This allows the user to store not only the key, but also data which may be either a 8-byte integer or a pointer to external data (if data size is more than 8 bytes).

- Combination of the two options above: User can provide key, precomputed hash and data.

Also, the API contains a method to allow the user to look up entries in bursts, achieving higher performance than looking up individual entries, as the function prefetches next entries at the time it is operating with the first ones, which reduces significantly the impact of the necessary memory accesses. Notice that this method uses a pipeline of 8 entries (4 stages of 2 entries), so it is highly recommended to use at least 8 entries per burst.

The actual data associated with each key can be either managed by the user using a separate table that mirrors the hash in terms of number of entries and position of each entry, as shown in the Flow Classification use case describes in the following sections, or stored in the hash table itself.

The example hash tables in the L2/L3 Forwarding sample applications defines which port to forward a packet to based on a packet flow identified by the five-tuple lookup. However, this table could also be used for more sophisticated features and provide many other functions and actions that could be performed on the packets and flows.

11.2 Multi-process support

The hash library can be used in a multi-process environment, minding that only lookups are thread-safe. The only function that can only be used in single-process mode is rte_hash_set_cmp_func(), which sets up a custom compare function, which is assigned to a function pointer (therefore, it is not supported in multi-process mode).

11.3 Implementation Details

The hash table has two main tables:

- First table is an array of entries which is further divided into buckets, with the same number of consecutive array entries in each bucket. Each entry contains the computed primary and secondary hashes of a given key (explained below), and an index to the second table.
- The second table is an array of all the keys stored in the hash table and its data associated to each key.

The hash library uses the cuckoo hash method to resolve collisions. For any input key, there are two possible buckets (primary and secondary/alternative location) where that key can be stored in the hash, therefore only the entries within those bucket need to be examined when the key is looked up. The lookup speed is achieved by reducing the number of entries to be scanned from the total number of hash entries down to the number of entries in the two hash buckets, as opposed to the basic method of linearly scanning all the entries in the array. The hash uses a hash function (configurable) to translate the input key into a 4-byte key signature. The bucket index is the key signature modulo the number of hash buckets.

Once the buckets are identified, the scope of the hash add, delete and lookup operations is reduced to the entries in those buckets (it is very likely that entries are in the primary bucket).

To speed up the search logic within the bucket, each hash entry stores the 4-byte key signature together with the full key for each hash entry. For large key sizes, comparing the input key

against a key from the bucket can take significantly more time than comparing the 4-byte signature of the input key against the signature of a key from the bucket. Therefore, the signature comparison is done first and the full key comparison done only when the signatures matches. The full key comparison is still necessary, as two input keys from the same bucket can still potentially have the same 4-byte hash signature, although this event is relatively rare for hash functions providing good uniform distributions for the set of input keys.

Example of lookup:

First of all, the primary bucket is identified and entry is likely to be stored there. If signature was stored there, we compare its key against the one provided and return the position where it was stored and/or the data associated to that key if there is a match. If signature is not in the primary bucket, the secondary bucket is looked up, where same procedure is carried out. If there is no match there either, key is considered not to be in the table.

Example of addition:

Like lookup, the primary and secondary buckets are identified. If there is an empty slot in the primary bucket, primary and secondary signatures are stored in that slot, key and data (if any) are added to the second table and an index to the position in the second table is stored in the slot of the first table. If there is no space in the primary bucket, one of the entries on that bucket is pushed to its alternative location, and the key to be added is inserted in its position. To know where the alternative bucket of the evicted entry is, the secondary signature is looked up and alternative bucket index is calculated from doing the modulo, as seen above. If there is room in the alternative bucket, the evicted entry is stored in it. If not, same process is repeated (one of the entries gets pushed) until a non full bucket is found. Notice that despite all the entry movement in the first table, the second table is not touched, which would impact greatly in performance.

In the very unlikely event that table enters in a loop where same entries are being evicted indefinitely, key is considered not able to be stored. With random keys, this method allows the user to get around 90% of the table utilization, without having to drop any stored entry (LRU) or allocate more memory (extended buckets).

11.4 Entry distribution in hash table

As mentioned above, Cuckoo hash implementation pushes elements out of their bucket, if there is a new entry to be added which primary location coincides with their current bucket, being pushed to their alternative location. Therefore, as user adds more entries to the hash table, distribution of the hash values in the buckets will change, being most of them in their primary location and a few in their secondary location, which the later will increase, as table gets busier. This information is quite useful, as performance may be lower as more entries are evicted to their secondary location.

See the tables below showing example entry distribution as table utilization increases.

Table 11.1: Entry distribution measured with an example table with 1024 random entries using jhash algorithm

% Table used	% In Primary location	% In Secondary location
25	100	0
50	96.1	3.9
75	88.2	11.8
80	86.3	13.7
85	83.1	16.9
90	77.3	22.7
95.8	64.5	35.5

Table 11.2: Entry distribution measured with an example table with 1 million random entries using jhash algorithm

% Table used	% In Primary location	% In Secondary location
50	96	4
75	86.9	13.1
80	83.9	16.1
85	80.1	19.9
90	74.8	25.2
94.5	67.4	32.6

Note: Last values on the tables above are the average maximum table utilization with random keys and using Jenkins hash function.

11.5 Use Case: Flow Classification

Flow classification is used to map each input packet to the connection/flow it belongs to. This operation is necessary as the processing of each input packet is usually done in the context of their connection, so the same set of operations is applied to all the packets from the same flow.

Applications using flow classification typically have a flow table to manage, with each separate flow having an entry associated with it in this table. The size of the flow table entry is application specific, with typical values of 4, 16, 32 or 64 bytes.

Each application using flow classification typically has a mechanism defined to uniquely identify a flow based on a number of fields read from the input packet that make up the flow key. One example is to use the DiffServ 5-tuple made up of the following fields of the IP and transport layer packet headers: Source IP Address, Destination IP Address, Protocol, Source Port, Destination Port.

The DPDK hash provides a generic method to implement an application specific flow classification mechanism. Given a flow table implemented as an array, the application should create a hash object with the same number of entries as the flow table and with the hash key size set to the number of bytes in the selected flow key.

The flow table operations on the application side are described below:

- Add flow: Add the flow key to hash. If the returned position is valid, use it to access the flow entry in the flow table for adding a new flow or updating the information associated with an existing flow. Otherwise, the flow addition failed, for example due to lack of free entries for storing new flows.
- Delete flow: Delete the flow key from the hash. If the returned position is valid, use it to access the flow entry in the flow table to invalidate the information associated with the flow.
- Lookup flow: Lookup for the flow key in the hash. If the returned position is valid (flow lookup hit), use the returned position to access the flow entry in the flow table. Otherwise (flow lookup miss) there is no flow registered for the current packet.

11.6 References

- Donald E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition), 1998, Addison-Wesley Professional

CHAPTER

TWELVE

LPM LIBRARY

The DPDK LPM library component implements the Longest Prefix Match (LPM) table search method for 32-bit keys that is typically used to find the best route match in IP forwarding applications.

12.1 LPM API Overview

The main configuration parameter for LPM component instances is the maximum number of rules to support. An LPM prefix is represented by a pair of parameters (32- bit key, depth), with depth in the range of 1 to 32. An LPM rule is represented by an LPM prefix and some user data associated with the prefix. The prefix serves as the unique identifier of the LPM rule. In this implementation, the user data is 1-byte long and is called next hop, in correlation with its main use of storing the ID of the next hop in a routing table entry.

The main methods exported by the LPM component are:

- Add LPM rule: The LPM rule is provided as input. If there is no rule with the same prefix present in the table, then the new rule is added to the LPM table. If a rule with the same prefix is already present in the table, the next hop of the rule is updated. An error is returned when there is no available rule space left.
- Delete LPM rule: The prefix of the LPM rule is provided as input. If a rule with the specified prefix is present in the LPM table, then it is removed.
- Lookup LPM key: The 32-bit key is provided as input. The algorithm selects the rule that represents the best match for the given key and returns the next hop of that rule. In the case that there are multiple rules present in the LPM table that have the same 32-bit key, the algorithm picks the rule with the highest depth as the best match rule, which means that the rule has the highest number of most significant bits matching between the input key and the rule key.

12.2 Implementation Details

The current implementation uses a variation of the DIR-24-8 algorithm that trades memory usage for improved LPM lookup speed. The algorithm allows the lookup operation to be performed with typically a single memory read access. In the statistically rare case when the best match rule is having a depth bigger than 24, the lookup operation requires two memory read accesses. Therefore, the performance of the LPM lookup operation is greatly influenced by whether the specific memory location is present in the processor cache or not.

The main data structure is built using the following elements:

- A table with 2^24 entries.
- A number of tables (RTE_LPM_TBL8_NUM_GROUPS) with 2^8 entries.

The first table, called tbl24, is indexed using the first 24 bits of the IP address to be looked up, while the second table(s), called tbl8, is indexed using the last 8 bits of the IP address. This means that depending on the outcome of trying to match the IP address of an incoming packet to the rule stored in the tbl24 we might need to continue the lookup process in the second level.

Since every entry of the tbl24 can potentially point to a tbl8, ideally, we would have 2^24 tbl8s, which would be the same as having a single table with 2^32 entries. This is not feasible due to resource restrictions. Instead, this approach takes advantage of the fact that rules longer than 24 bits are very rare. By splitting the process in two different tables/levels and limiting the number of tbl8s, we can greatly reduce memory consumption while maintaining a very good lookup speed (one memory access, most of the times).

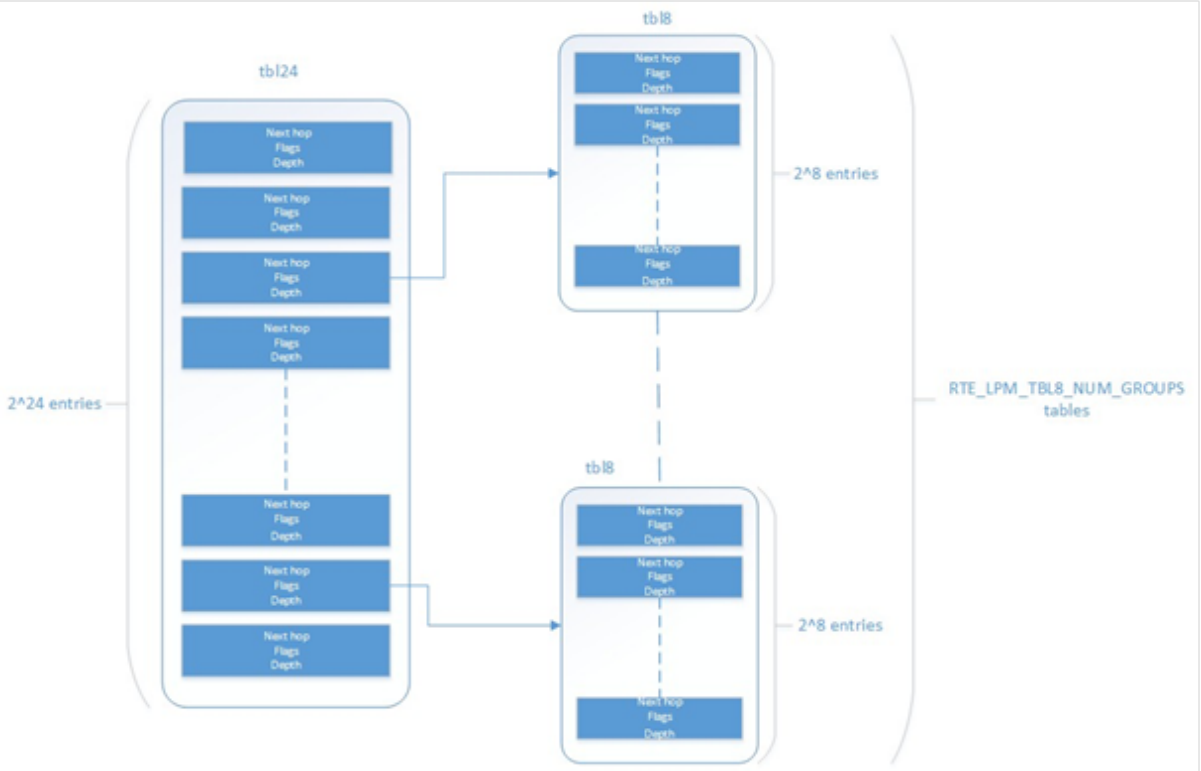


Fig. 12.1: Table split into different levels

An entry in tbl24 contains the following fields:

- next hop / index to the tbl8
- valid flag
- external entry flag
- depth of the rule (length)

The first field can either contain a number indicating the tbl8 in which the lookup process should continue or the next hop itself if the longest prefix match has already been found. The two flags are used to determine whether the entry is valid or not and whether the search process have

finished or not respectively. The depth or length of the rule is the number of bits of the rule that is stored in a specific entry.

An entry in a tbl8 contains the following fields:

- next hop
- valid
- valid group
- depth

Next hop and depth contain the same information as in the tbl24. The two flags show whether the entry and the table are valid respectively.

The other main data structure is a table containing the main information about the rules (IP and next hop). This is a higher level table, used for different things:

- Check whether a rule already exists or not, prior to addition or deletion, without having to actually perform a lookup.
- When deleting, to check whether there is a rule containing the one that is to be deleted. This is important, since the main data structure will have to be updated accordingly.

12.2.1 Addition

When adding a rule, there are different possibilities. If the rule's depth is exactly 24 bits, then:

- Use the rule (IP address) as an index to the tbl24.
- If the entry is invalid (i.e. it doesn't already contain a rule) then set its next hop to its value, the valid flag to 1 (meaning this entry is in use), and the external entry flag to 0 (meaning the lookup process ends at this point, since this is the longest prefix that matches).

If the rule's depth is exactly 32 bits, then:

- Use the first 24 bits of the rule as an index to the tbl24.
- If the entry is invalid (i.e. it doesn't already contain a rule) then look for a free tbl8, set the index to the tbl8 to this value, the valid flag to 1 (meaning this entry is in use), and the external entry flag to 1 (meaning the lookup process must continue since the rule hasn't been explored completely).

If the rule's depth is any other value, prefix expansion must be performed. This means the rule is copied to all the entries (as long as they are not in use) which would also cause a match.

As a simple example, let's assume the depth is 20 bits. This means that there are 2^(24 - 20) = 16 different combinations of the first 24 bits of an IP address that would cause a match. Hence, in this case, we copy the exact same entry to every position indexed by one of these combinations.

By doing this we ensure that during the lookup process, if a rule matching the IP address exists, it is found in either one or two memory accesses, depending on whether we need to move to the next table or not. Prefix expansion is one of the keys of this algorithm, since it improves the speed dramatically by adding redundancy.

12.2.2 Lookup

The lookup process is much simpler and quicker. In this case:

- Use the first 24 bits of the IP address as an index to the tbl24. If the entry is not in use, then it means we don't have a rule matching this IP. If it is valid and the external entry flag is set to 0, then the next hop is returned.
- If it is valid and the external entry flag is set to 1, then we use the tbl8 index to find out the tbl8 to be checked, and the last 8 bits of the IP address as an index to this table. Similarly, if the entry is not in use, then we don't have a rule matching this IP address. If it is valid then the next hop is returned.

12.2.3 Limitations in the Number of Rules

There are different things that limit the number of rules that can be added. The first one is the maximum number of rules, which is a parameter passed through the API. Once this number is reached, it is not possible to add any more rules to the routing table unless one or more are removed.

The second reason is an intrinsic limitation of the algorithm. As explained before, to avoid high memory consumption, the number of tbl8s is limited in compilation time (this value is by default 256). If we exhaust tbl8s, we won't be able to add any more rules. How many of them are necessary for a specific routing table is hard to determine in advance.

A tbl8 is consumed whenever we have a new rule with depth bigger than 24, and the first 24 bits of this rule are not the same as the first 24 bits of a rule previously added. If they are, then the new rule will share the same tbl8 than the previous one, since the only difference between the two rules is within the last byte.

With the default value of 256, we can have up to 256 rules longer than 24 bits that differ on their first three bytes. Since routes longer than 24 bits are unlikely, this shouldn't be a problem in most setups. Even if it is, however, the number of tbl8s can be modified.

12.2.4 Use Case: IPv4 Forwarding

The LPM algorithm is used to implement Classless Inter-Domain Routing (CIDR) strategy used by routers implementing IPv4 forwarding.

12.2.5 References

- RFC1519 Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy, http://www.ietf.org/rfc/rfc1519
- Pankaj Gupta, Algorithms for Routing Lookups and Packet Classification, PhD Thesis, Stanford University, 2000 (http://klamath.stanford.edu/~pankaj/thesis/ thesis_1sided.pdf)

CHAPTER

THIRTEEN

LPM6 LIBRARY

The LPM6 (LPM for IPv6) library component implements the Longest Prefix Match (LPM) table search method for 128-bit keys that is typically used to find the best match route in IPv6 forwarding applications.

13.1 LPM6 API Overview

The main configuration parameters for the LPM6 library are:

- Maximum number of rules: This defines the size of the table that holds the rules, and therefore the maximum number of rules that can be added.
- Number of tbl8s: A tbl8 is a node of the trie that the LPM6 algorithm is based on.

This parameter is related to the number of rules you can have, but there is no way to accurately predict the number needed to hold a specific number of rules, since it strongly depends on the depth and IP address of every rule. One tbl8 consumes 1 kb of memory. As a recommendation, 65536 tbl8s should be sufficient to store several thousand IPv6 rules, but the number can vary depending on the case.

An LPM prefix is represented by a pair of parameters (128-bit key, depth), with depth in the range of 1 to 128. An LPM rule is represented by an LPM prefix and some user data associated with the prefix. The prefix serves as the unique identifier for the LPM rule. In this implementation, the user data is 1-byte long and is called “next hop”, which corresponds to its main use of storing the ID of the next hop in a routing table entry.

The main methods exported for the LPM component are:

- Add LPM rule: The LPM rule is provided as input. If there is no rule with the same prefix present in the table, then the new rule is added to the LPM table. If a rule with the same prefix is already present in the table, the next hop of the rule is updated. An error is returned when there is no available space left.
- Delete LPM rule: The prefix of the LPM rule is provided as input. If a rule with the specified prefix is present in the LPM table, then it is removed.
- Lookup LPM key: The 128-bit key is provided as input. The algorithm selects the rule that represents the best match for the given key and returns the next hop of that rule. In the case that there are multiple rules present in the LPM table that have the same 128-bit value, the algorithm picks the rule with the highest depth as the best match rule, which means the rule has the highest number of most significant bits matching between the input key and the rule key.

13.1.1 Implementation Details

This is a modification of the algorithm used for IPv4 (see *Implementation Details*). In this case, instead of using two levels, one with a tbl24 and a second with a tbl8, 14 levels are used.

The implementation can be seen as a multi-bit trie where the *stride* or number of bits inspected on each level varies from level to level. Specifically, 24 bits are inspected on the root node, and the remaining 104 bits are inspected in groups of 8 bits. This effectively means that the trie has 14 levels at the most, depending on the rules that are added to the table.

The algorithm allows the lookup operation to be performed with a number of memory accesses that directly depends on the length of the rule and whether there are other rules with bigger depths and the same key in the data structure. It can vary from 1 to 14 memory accesses, with 5 being the average value for the lengths that are most commonly used in IPv6.

The main data structure is built using the following elements:

- A table with 224 entries
- A number of tables, configurable by the user through the API, with 28 entries

The first table, called tbl24, is indexed using the first 24 bits of the IP address be looked up, while the rest of the tables, called tbl8s, are indexed using the rest of the bytes of the IP address, in chunks of 8 bits. This means that depending on the outcome of trying to match the IP address of an incoming packet to the rule stored in the tbl24 or the subsequent tbl8s we might need to continue the lookup process in deeper levels of the tree.

Similar to the limitation presented in the algorithm for IPv4, to store every possible IPv6 rule, we would need a table with 2^128 entries. This is not feasible due to resource restrictions.

By splitting the process in different tables/levels and limiting the number of tbl8s, we can greatly reduce memory consumption while maintaining a very good lookup speed (one memory access per level).

An entry in a table contains the following fields:

- next hop / index to the tbl8
- depth of the rule (length)
- valid flag
- valid group flag
- external entry flag

The first field can either contain a number indicating the tbl8 in which the lookup process should continue or the next hop itself if the longest prefix match has already been found. The depth or length of the rule is the number of bits of the rule that is stored in a specific entry. The flags are used to determine whether the entry/table is valid or not and whether the search process have finished or not respectively.

Both types of tables share the same structure.

The other main data structure is a table containing the main information about the rules (IP, next hop and depth). This is a higher level table, used for different things:

- Check whether a rule already exists or not, prior to addition or deletion, without having to actually perform a lookup.

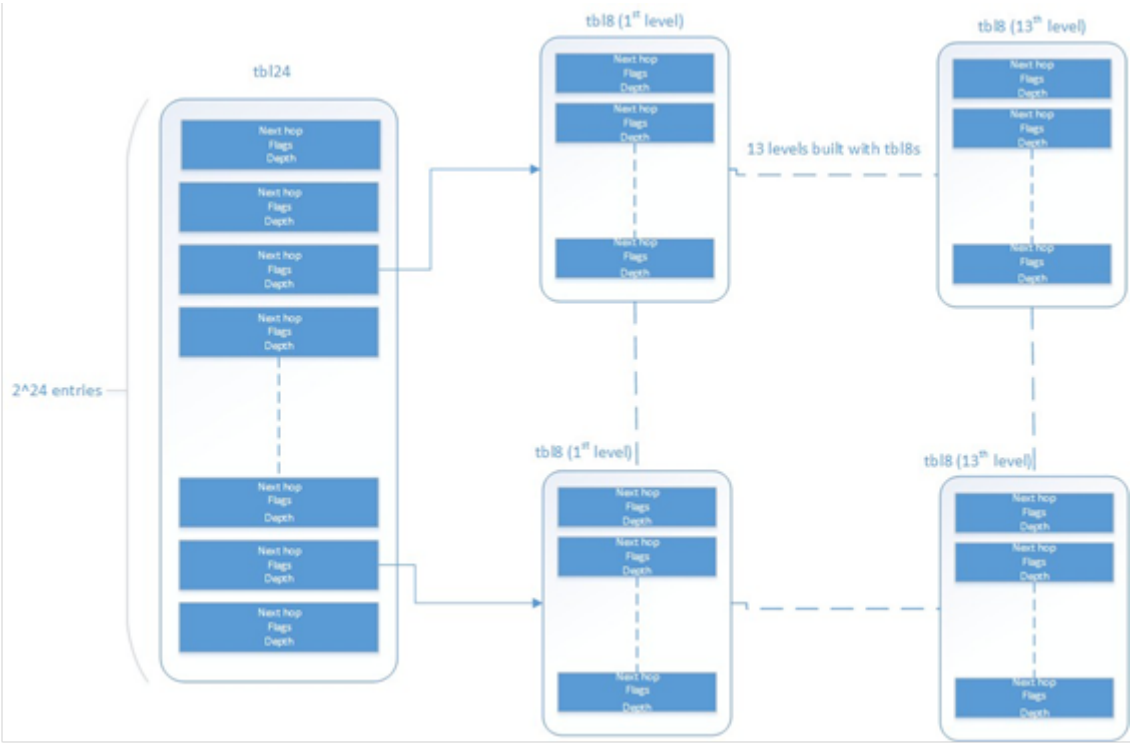


Fig. 13.1: Table split into different levels

When deleting, to check whether there is a rule containing the one that is to be deleted. This is important, since the main data structure will have to be updated accordingly.

13.1.2 Addition

When adding a rule, there are different possibilities. If the rule's depth is exactly 24 bits, then:

- Use the rule (IP address) as an index to the tbl24.
- If the entry is invalid (i.e. it doesn't already contain a rule) then set its next hop to its value, the valid flag to 1 (meaning this entry is in use), and the external entry flag to 0 (meaning the lookup process ends at this point, since this is the longest prefix that matches).

If the rule's depth is bigger than 24 bits but a multiple of 8, then:

- Use the first 24 bits of the rule as an index to the tbl24.
- If the entry is invalid (i.e. it doesn't already contain a rule) then look for a free tbl8, set the index to the tbl8 to this value, the valid flag to 1 (meaning this entry is in use), and the external entry flag to 1 (meaning the lookup process must continue since the rule hasn't been explored completely).
- Use the following 8 bits of the rule as an index to the next tbl8.
- Repeat the process until the tbl8 at the right level (depending on the depth) has been reached and fill it with the next hop, setting the next entry flag to 0.

If the rule's depth is any other value, prefix expansion must be performed. This means the rule is copied to all the entries (as long as they are not in use) which would also cause a match.

As a simple example, let's assume the depth is 20 bits. This means that there are 2^(24-20) = 16 different combinations of the first 24 bits of an IP address that would cause a match. Hence, in this case, we copy the exact same entry to every position indexed by one of these combinations.

By doing this we ensure that during the lookup process, if a rule matching the IP address exists, it is found in, at the most, 14 memory accesses, depending on how many times we need to move to the next table. Prefix expansion is one of the keys of this algorithm, since it improves the speed dramatically by adding redundancy.

Prefix expansion can be performed at any level. So, for example, is the depth is 34 bits, it will be performed in the third level (second tbl8-based level).

13.1.3 Lookup

The lookup process is much simpler and quicker. In this case:

- Use the first 24 bits of the IP address as an index to the tbl24. If the entry is not in use, then it means we don't have a rule matching this IP. If it is valid and the external entry flag is set to 0, then the next hop is returned.
- If it is valid and the external entry flag is set to 1, then we use the tbl8 index to find out the tbl8 to be checked, and the next 8 bits of the IP address as an index to this table. Similarly, if the entry is not in use, then we don't have a rule matching this IP address. If it is valid then check the external entry flag for a new tbl8 to be inspected.
- Repeat the process until either we find an invalid entry (lookup miss) or a valid entry with the external entry flag set to 0. Return the next hop in the latter case.

13.1.4 Limitations in the Number of Rules

There are different things that limit the number of rules that can be added. The first one is the maximum number of rules, which is a parameter passed through the API. Once this number is reached, it is not possible to add any more rules to the routing table unless one or more are removed.

The second limitation is in the number of tbl8s available. If we exhaust tbl8s, we won't be able to add any more rules. How to know how many of them are necessary for a specific routing table is hard to determine in advance.

In this algorithm, the maximum number of tbl8s a single rule can consume is 13, which is the number of levels minus one, since the first three bytes are resolved in the tbl24. However:

- Typically, on IPv6, routes are not longer than 48 bits, which means rules usually take up to 3 tbl8s.

As explained in the LPM for IPv4 algorithm, it is possible and very likely that several rules will share one or more tbl8s, depending on what their first bytes are. If they share the same first 24 bits, for instance, the tbl8 at the second level will be shared. This might happen again in deeper levels, so, effectively, two 48 bit-long rules may use the same three tbl8s if the only difference is in their last byte.

The number of tbl8s is a parameter exposed to the user through the API in this version of the algorithm, due to its impact in memory consumption and the number or rules that can be added to the LPM table. One tbl8 consumes 1 kilobyte of memory.

13.2 Use Case: IPv6 Forwarding

The LPM algorithm is used to implement the Classless Inter-Domain Routing (CIDR) strategy used by routers implementing IP forwarding.

CHAPTER

FOURTEEN

PACKET DISTRIBUTOR LIBRARY

The DPDK Packet Distributor library is a library designed to be used for dynamic load balancing of traffic while supporting single packet at a time operation. When using this library, the logical cores in use are to be considered in two roles: firstly a distributor lcore, which is responsible for load balancing or distributing packets, and a set of worker lcores which are responsible for receiving the packets from the distributor and operating on them. The model of operation is shown in the diagram below.

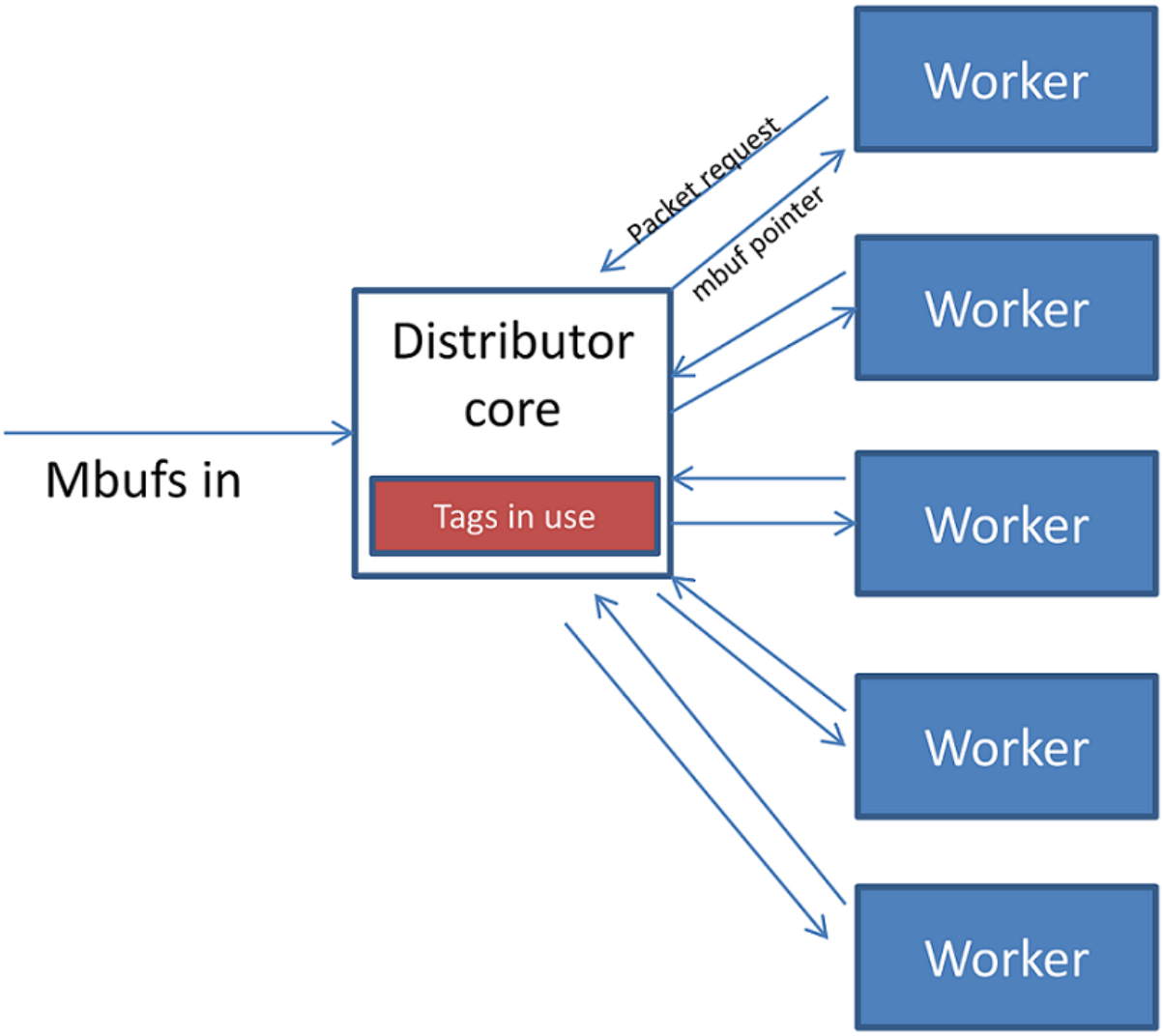


Fig. 14.1: Packet Distributor mode of operation

14.1 Distributor Core Operation

The distributor core does the majority of the processing for ensuring that packets are fairly shared among workers. The operation of the distributor is as follows:

1. Packets are passed to the distributor component by having the distributor lcore thread call the "rte_distributor_process()" API
2. The worker lcores all share a single cache line with the distributor core in order to pass messages and packets to and from the worker. The process API call will poll all the worker cache lines to see what workers are requesting packets.
3. As workers request packets, the distributor takes packets from the set of packets passed in and distributes them to the workers. As it does so, it examines the "tag" – stored in the RSS hash field in the mbuf – for each packet and records what tags are being processed by each worker.
4. If the next packet in the input set has a tag which is already being processed by a worker, then that packet will be queued up for processing by that worker and given to it in preference to other packets when that work next makes a request for work. This ensures that no two packets with the same tag are processed in parallel, and that all packets with the same tag are processed in input order.
5. Once all input packets passed to the process API have either been distributed to workers or been queued up for a worker which is processing a given tag, then the process API returns to the caller.

Other functions which are available to the distributor lcore are:

- rte_distributor_returned_pkts()
- rte_distributor_flush()
- rte_distributor_clear_returns()

Of these the most important API call is "rte_distributor_returned_pkts()" which should only be called on the lcore which also calls the process API. It returns to the caller all packets which have finished processing by all worker cores. Within this set of returned packets, all packets sharing the same tag will be returned in their original order.

NOTE: If worker lcores buffer up packets internally for transmission in bulk afterwards, the packets sharing a tag will likely get out of order. Once a worker lcore requests a new packet, the distributor assumes that it has completely finished with the previous packet and therefore that additional packets with the same tag can safely be distributed to other workers – who may then flush their buffered packets sooner and cause packets to get out of order.

NOTE: No packet ordering guarantees are made about packets which do not share a common packet tag.

Using the process and returned_pkts API, the following application workflow can be used, while allowing packet order within a packet flow – identified by a tag – to be maintained.

The flush and clear_returns API calls, mentioned previously, are likely of less use that the process and returned_pkts APIS, and are principally provided to aid in unit testing of the library. Descriptions of these functions and their use can be found in the DPDK API Reference document.

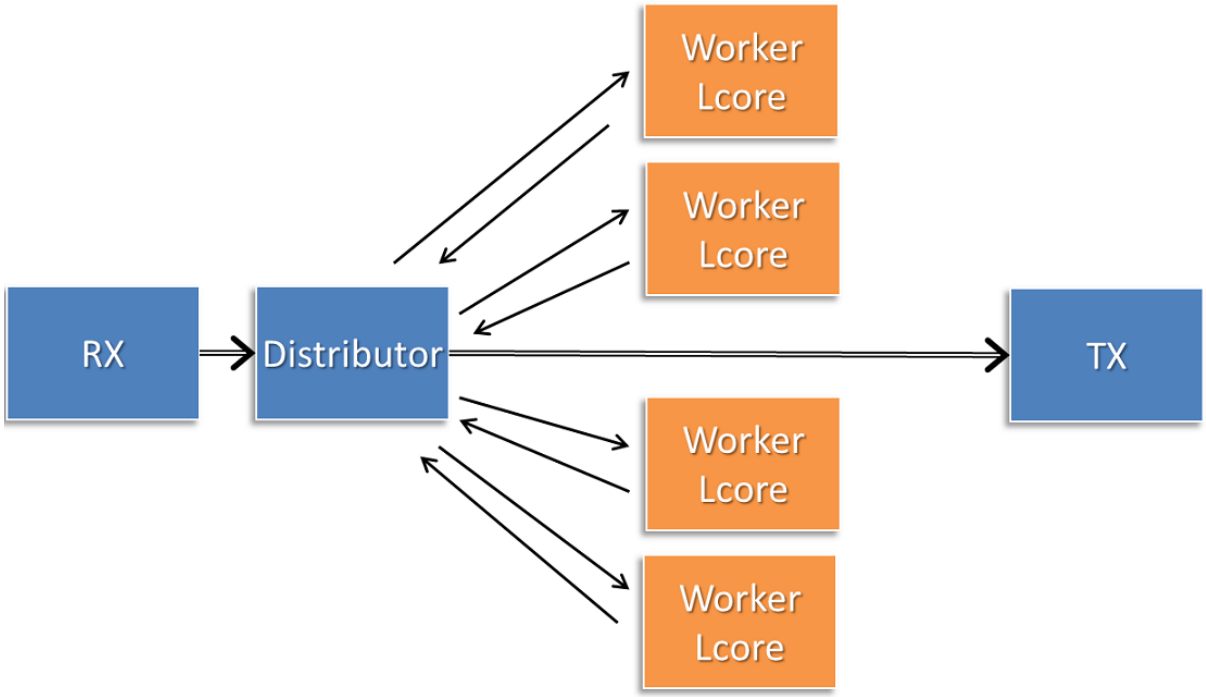


Fig. 14.2: Application workflow

14.2 Worker Operation

Worker cores are the cores which do the actual manipulation of the packets distributed by the packet distributor. Each worker calls "rte_distributor_get_pkt()" API to request a new packet when it has finished processing the previous one. [The previous packet should be returned to the distributor component by passing it as the final parameter to this API call.]

Since it may be desirable to vary the number of worker cores, depending on the traffic load i.e. to save power at times of lighter load, it is possible to have a worker stop processing packets by calling "rte_distributor_return_pkt()" to indicate that it has finished the current packet and does not want a new one.

CHAPTER

FIFTEEN

REORDER LIBRARY

The Reorder Library provides a mechanism for reordering mbufs based on their sequence number.

15.1 Operation

The reorder library is essentially a buffer that reorders mbufs. The user inserts out of order mbufs into the reorder buffer and pulls in-order mbufs from it.

At a given time, the reorder buffer contains mbufs whose sequence number are inside the sequence window. The sequence window is determined by the minimum sequence number and the number of entries that the buffer was configured to hold. For example, given a reorder buffer with 200 entries and a minimum sequence number of 350, the sequence window has low and high limits of 350 and 550 respectively.

When inserting mbufs, the reorder library differentiates between valid, early and late mbufs depending on the sequence number of the inserted mbuf:

- valid: the sequence number is inside the window.
- late: the sequence number is outside the window and less than the low limit.
- early: the sequence number is outside the window and greater than the high limit.

The reorder buffer directly returns late mbufs and tries to accommodate early mbufs.

15.2 Implementation Details

The reorder library is implemented as a pair of buffers, which referred to as the *Order* buffer and the *Ready* buffer.

On an insert call, valid mbufs are inserted directly into the Order buffer and late mbufs are returned to the user with an error.

In the case of early mbufs, the reorder buffer will try to move the window (incrementing the minimum sequence number) so that the mbuf becomes a valid one. To that end, mbufs in the Order buffer are moved into the Ready buffer. Any mbufs that have not arrived yet are ignored and therefore will become late mbufs. This means that as long as there is room in the Ready buffer, the window will be moved to accommodate early mbufs that would otherwise be outside the reordering window.

For example, assuming that we have a buffer of 200 entries with a 350 minimum sequence number, and we need to insert an early mbuf with 565 sequence number. That means that we would need to move the windows at least 15 positions to accommodate the mbuf. The reorder buffer would try to move mbufs from at least the next 15 slots in the Order buffer to the Ready buffer, as long as there is room in the Ready buffer. Any gaps in the Order buffer at that point are skipped, and those packet will be reported as late packets when they arrive. The process of moving packets to the Ready buffer continues beyond the minimum required until a gap, i.e. missing mbuf, in the Order buffer is encountered.

When draining mbufs, the reorder buffer would return mbufs in the Ready buffer first and then from the Order buffer until a gap is found (mbufs that have not arrived yet).

15.3 Use Case: Packet Distributor

An application using the DPDK packet distributor could make use of the reorder library to transmit packets in the same order they were received.

A basic packet distributor use case would consist of a distributor with multiple workers cores. The processing of packets by the workers is not guaranteed to be in order, hence a reorder buffer can be used to order as many packets as possible.

In such a scenario, the distributor assigns a sequence number to mbufs before delivering them to the workers. As the workers finish processing the packets, the distributor inserts those mbufs into the reorder buffer and finally transmit drained mbufs.

NOTE: Currently the reorder buffer is not thread safe so the same thread is responsible for inserting and draining mbufs.

CHAPTER

SIXTEEN

IP FRAGMENTATION AND REASSEMBLY LIBRARY

The IP Fragmentation and Reassembly Library implements IPv4 and IPv6 packet fragmentation and reassembly.

16.1 Packet fragmentation

Packet fragmentation routines divide input packet into number of fragments. Both rte_ipv4_fragment_packet() and rte_ipv6_fragment_packet() functions assume that input mbuf data points to the start of the IP header of the packet (i.e. L2 header is already stripped out). To avoid copying of the actual packet's data zero-copy technique is used (rte_pktmbuf_attach). For each fragment two new mbufs are created:

- Direct mbuf – mbuf that will contain L3 header of the new fragment.
- Indirect mbuf – mbuf that is attached to the mbuf with the original packet. It's data field points to the start of the original packets data plus fragment offset.

Then L3 header is copied from the original mbuf into the 'direct' mbuf and updated to reflect new fragmented status. Note that for IPv4, header checksum is not recalculated and is set to zero.

Finally 'direct' and 'indirect' mbufs for each fragment are linked together via mbuf's next filed to compose a packet for the new fragment.

The caller has an ability to explicitly specify which mempools should be used to allocate 'direct' and 'indirect' mbufs from.

For more information about direct and indirect mbufs, refer to *Direct and Indirect Buffers*.

16.2 Packet reassembly

16.2.1 IP Fragment Table

Fragment table maintains information about already received fragments of the packet.

Each IP packet is uniquely identified by triple <Source IP address>, <Destination IP address>, <ID>.

Note that all update/lookup operations on Fragment Table are not thread safe. So if different execution contexts (threads/processes) will access the same table simultaneously, then some external syncing mechanism have to be provided.

Each table entry can hold information about packets consisting of up to RTE_LIBRTE_IP_FRAG_MAX (by default: 4) fragments.

Code example, that demonstrates creation of a new Fragment table:

```
frag_cycles = (rte_get_tsc_hz() + MS_PER_S - 1) / MS_PER_S * max_flow_ttl;
bucket_num = max_flow_num + max_flow_num / 4;
frag_tbl = rte_ip_frag_table_create(max_flow_num, bucket_entries, max_flow_num, frag_cycles, so
```

Internally Fragment table is a simple hash table. The basic idea is to use two hash functions and <bucket_entries> * associativity. This provides 2 * <bucket_entries> possible locations in the hash table for each key. When the collision occurs and all 2 * <bucket_entries> are occupied, instead of reinserting existing keys into alternative locations, ip_frag_tbl_add() just returns a failure.

Also, entries that resides in the table longer then <max_cycles> are considered as invalid, and could be removed/replaced by the new ones.

Note that reassembly demands a lot of mbuf's to be allocated. At any given time up to (2 * bucket_entries * RTE_LIBRTE_IP_FRAG_MAX * <maximum number of mbufs per packet>) can be stored inside Fragment Table waiting for remaining fragments.

16.2.2 Packet Reassembly

Fragmented packets processing and reassembly is done by the rte_ipv4_frag_reassemble_packet()/rte_ipv6_frag_reassemble_packet. Functions. They either return a pointer to valid mbuf that contains reassembled packet, or NULL (if the packet can't be reassembled for some reason).

These functions are responsible for:

1. Search the Fragment Table for entry with packet's <IPv4 Source Address, IPv4 Destination Address, Packet ID>.
2. If the entry is found, then check if that entry already timed-out. If yes, then free all previously received fragments, and remove information about them from the entry.
3. If no entry with such key is found, then try to create a new one by one of two ways:
   (a) Use as empty entry.
   (b) Delete a timed-out entry, free mbufs associated with it mbufs and store a new entry with specified key in it.
4. Update the entry with new fragment information and check if a packet can be reassembled (the packet's entry contains all fragments).
   (a) If yes, then, reassemble the packet, mark table's entry as empty and return the reassembled mbuf to the caller.
   (b) If no, then return a NULL to the caller.

If at any stage of packet processing an error is encountered (e.g: can't insert new entry into the Fragment Table, or invalid/timed-out fragment), then the function will free all associated with the packet fragments, mark the table entry as invalid and return NULL to the caller.

16.2.3 Debug logging and Statistics Collection

The RTE_LIBRTE_IP_FRAG_TBL_STAT config macro controls statistics collection for the Fragment Table. This macro is not enabled by default.

The RTE_LIBRTE_IP_FRAG_DEBUG controls debug logging of IP fragments processing and reassembling. This macro is disabled by default. Note that while logging contains a lot of detailed information, it slows down packet processing and might cause the loss of a lot of packets.

CHAPTER

SEVENTEEN

THE LIBRTE_PDUMP LIBRARY

The librte_pdump library provides a framework for packet capturing in DPDK. The library does the complete copy of the Rx and Tx mbufs to a new mempool and hence it slows down the performance of the applications, so it is recommended to use this library for debugging purposes.

The library provides the following APIs to initialize the packet capture framework, to enable or disable the packet capture, and to uninitialize it:

- rte_pdump_init(): This API initializes the packet capture framework.
- rte_pdump_enable(): This API enables the packet capture on a given port and queue. Note: The filter option in the API is a place holder for future enhancements.
- rte_pdump_enable_by_deviceid(): This API enables the packet capture on a given device id (vdev name or pci address) and queue. Note: The filter option in the API is a place holder for future enhancements.
- rte_pdump_disable(): This API disables the packet capture on a given port and queue.
- rte_pdump_disable_by_deviceid(): This API disables the packet capture on a given device id (vdev name or pci address) and queue.
- rte_pdump_uninit(): This API uninitializes the packet capture framework.
- rte_pdump_set_socket_dir(): This API sets the server and client socket paths. Note: This API is not thread-safe.

17.1 Operation

The librte_pdump library works on a client/server model. The server is responsible for enabling or disabling the packet capture and the clients are responsible for requesting the enabling or disabling of the packet capture.

The packet capture framework, as part of its initialization, creates the pthread and the server socket in the pthread. The application that calls the framework initialization will have the server socket created, either under the path that the application has passed or under the default path i.e. either /var/run/.dpdk for root user or ~/.dpdk for non root user.

Applications that request enabling or disabling of the packet capture will have the client socket created either under the path that the application has passed or under the default path i.e. either /var/run/.dpdk for root user or ~/.dpdk for not root user to send the requests to

the server. The server socket will listen for client requests for enabling or disabling the packet capture.

17.2 Implementation Details

The library API `rte_pdump_init()`, initializes the packet capture framework by creating the pthread and the server socket. The server socket in the pthread context will be listening to the client requests to enable or disable the packet capture.

The library APIs `rte_pdump_enable()` and `rte_pdump_enable_by_deviceid()` enables the packet capture. On each call to these APIs, the library creates a separate client socket, creates the "pdump enable" request and sends the request to the server. The server that is listening on the socket will take the request and enable the packet capture by registering the Ethernet RX and TX callbacks for the given port or device_id and queue combinations. Then the server will mirror the packets to the new mempool and enqueue them to the rte_ring that clients have passed to these APIs. The server also sends the response back to the client about the status of the request that was processed. After the response is received from the server, the client socket is closed.

The library APIs `rte_pdump_disable()` and `rte_pdump_disable_by_deviceid()` disables the packet capture. On each call to these APIs, the library creates a separate client socket, creates the "pdump disable" request and sends the request to the server. The server that is listening on the socket will take the request and disable the packet capture by removing the Ethernet RX and TX callbacks for the given port or device_id and queue combinations. The server also sends the response back to the client about the status of the request that was processed. After the response is received from the server, the client socket is closed.

The library API `rte_pdump_uninit()`, uninitializes the packet capture framework by closing the pthread and the server socket.

The library API `rte_pdump_set_socket_dir()`, sets the given path as either server socket path or client socket path based on the `type` argument of the API. If the given path is `NULL`, default path will be selected, i.e. either `/var/run/.dpdk` for root user or `~/.dpdk` for non root user. Clients also need to call this API to set their server socket path if the server socket path is different from default path.

17.3 Use Case: Packet Capturing

The DPDK `app/pdump` tool is developed based on this library to capture packets in DPDK. Users can use this as an example to develop their own packet capturing tools.

CHAPTER

EIGHTEEN

MULTI-PROCESS SUPPORT

In the DPDK, multi-process support is designed to allow a group of DPDK processes to work together in a simple transparent manner to perform packet processing, or other workloads. To support this functionality, a number of additions have been made to the core DPDK Environment Abstraction Layer (EAL).

The EAL has been modified to allow different types of DPDK processes to be spawned, each with different permissions on the hugepage memory used by the applications. For now, there are two types of process specified:

- primary processes, which can initialize and which have full permissions on shared memory
- secondary processes, which cannot initialize shared memory, but can attach to pre- initialized shared memory and create objects in it.

Standalone DPDK processes are primary processes, while secondary processes can only run alongside a primary process or after a primary process has already configured the hugepage shared memory for them.

To support these two process types, and other multi-process setups described later, two additional command-line parameters are available to the EAL:

- `--proc-type`: for specifying a given process instance as the primary or secondary DPDK instance
- `--file-prefix`: to allow processes that do not want to co-operate to have different memory regions

A number of example applications are provided that demonstrate how multiple DPDK processes can be used together. These are more fully documented in the “Multi- process Sample Application” chapter in the *DPDK Sample Application’s User Guide*.

18.1 Memory Sharing

The key element in getting a multi-process application working using the DPDK is to ensure that memory resources are properly shared among the processes making up the multi-process application. Once there are blocks of shared memory available that can be accessed by multiple processes, then issues such as inter-process communication (IPC) becomes much simpler.

On application start-up in a primary or standalone process, the DPDK records to memory-mapped files the details of the memory configuration it is using - hugepages in use, the virtual

addresses they are mapped at, the number of memory channels present, etc. When a secondary process is started, these files are read and the EAL recreates the same memory configuration in the secondary process so that all memory zones are shared between processes and all pointers to that memory are valid, and point to the same objects, in both processes.

Note: Refer to *Multi-process Limitations* for details of how Linux kernel Address-Space Layout Randomization (ASLR) can affect memory sharing.

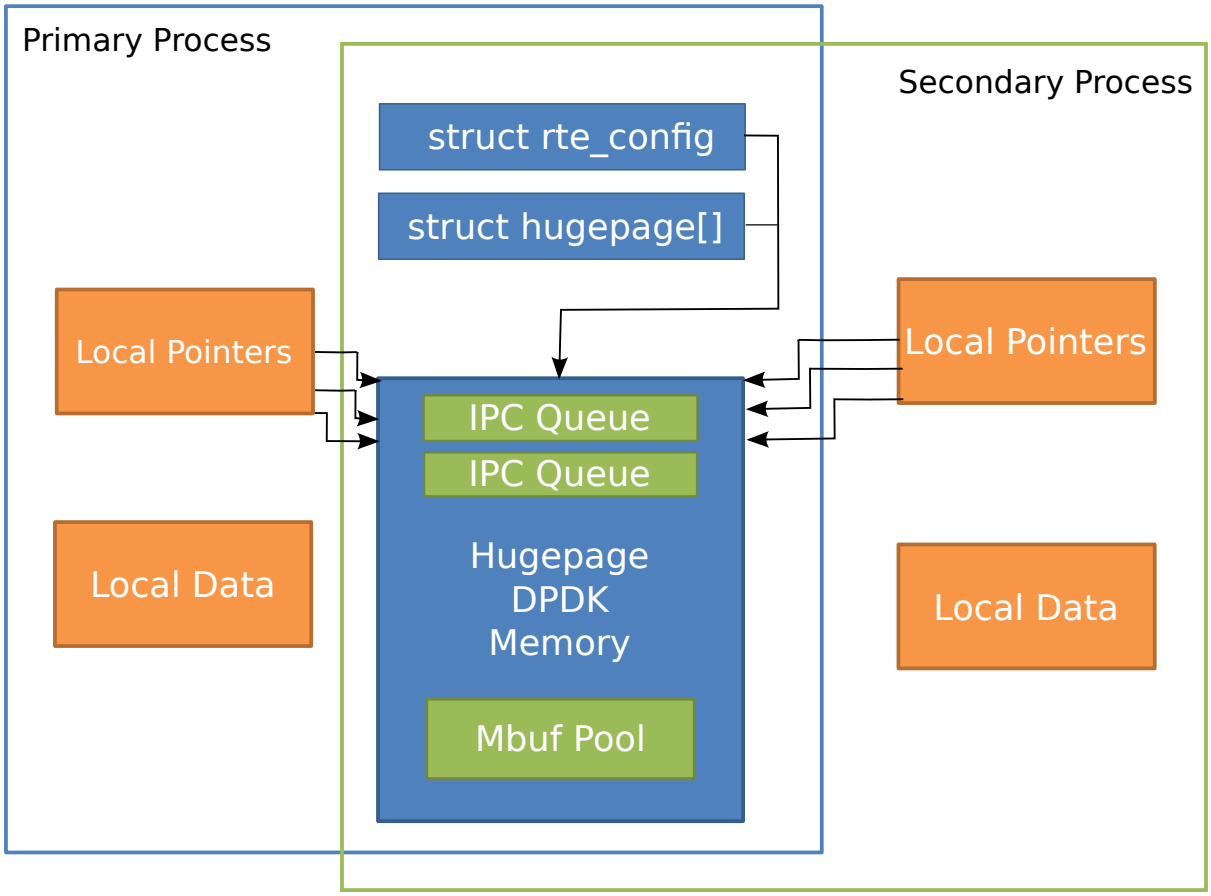


Fig. 18.1: Memory Sharing in the DPDK Multi-process Sample Application

The EAL also supports an auto-detection mode (set by EAL `--proc-type=auto` flag), whereby an DPDK process is started as a secondary instance if a primary instance is already running.

18.2 Deployment Models

18.2.1 Symmetric/Peer Processes

DPDK multi-process support can be used to create a set of peer processes where each process performs the same workload. This model is equivalent to having multiple threads each running the same main-loop function, as is done in most of the supplied DPDK sample applications. In this model, the first of the processes spawned should be spawned using the `--proc-type=primary` EAL flag, while all subsequent instances should be spawned using the `--proc-type=secondary` flag.

The simple_mp and symmetric_mp sample applications demonstrate this usage model. They are described in the "Multi-process Sample Application" chapter in the *DPDK Sample Application's User Guide*.

18.2.2 Asymmetric/Non-Peer Processes

An alternative deployment model that can be used for multi-process applications is to have a single primary process instance that acts as a load-balancer or server distributing received packets among worker or client threads, which are run as secondary processes. In this case, extensive use of rte_ring objects is made, which are located in shared hugepage memory.

The client_server_mp sample application shows this usage model. It is described in the "Multi-process Sample Application" chapter in the *DPDK Sample Application's User Guide*.

18.2.3 Running Multiple Independent DPDK Applications

In addition to the above scenarios involving multiple DPDK processes working together, it is possible to run multiple DPDK processes side-by-side, where those processes are all working independently. Support for this usage scenario is provided using the `--file-prefix` parameter to the EAL.

By default, the EAL creates hugepage files on each hugetlbfs filesystem using the rtemap_X filename, where X is in the range 0 to the maximum number of hugepages -1. Similarly, it creates shared configuration files, memory mapped in each process, using the /var/run/.rte_config filename, when run as root (or $HOME/.rte_config when run as a non-root user; if filesystem and device permissions are set up to allow this). The rte part of the filenames of each of the above is configurable using the file-prefix parameter.

In addition to specifying the file-prefix parameter, any DPDK applications that are to be run side-by-side must explicitly limit their memory use. This is done by passing the -m flag to each process to specify how much hugepage memory, in megabytes, each process can use (or passing `--socket-mem` to specify how much hugepage memory on each socket each process can use).

Note: Independent DPDK instances running side-by-side on a single machine cannot share any network ports. Any network ports being used by one process should be blacklisted in every other process.

18.2.4 Running Multiple Independent Groups of DPDK Applications

In the same way that it is possible to run independent DPDK applications side- by-side on a single system, this can be trivially extended to multi-process groups of DPDK applications running side-by-side. In this case, the secondary processes must use the same `--file-prefix` parameter as the primary process whose shared memory they are connecting to.

Note: All restrictions and issues with multiple independent DPDK processes running side-by-side apply in this usage scenario also.

18.3 Multi-process Limitations

There are a number of limitations to what can be done when running DPDK multi-process applications. Some of these are documented below:

- The multi-process feature requires that the exact same hugepage memory mappings be present in all applications. The Linux security feature - Address-Space Layout Randomization (ASLR) can interfere with this mapping, so it may be necessary to disable this feature in order to reliably run multi-process applications.

Warning: Disabling Address-Space Layout Randomization (ASLR) may have security implications, so it is recommended that it be disabled only when absolutely necessary, and only when the implications of this change have been understood.

- All DPDK processes running as a single application and using shared memory must have distinct coremask arguments. It is not possible to have a primary and secondary instance, or two secondary instances, using any of the same logical cores. Attempting to do so can cause corruption of memory pool caches, among other issues.
- The delivery of interrupts, such as Ethernet* device link status interrupts, do not work in secondary processes. All interrupts are triggered inside the primary process only. Any application needing interrupt notification in multiple processes should provide its own mechanism to transfer the interrupt information from the primary process to any secondary process that needs the information.
- The use of function pointers between multiple processes running based of different compiled binaries is not supported, since the location of a given function in one process may be different to its location in a second. This prevents the librte_hash library from behaving properly as in a multi-threaded instance, since it uses a pointer to the hash function internally.

To work around this issue, it is recommended that multi-process applications perform the hash calculations by directly calling the hashing function from the code and then using the rte_hash_add_with_hash()/rte_hash_lookup_with_hash() functions instead of the functions which do the hashing internally, such as rte_hash_add()/rte_hash_lookup().

- Depending upon the hardware in use, and the number of DPDK processes used, it may not be possible to have HPET timers available in each DPDK instance. The minimum number of HPET comparators available to Linux* userspace can be just a single comparator, which means that only the first, primary DPDK process instance can open and mmap /dev/hpet. If the number of required DPDK processes exceeds that of the number of available HPET comparators, the TSC (which is the default timer in this release) must be used as a time source across all processes instead of the HPET.

CHAPTER

NINETEEN

KERNEL NIC INTERFACE

The DPDK Kernel NIC Interface (KNI) allows userspace applications access to the Linux* control plane.

The benefits of using the DPDK KNI are:

- Faster than existing Linux TUN/TAP interfaces (by eliminating system calls and copy_to_user()/copy_from_user() operations.
- Allows management of DPDK ports using standard Linux net tools such as ethtool, ifconfig and tcpdump.
- Allows an interface with the kernel network stack.

The components of an application using the DPDK Kernel NIC Interface are shown in Fig. 19.1.

19.1 The DPDK KNI Kernel Module

The KNI kernel loadable module provides support for two types of devices:

- A Miscellaneous device (/dev/kni) that:
  - Creates net devices (via ioctl calls).
  - Maintains a kernel thread context shared by all KNI instances (simulating the RX side of the net driver).
  - For single kernel thread mode, maintains a kernel thread context shared by all KNI instances (simulating the RX side of the net driver).
  - For multiple kernel thread mode, maintains a kernel thread context for each KNI instance (simulating the RX side of the new driver).
- Net device:
  - Net functionality provided by implementing several operations such as netdev_ops, header_ops, ethtool_ops that are defined by struct net_device, including support for DPDK mbufs and FIFOs.
  - The interface name is provided from userspace.
  - The MAC address can be the real NIC MAC address or random.

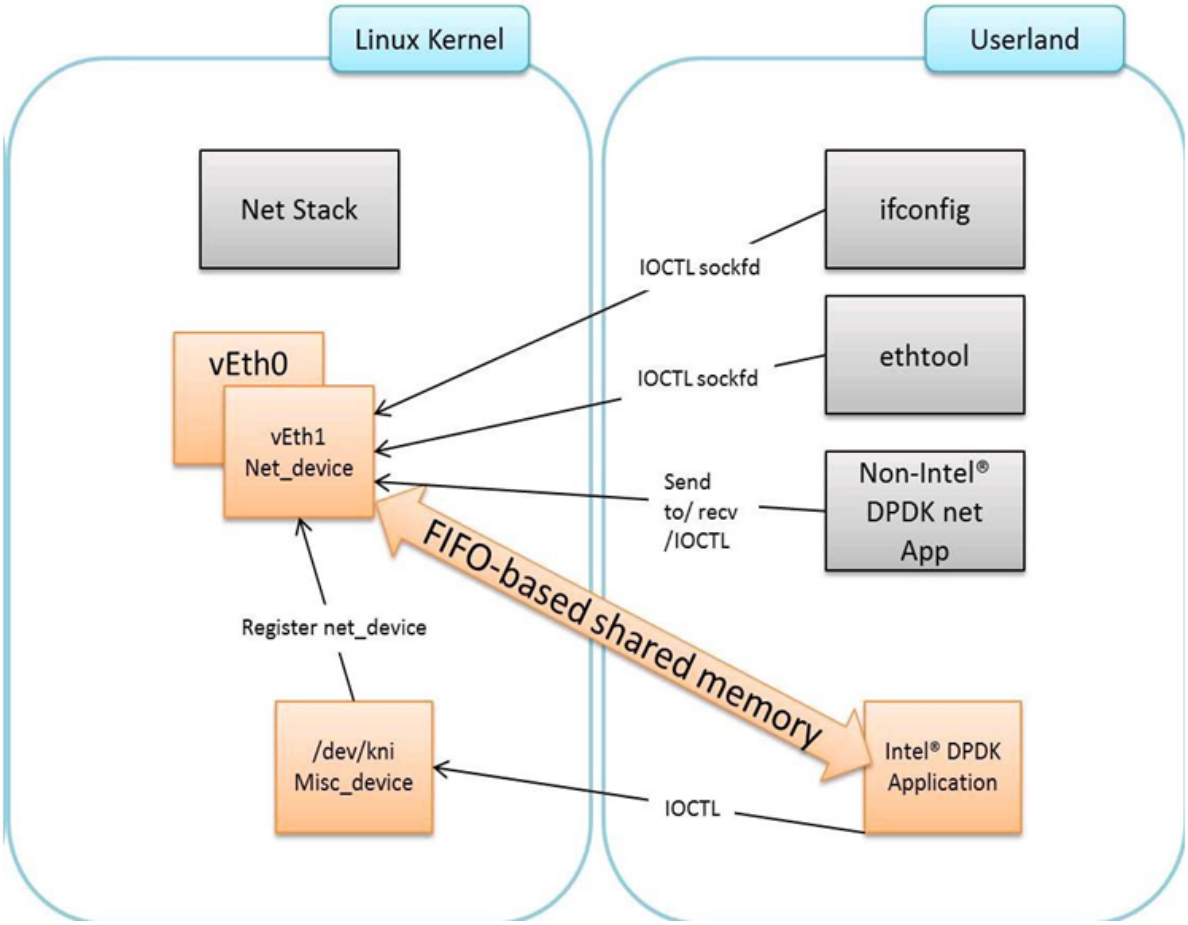


Fig. 19.1: Components of a DPDK KNI Application

19.2 KNI Creation and Deletion

The KNI interfaces are created by a DPDK application dynamically. The interface name and FIFO details are provided by the application through an ioctl call using the rte_kni_device_info struct which contains:

- The interface name.
- Physical addresses of the corresponding memzones for the relevant FIFOs.
- Mbuf mempool details, both physical and virtual (to calculate the offset for mbuf pointers).
- PCI information.
- Core affinity.

Refer to rte_kni_common.h in the DPDK source code for more details.

The physical addresses will be re-mapped into the kernel address space and stored in separate KNI contexts.

The affinity of kernel RX thread (both single and multi-threaded modes) is controlled by force_bind and core_id config parameters.

The KNI interfaces can be deleted by a DPDK application dynamically after being created. Furthermore, all those KNI interfaces not deleted will be deleted on the release operation of the miscellaneous device (when the DPDK application is closed).

19.3 DPDK mbuf Flow

To minimize the amount of DPDK code running in kernel space, the mbuf mempool is managed in userspace only. The kernel module will be aware of mbufs, but all mbuf allocation and free operations will be handled by the DPDK application only.

Fig. 19.2 shows a typical scenario with packets sent in both directions.

19.4 Use Case: Ingress

On the DPDK RX side, the mbuf is allocated by the PMD in the RX thread context. This thread will enqueue the mbuf in the rx_q FIFO. The KNI thread will poll all KNI active devices for the rx_q. If an mbuf is dequeued, it will be converted to a sk_buff and sent to the net stack via netif_rx(). The dequeued mbuf must be freed, so the same pointer is sent back in the free_q FIFO.

The RX thread, in the same main loop, polls this FIFO and frees the mbuf after dequeuing it.

19.5 Use Case: Egress

For packet egress the DPDK application must first enqueue several mbufs to create an mbuf cache on the kernel side.

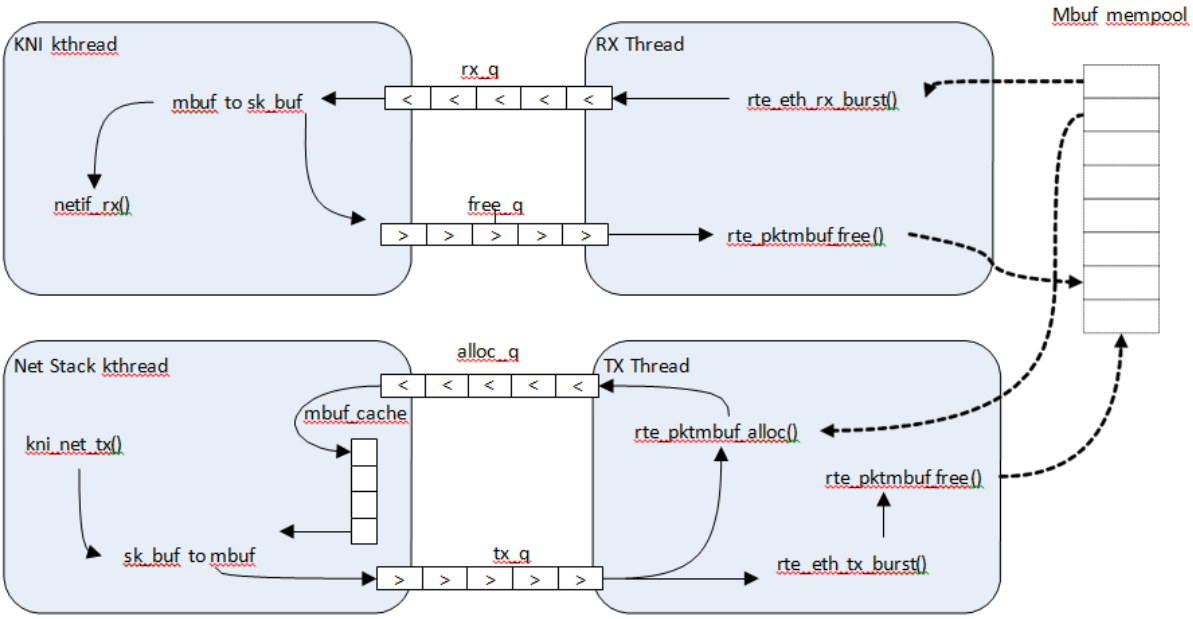


Fig. 19.2: Packet Flow via mbufs in the DPDK KNI

The packet is received from the Linux net stack, by calling the kni_net_tx() callback. The mbuf is dequeued (without waiting due the cache) and filled with data from sk_buff. The sk_buff is then freed and the mbuf sent in the tx_q FIFO.

The DPDK TX thread dequeues the mbuf and sends it to the PMD (via rte_eth_tx_burst()). It then puts the mbuf back in the cache.

19.6 Ethtool

Ethtool is a Linux-specific tool with corresponding support in the kernel where each net device must register its own callbacks for the supported operations. The current implementation uses the igb/ixgbe modified Linux drivers for ethtool support. Ethtool is not supported in i40e and VMs (VF or EM devices).

19.7 Link state and MTU change

Link state and MTU change are network interface specific operations usually done via ifconfig. The request is initiated from the kernel side (in the context of the ifconfig process) and handled by the user space DPDK application. The application polls the request, calls the application handler and returns the response back into the kernel space.

The application handlers can be registered upon interface creation or explicitly registered/unregistered in runtime. This provides flexibility in multiprocess scenarios (where the KNI is created in the primary process but the callbacks are handled in the secondary one). The constraint is that a single process can register and handle the requests.

19.8 KNI Working as a Kernel vHost Backend

vHost is a kernel module usually working as the backend of virtio (a para- virtualization driver framework) to accelerate the traffic from the guest to the host. The DPDK Kernel NIC interface provides the ability to hookup vHost traffic into userspace DPDK application. Together with the DPDK PMD virtio, it significantly improves the throughput between guest and host. In the scenario where DPDK is running as fast path in the host, kni-vhost is an efficient path for the traffic.

19.8.1 Overview

vHost-net has three kinds of real backend implementations. They are: 1) tap, 2) macvtap and 3) RAW socket. The main idea behind kni-vhost is making the KNI work as a RAW socket, attaching it as the backend instance of vHost-net. It is using the existing interface with vHost-net, so it does not require any kernel hacking, and is fully-compatible with the kernel vhost module. As vHost is still taking responsibility for communicating with the front-end virtio, it naturally supports both legacy virtio -net and the DPDK PMD virtio. There is a little penalty that comes from the non-polling mode of vhost. However, it scales throughput well when using KNI in multi-thread mode.

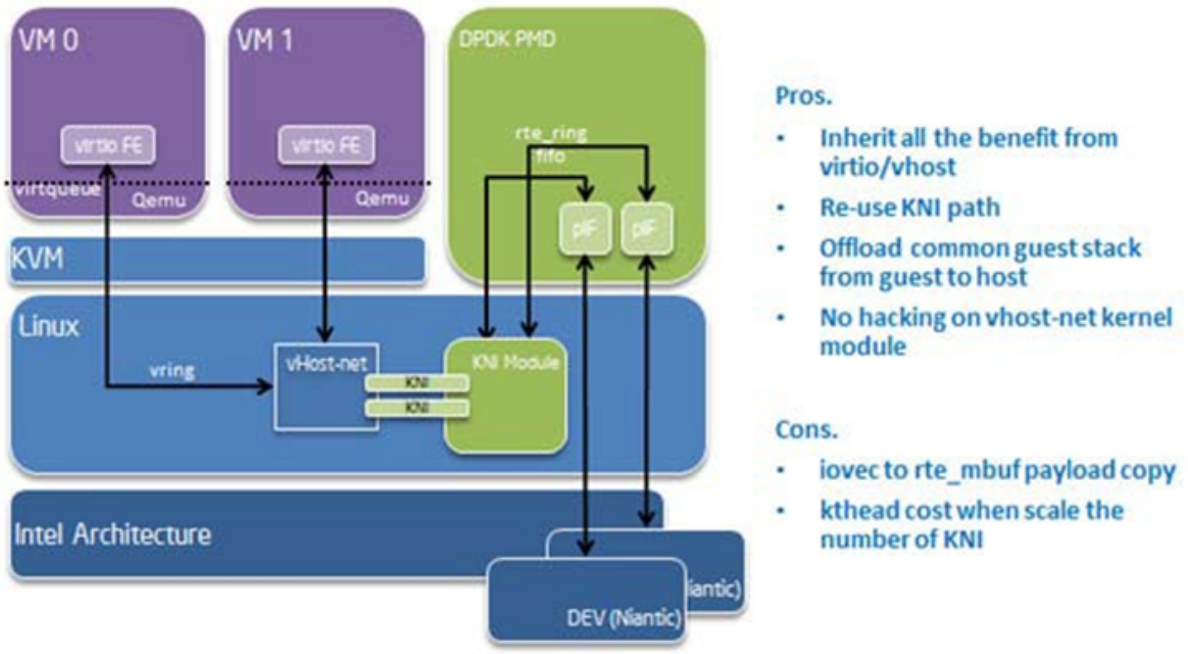


Fig. 19.3: vHost-net Architecture Overview

19.8.2 Packet Flow

There is only a minor difference from the original KNI traffic flows. On transmit side, vhost kthread calls the RAW socket's ops sendmsg and it puts the packets into the KNI transmit FIFO. On the receive side, the kni kthread gets packets from the KNI receive FIFO, puts them into the queue of the raw socket, and wakes up the task in vhost kthread to begin receiving. All the packet copying, irrespective of whether it is on the transmit or receive side, happens in the

context of vhost kthread. Every vhost-net device is exposed to a front end virtio device in the guest.

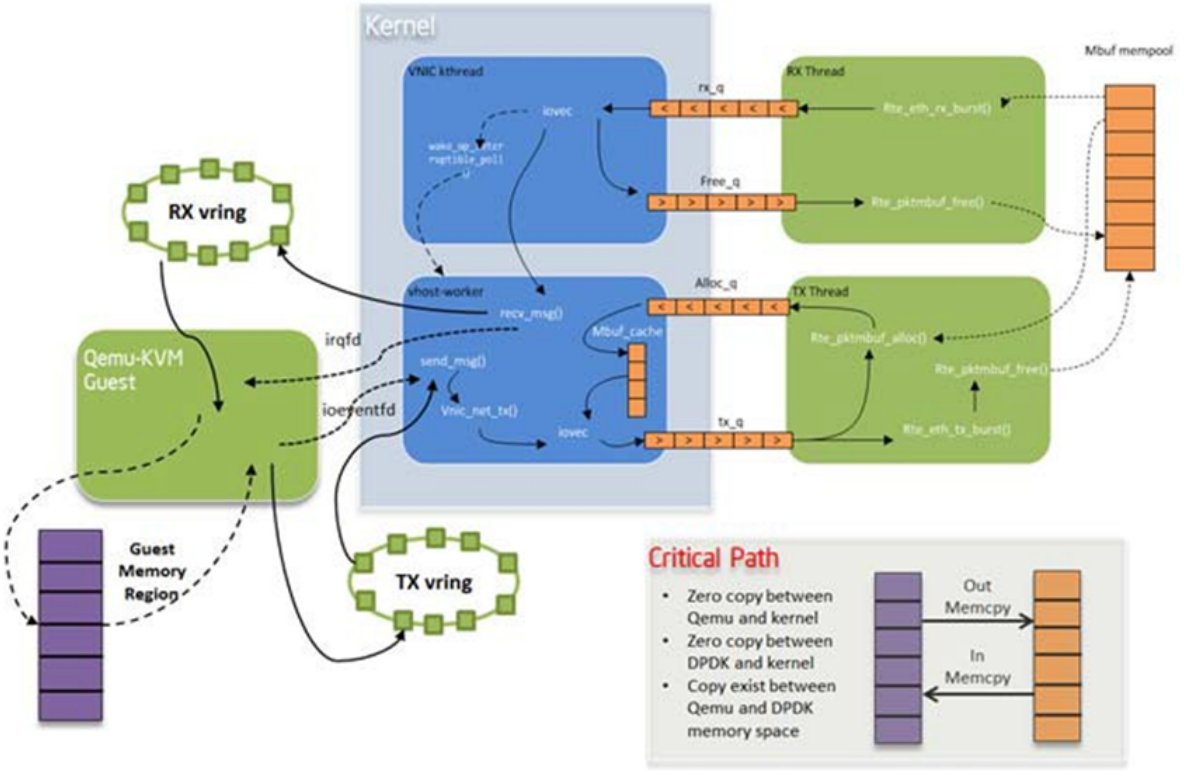


Fig. 19.4: KNI Traffic Flow

19.8.3 Sample Usage

Before starting to use KNI as the backend of vhost, the CONFIG_RTE_KNI_VHOST configuration option must be turned on. Otherwise, by default, KNI will not enable its backend support capability.

Of course, as a prerequisite, the vhost/vhost-net kernel CONFIG should be chosen before compiling the kernel.

1. Compile the DPDK and insert uio_pci_generic/igb_uio kernel modules as normal.

2. Insert the KNI kernel module:

   ```
   insmod ./rte_kni.ko
   ```

   If using KNI in multi-thread mode, use the following command line:

   ```
   insmod ./rte_kni.ko kthread_mode=multiple
   ```

3. Running the KNI sample application:

   ```
   examples/kni/build/app/kni -c -0xf0 -n 4 -- -p 0x3 -P --config="(0,4,6),(1,5,7)"
   ```

   This command runs the kni sample application with two physical ports. Each port pins two forwarding cores (ingress/egress) in user space.

4. Assign a raw socket to vhost-net during qemu-kvm startup. The DPDK does not provide a script to do this since it is easy for the user to customize. The following shows the key steps to launch qemu-kvm with kni-vhost:

```
#!/bin/bash
echo 1 > /sys/class/net/vEth0/sock_en
fd=`cat /sys/class/net/vEth0/sock_fd`
qemu-kvm \
-name vm1 -cpu host -m 2048 -smp 1 -hda /opt/vm-fc16.img \
-netdev tap,fd=$fd,id=hostnet1,vhost=on \
-device virti-net-pci,netdev=hostnet1,id=net1,bus=pci.0,addr=0x4
```

It is simple to enable raw socket using sysfs sock_en and get raw socket fd using sock_fd under the KNI device node.

Then, using the qemu-kvm command with the -netdev option to assign such raw socket fd as vhost's backend.

Note: The key word tap must exist as qemu-kvm now only supports vhost with a tap backend, so here we cheat qemu-kvm by an existing fd.

19.8.4 Compatibility Configure Option

There is a CONFIG_RTE_KNI_VHOST_VNET_HDR_EN configuration option in DPDK configuration file. By default, it set to n, which means do not turn on the virtio net header, which is used to support additional features (such as, csum offload, vlan offload, generic-segmentation and so on), since the kni-vhost does not yet support those features.

Even if the option is turned on, kni-vhost will ignore the information that the header contains. When working with legacy virtio on the guest, it is better to turn off unsupported offload features using ethtool -K. Otherwise, there may be problems such as an incorrect L4 checksum error.

CHAPTER

TWENTY

THREAD SAFETY OF DPDK FUNCTIONS

The DPDK is comprised of several libraries. Some of the functions in these libraries can be safely called from multiple threads simultaneously, while others cannot. This section allows the developer to take these issues into account when building their own application.

The run-time environment of the DPDK is typically a single thread per logical core. In some cases, it is not only multi-threaded, but multi-process. Typically, it is best to avoid sharing data structures between threads and/or processes where possible. Where this is not possible, then the execution blocks must access the data in a thread- safe manner. Mechanisms such as atomics or locking can be used that will allow execution blocks to operate serially. However, this can have an effect on the performance of the application.

20.1 Fast-Path APIs

Applications operating in the data plane are performance sensitive but certain functions within those libraries may not be safe to call from multiple threads simultaneously. The hash, LPM and mempool libraries and RX/TX in the PMD are examples of this.

The hash and LPM libraries are, by design, thread unsafe in order to maintain performance. However, if required the developer can add layers on top of these libraries to provide thread safety. Locking is not needed in all situations, and in both the hash and LPM libraries, lookups of values can be performed in parallel in multiple threads. Adding, removing or modifying values, however, cannot be done in multiple threads without using locking when a single hash or LPM table is accessed. Another alternative to locking would be to create multiple instances of these tables allowing each thread its own copy.

The RX and TX of the PMD are the most critical aspects of a DPDK application and it is recommended that no locking be used as it will impact performance. Note, however, that these functions can safely be used from multiple threads when each thread is performing I/O on a different NIC queue. If multiple threads are to use the same hardware queue on the same NIC port, then locking, or some other form of mutual exclusion, is necessary.

The ring library is based on a lockless ring-buffer algorithm that maintains its original design for thread safety. Moreover, it provides high performance for either multi- or single-consumer/producer enqueue/dequeue operations. The mempool library is based on the DPDK lockless ring library and therefore is also multi-thread safe.

20.2 Performance Insensitive API

Outside of the performance sensitive areas described in Section 25.1, the DPDK provides a thread-safe API for most other libraries. For example, malloc and memzone functions are safe for use in multi-threaded and multi-process environments.

The setup and configuration of the PMD is not performance sensitive, but is not thread safe either. It is possible that the multiple read/writes during PMD setup and configuration could be corrupted in a multi-thread environment. Since this is not performance sensitive, the developer can choose to add their own layer to provide thread-safe setup and configuration. It is expected that, in most applications, the initial configuration of the network ports would be done by a single thread at startup.

20.3 Library Initialization

It is recommended that DPDK libraries are initialized in the main thread at application startup rather than subsequently in the forwarding threads. However, the DPDK performs checks to ensure that libraries are only initialized once. If initialization is attempted more than once, an error is returned.

In the multi-process case, the configuration information of shared memory will only be initialized by the master process. Thereafter, both master and secondary processes can allocate/release any objects of memory that finally rely on rte_malloc or memzones.

20.4 Interrupt Thread

The DPDK works almost entirely in Linux user space in polling mode. For certain infrequent operations, such as receiving a PMD link status change notification, callbacks may be called in an additional thread outside the main DPDK processing threads. These function callbacks should avoid manipulating DPDK objects that are also managed by the normal DPDK threads, and if they need to do so, it is up to the application to provide the appropriate locking or mutual exclusion restrictions around those objects.

CHAPTER

TWENTYONE

QUALITY OF SERVICE (QOS) FRAMEWORK

This chapter describes the DPDK Quality of Service (QoS) framework.

21.1 Packet Pipeline with QoS Support

An example of a complex packet processing pipeline with QoS support is shown in the following figure.

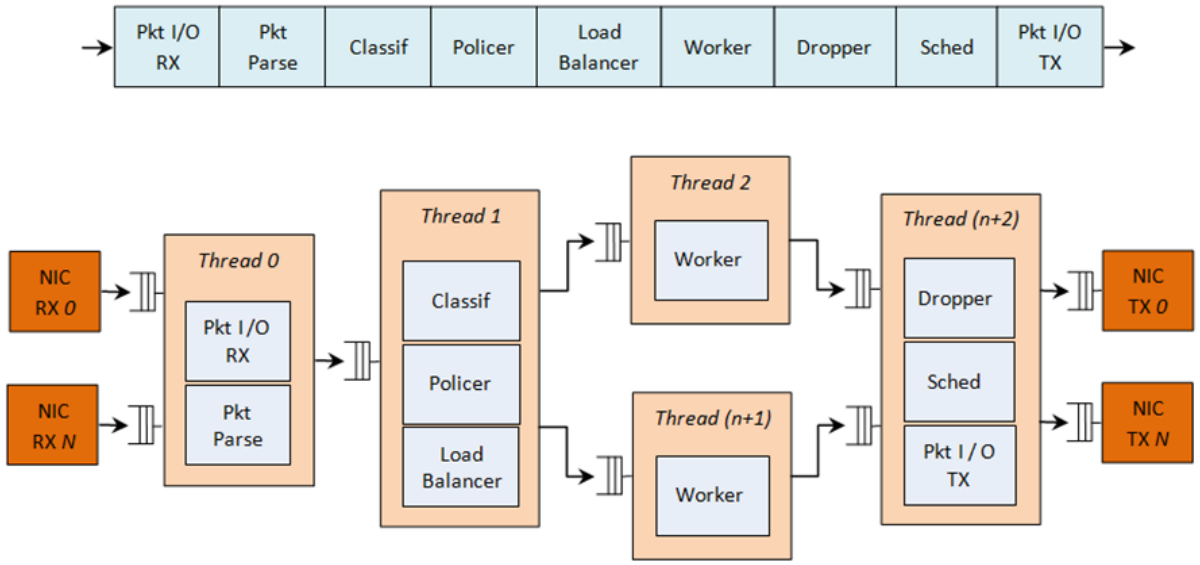


Fig. 21.1: Complex Packet Processing Pipeline with QoS Support

This pipeline can be built using reusable DPDK software libraries. The main blocks implementing QoS in this pipeline are: the policer, the dropper and the scheduler. A functional description of each block is provided in the following table.

Table 21.1: Packet Processing Pipeline Implementing QoS

#	Block	Functional Description
1	Packet I/O RX & TX	Packet reception/ transmission from/to multiple NIC ports. Poll mode drivers (PMDs) for Intel 1 GbE/10 GbE NICs.
2	Packet parser	Identify the protocol stack of the input packet. Check the integrity of the packet headers.
3	Flow classification	Map the input packet to one of the known traffic flows. Exact match table lookup using configurable hash function (jhash, CRC and so on) and bucket logic to handle collisions.
4	Policer	Packet metering using srTCM (RFC 2697) or trTCM (RFC2698) algorithms.
5	Load Balancer	Distribute the input packets to the application workers. Provide uniform load to each worker. Preserve the affinity of traffic flows to workers and the packet order within each flow.
6	Worker threads	Placeholders for the customer specific application workload (for example, IP stack and so on).
7	Dropper	Congestion management using the Random Early Detection (RED) algorithm (specified by the Sally Floyd - Van Jacobson paper) or Weighted RED (WRED). Drop packets based on the current scheduler queue load level and packet priority. When congestion is experienced, lower priority packets are dropped first.
8	Hierarchical Scheduler	5-level hierarchical scheduler (levels are: output port, subport, pipe, traffic class and queue) with thousands (typically 64K) leaf nodes (queues). Implements traffic shaping (for subport and pipe levels), strict priority (for traffic class level) and Weighted Round Robin (WRR) (for queues within each pipe traffic class).

The infrastructure blocks used throughout the packet processing pipeline are listed in the following table.

Table 21.2: Infrastructure Blocks Used by the Packet Processing Pipeline

#	Block	Functional Description
1	Buffer manager	Support for global buffer pools and private per-thread buffer caches.
2	Queue manager	Support for message passing between pipeline blocks.
3	Power saving	Support for power saving during low activity periods.

The mapping of pipeline blocks to CPU cores is configurable based on the performance level required by each specific application and the set of features enabled for each block. Some blocks might consume more than one CPU core (with each CPU core running a different instance of the same block on different input packets), while several other blocks could be mapped to the same CPU core.

21.2 Hierarchical Scheduler

The hierarchical scheduler block, when present, usually sits on the TX side just before the transmission stage. Its purpose is to prioritize the transmission of packets from different users and different traffic classes according to the policy specified by the Service Level Agreements (SLAs) of each network node.

21.2.1 Overview

The hierarchical scheduler block is similar to the traffic manager block used by network processors that typically implement per flow (or per group of flows) packet queuing and scheduling. It typically acts like a buffer that is able to temporarily store a large number of packets just before their transmission (enqueue operation); as the NIC TX is requesting more packets for transmission, these packets are later on removed and handed over to the NIC TX with the packet selection logic observing the predefined SLAs (dequeue operation).

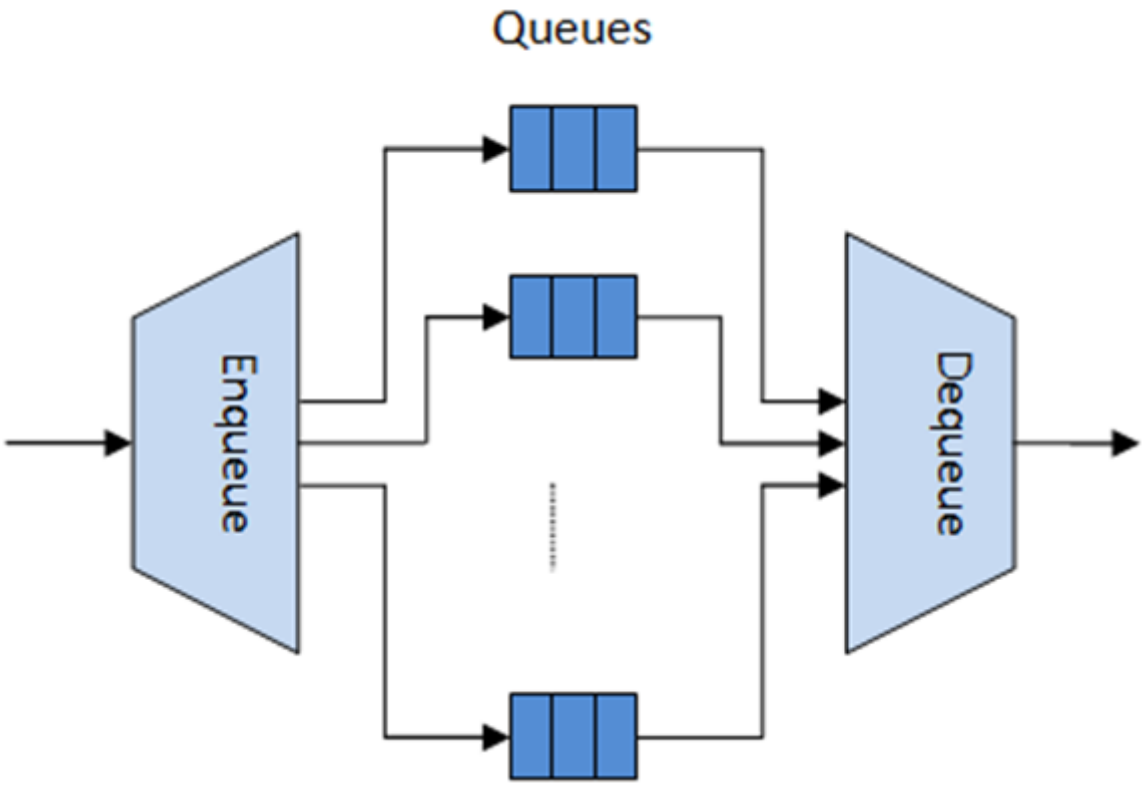


Fig. 21.2: Hierarchical Scheduler Block Internal Diagram

The hierarchical scheduler is optimized for a large number of packet queues. When only a small number of queues are needed, message passing queues should be used instead of this block. See *Worst Case Scenarios for Performance* for a more detailed discussion.

21.2.2 Scheduling Hierarchy

The scheduling hierarchy is shown in Fig. 21.3. The first level of the hierarchy is the Ethernet TX port 1/10/40 GbE, with subsequent hierarchy levels defined as subport, pipe, traffic class and queue.

Typically, each subport represents a predefined group of users, while each pipe represents an individual user/subscriber. Each traffic class is the representation of a different traffic type with specific loss rate, delay and jitter requirements, such as voice, video or data transfers. Each queue hosts packets from one or multiple connections of the same type belonging to the same user.

The functionality of each hierarchical level is detailed in the following table.

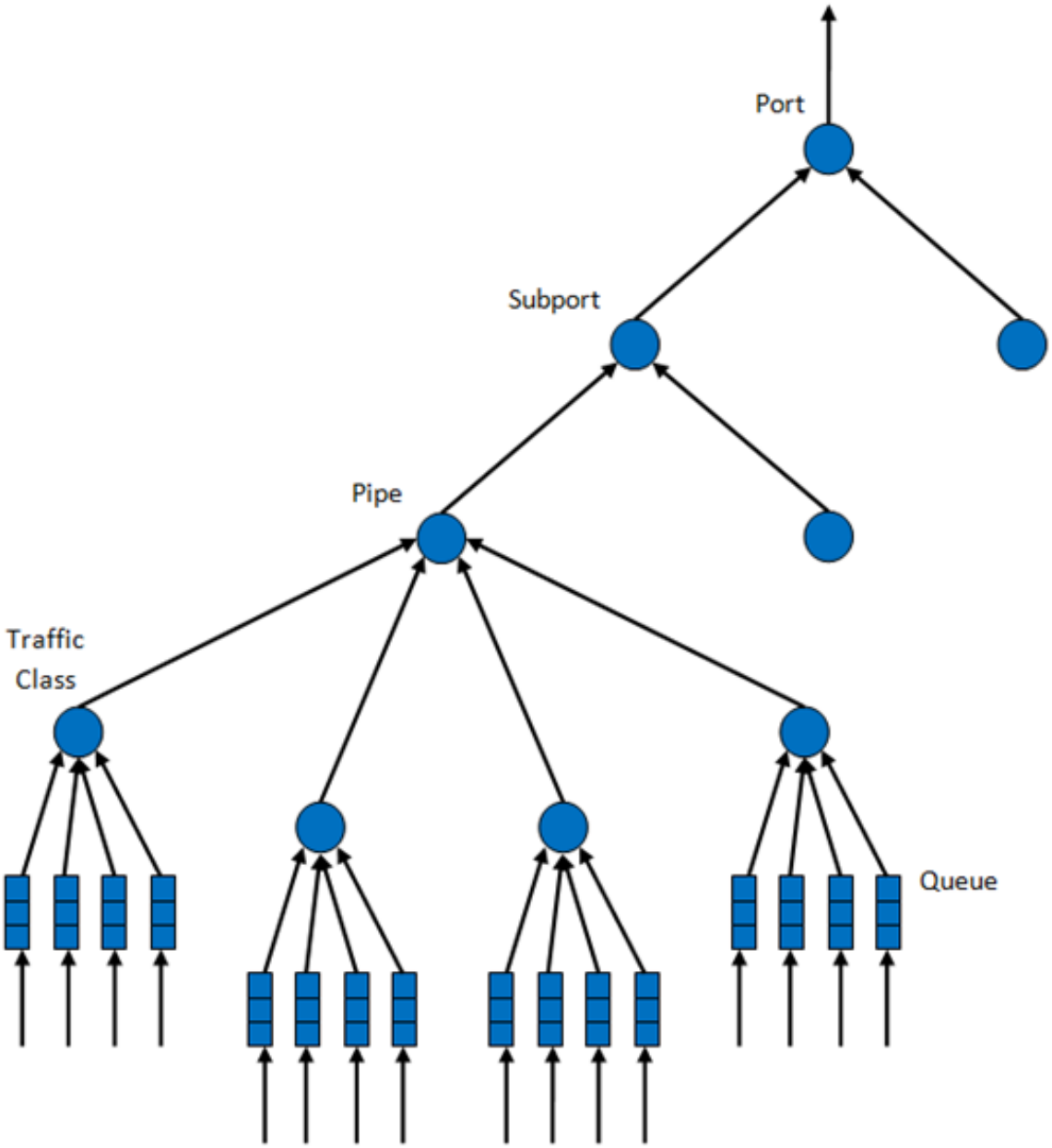


Fig. 21.3: Scheduling Hierarchy per Port

Table 21.3: Port Scheduling Hierarchy

#	Level	Siblings per Parent	Functional Description
1	Port	•	1. Output Ethernet port 1/10/40 GbE. 2. Multiple ports are scheduled in round robin order with all ports having equal priority.
2	Subport	Configurable (default: 8)	1. Traffic shaping using token bucket algorithm (one token bucket per subport). 2. Upper limit enforced per Traffic Class (TC) at the subport level. 3. Lower priority TCs able to reuse subport bandwidth currently unused by higher priority TCs.
3	Pipe	Configurable (default: 4K)	1. Traffic shaping using the token bucket algorithm (one token bucket per pipe.
4	Traffic Class (TC)	4	1. TCs of the same pipe handled in strict priority order. 2. Upper limit enforced per TC at the pipe level. 3. Lower priority TCs able to reuse pipe bandwidth currently unused by higher priority TCs. 4. When subport

21.2.3 Application Programming Interface (API)

Port Scheduler Configuration API

The rte_sched.h file contains configuration functions for port, subport and pipe.

Port Scheduler Enqueue API

The port scheduler enqueue API is very similar to the API of the DPDK PMD TX function.

```
int rte_sched_port_enqueue(struct rte_sched_port *port, struct rte_mbuf **pkts, uint32_t n_pkts
```

Port Scheduler Dequeue API

The port scheduler dequeue API is very similar to the API of the DPDK PMD RX function.

```
int rte_sched_port_dequeue(struct rte_sched_port *port, struct rte_mbuf **pkts, uint32_t n_pkts
```

Usage Example

```
/* File "application.c" */

#define N_PKTS_RX   64
#define N_PKTS_TX   48
#define NIC_RX_PORT 0
#define NIC_RX_QUEUE 0
#define NIC_TX_PORT 1
#define NIC_TX_QUEUE 0

struct rte_sched_port *port = NULL;
struct rte_mbuf *pkts_rx[N_PKTS_RX], *pkts_tx[N_PKTS_TX];
uint32_t n_pkts_rx, n_pkts_tx;

/* Initialization */

<initialization code>

/* Runtime */
while (1) {
    /* Read packets from NIC RX queue */

    n_pkts_rx = rte_eth_rx_burst(NIC_RX_PORT, NIC_RX_QUEUE, pkts_rx, N_PKTS_RX);

    /* Hierarchical scheduler enqueue */

    rte_sched_port_enqueue(port, pkts_rx, n_pkts_rx);

    /* Hierarchical scheduler dequeue */

    n_pkts_tx = rte_sched_port_dequeue(port, pkts_tx, N_PKTS_TX);

    /* Write packets to NIC TX queue */

    rte_eth_tx_burst(NIC_TX_PORT, NIC_TX_QUEUE, pkts_tx, n_pkts_tx);
}
```

21.2.4 Implementation

Internal Data Structures per Port

A schematic of the internal data structures in shown in with details in.

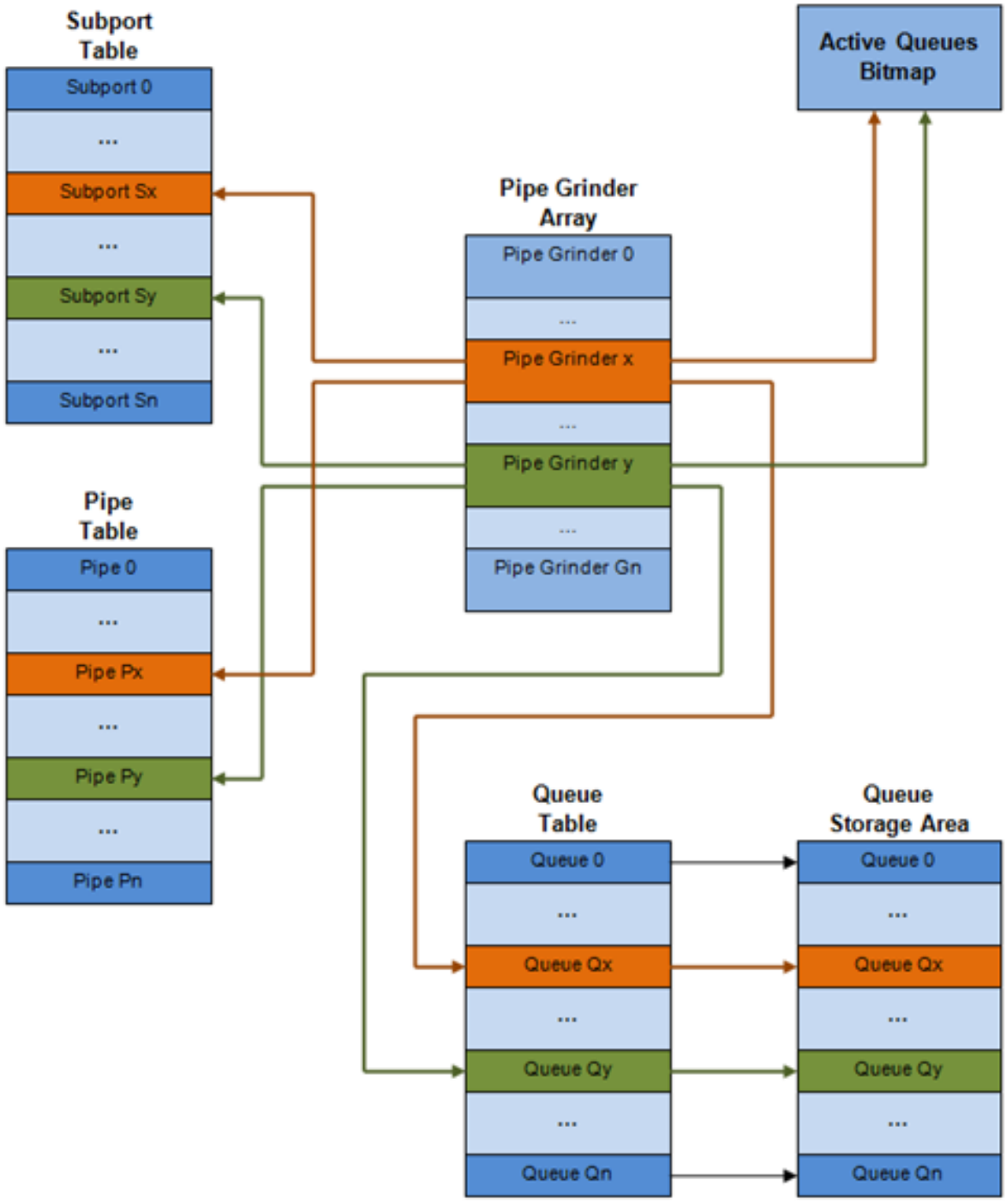


Fig. 21.4: Internal Data Structures per Port

Table 21.4: Scheduler Internal Data Structures per Port

#	Data structure	Size (bytes)	# per port	Access type		Description
				Enq	Deq	
1	Subport table entry	64	# subports per port	•	Rd, Wr	Persistent subport data (credits, etc).
2	Pipe table entry	64	# pipes per port	•	Rd, Wr	Persistent data for pipe, its TCs and its queues (credits, etc) that is updated during run-time. The pipe configuration parameters do not change during run-time. The same pipe configuration parameters are shared by multiple pipes, therefore they are not part of pipe table entry.
3	Queue table entry	4	#queues per port	Rd, Wr	Rd, Wr	Persistent queue data (read and write pointers). The queue size is the same per TC for all queues, allowing the queue base address to be computed using a fast formula, so these two parameters are not part of queue table entry. The queue table entries

Multicore Scaling Strategy

The multicore scaling strategy is:

1. Running different physical ports on different threads. The enqueue and dequeue of the same port are run by the same thread.
2. Splitting the same physical port to different threads by running different sets of subports of the same physical port (virtual ports) on different threads. Similarly, a subport can be split into multiple subports that are each run by a different thread. The enqueue and dequeue of the same port are run by the same thread. This is only required if, for performance reasons, it is not possible to handle a full port with a single core.

Enqueue and Dequeue for the Same Output Port

Running enqueue and dequeue operations for the same output port from different cores is likely to cause significant impact on scheduler's performance and it is therefore not recommended.

The port enqueue and dequeue operations share access to the following data structures:

1. Packet descriptors
2. Queue table
3. Queue storage area
4. Bitmap of active queues

The expected drop in performance is due to:

1. Need to make the queue and bitmap operations thread safe, which requires either using locking primitives for access serialization (for example, spinlocks/ semaphores) or using atomic primitives for lockless access (for example, Test and Set, Compare And Swap, an so on). The impact is much higher in the former case.
2. Ping-pong of cache lines storing the shared data structures between the cache hierarchies of the two cores (done transparently by the MESI protocol cache coherency CPU hardware).

Therefore, the scheduler enqueue and dequeue operations have to be run from the same thread, which allows the queues and the bitmap operations to be non-thread safe and keeps the scheduler data structures internal to the same core.

Performance Scaling

Scaling up the number of NIC ports simply requires a proportional increase in the number of CPU cores to be used for traffic scheduling.

Enqueue Pipeline

The sequence of steps per packet:

1. *Access* the mbuf to read the data fields required to identify the destination queue for the packet. These fields are: port, subport, traffic class and queue within traffic class, and are typically set by the classification stage.

2. *Access* the queue structure to identify the write location in the queue array. If the queue is full, then the packet is discarded.
3. *Access* the queue array location to store the packet (i.e. write the mbuf pointer).

It should be noted the strong data dependency between these steps, as steps 2 and 3 cannot start before the result from steps 1 and 2 becomes available, which prevents the processor out of order execution engine to provide any significant performance optimizations.

Given the high rate of input packets and the large amount of queues, it is expected that the data structures accessed to enqueue the current packet are not present in the L1 or L2 data cache of the current core, thus the above 3 memory accesses would result (on average) in L1 and L2 data cache misses. A number of 3 L1/L2 cache misses per packet is not acceptable for performance reasons.

The workaround is to prefetch the required data structures in advance. The prefetch operation has an execution latency during which the processor should not attempt to access the data structure currently under prefetch, so the processor should execute other work. The only other work available is to execute different stages of the enqueue sequence of operations on other input packets, thus resulting in a pipelined implementation for the enqueue operation.

Fig. 21.5 illustrates a pipelined implementation for the enqueue operation with 4 pipeline stages and each stage executing 2 different input packets. No input packet can be part of more than one pipeline stage at a given time.

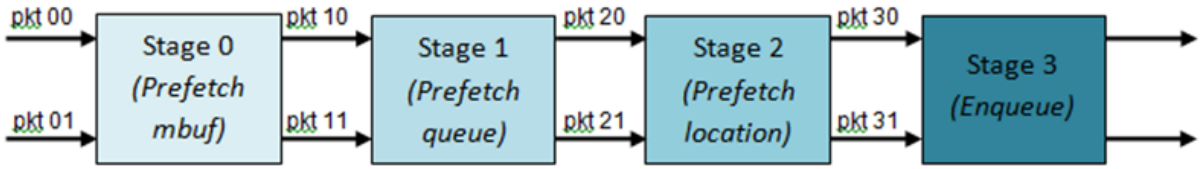


Fig. 21.5: Prefetch Pipeline for the Hierarchical Scheduler Enqueue Operation

The congestion management scheme implemented by the enqueue pipeline described above is very basic: packets are enqueued until a specific queue becomes full, then all the packets destined to the same queue are dropped until packets are consumed (by the dequeue operation). This can be improved by enabling RED/WRED as part of the enqueue pipeline which looks at the queue occupancy and packet priority in order to yield the enqueue/drop decision for a specific packet (as opposed to enqueuing all packets / dropping all packets indiscriminately).

Dequeue State Machine

The sequence of steps to schedule the next packet from the current pipe is:

1. Identify the next active pipe using the bitmap scan operation, *prefetch* pipe.
2. *Read* pipe data structure. Update the credits for the current pipe and its subport. Identify the first active traffic class within the current pipe, select the next queue using WRR, *prefetch* queue pointers for all the 16 queues of the current pipe.
3. *Read* next element from the current WRR queue and *prefetch* its packet descriptor.
4. *Read* the packet length from the packet descriptor (mbuf structure). Based on the packet length and the available credits (of current pipe, pipe traffic class, subport and subport traffic class), take the go/no go scheduling decision for the current packet.

To avoid the cache misses, the above data structures (pipe, queue, queue array, mbufs) are prefetched in advance of being accessed. The strategy of hiding the latency of the prefetch operations is to switch from the current pipe (in grinder A) to another pipe (in grinder B) immediately after a prefetch is issued for the current pipe. This gives enough time to the prefetch operation to complete before the execution switches back to this pipe (in grinder A).

The dequeue pipe state machine exploits the data presence into the processor cache, therefore it tries to send as many packets from the same pipe TC and pipe as possible (up to the available packets and credits) before moving to the next active TC from the same pipe (if any) or to another active pipe.

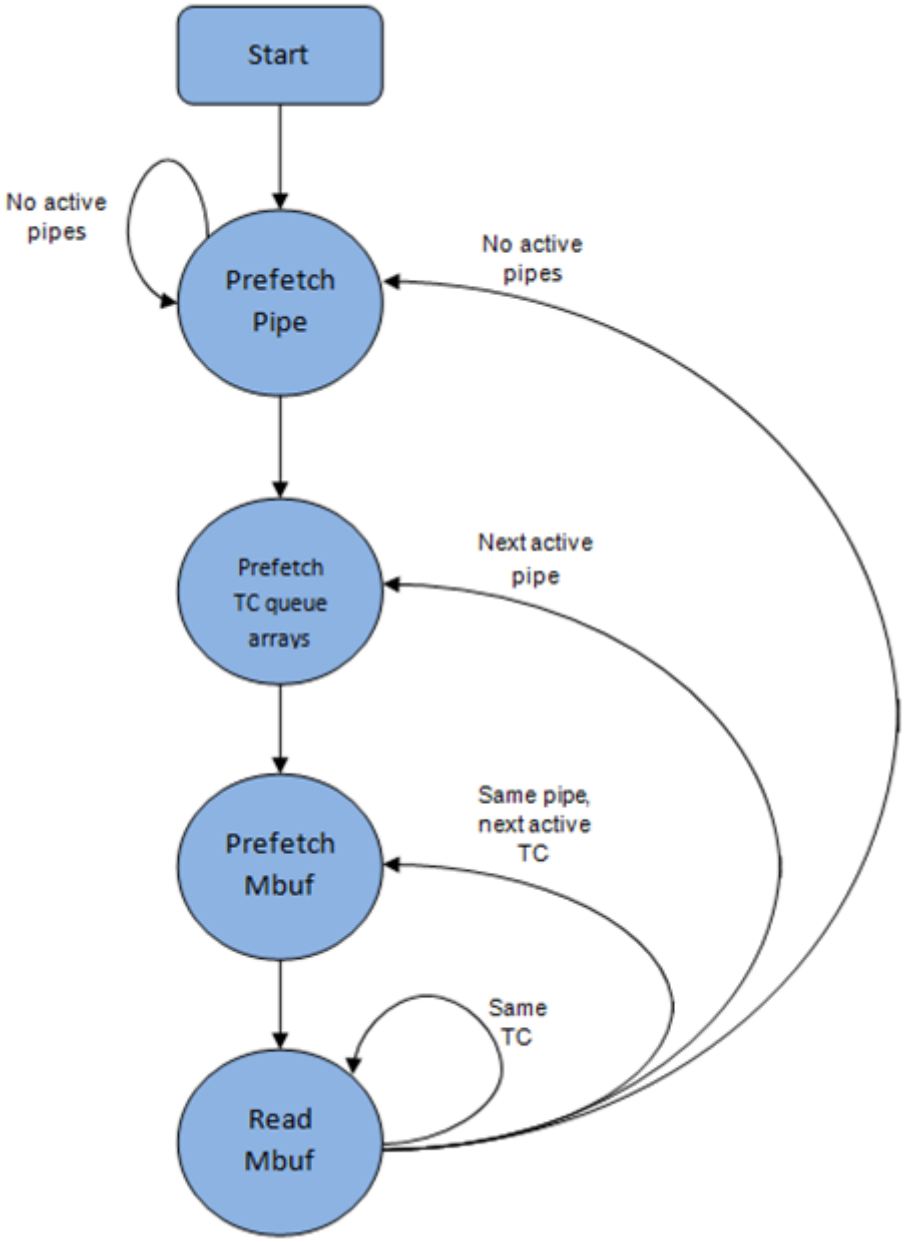


Fig. 21.6: Pipe Prefetch State Machine for the Hierarchical Scheduler Dequeue Operation

Timing and Synchronization

The output port is modeled as a conveyor belt of byte slots that need to be filled by the scheduler with data for transmission. For 10 GbE, there are 1.25 billion byte slots that need to be filled by the port scheduler every second. If the scheduler is not fast enough to fill the slots, provided that enough packets and credits exist, then some slots will be left unused and bandwidth will be wasted.

In principle, the hierarchical scheduler dequeue operation should be triggered by NIC TX. Usually, once the occupancy of the NIC TX input queue drops below a predefined threshold, the port scheduler is woken up (interrupt based or polling based, by continuously monitoring the queue occupancy) to push more packets into the queue.

Internal Time Reference

The scheduler needs to keep track of time advancement for the credit logic, which requires credit updates based on time (for example, subport and pipe traffic shaping, traffic class upper limit enforcement, and so on).

Every time the scheduler decides to send a packet out to the NIC TX for transmission, the scheduler will increment its internal time reference accordingly. Therefore, it is convenient to keep the internal time reference in units of bytes, where a byte signifies the time duration required by the physical interface to send out a byte on the transmission medium. This way, as a packet is scheduled for transmission, the time is incremented with (n + h), where n is the packet length in bytes and h is the number of framing overhead bytes per packet.

Internal Time Reference Re-synchronization

The scheduler needs to align its internal time reference to the pace of the port conveyor belt. The reason is to make sure that the scheduler does not feed the NIC TX with more bytes than the line rate of the physical medium in order to prevent packet drop (by the scheduler, due to the NIC TX input queue being full, or later on, internally by the NIC TX).

The scheduler reads the current time on every dequeue invocation. The CPU time stamp can be obtained by reading either the Time Stamp Counter (TSC) register or the High Precision Event Timer (HPET) register. The current CPU time stamp is converted from number of CPU clocks to number of bytes: *time_bytes = time_cycles / cycles_per_byte, where cycles_per_byte* is the amount of CPU cycles that is equivalent to the transmission time for one byte on the wire (e.g. for a CPU frequency of 2 GHz and a 10GbE port,*cycles_per_byte = 1.6*).

The scheduler maintains an internal time reference of the NIC time. Whenever a packet is scheduled, the NIC time is incremented with the packet length (including framing overhead). On every dequeue invocation, the scheduler checks its internal reference of the NIC time against the current time:

1. If NIC time is in the future (NIC time >= current time), no adjustment of NIC time is needed. This means that scheduler is able to schedule NIC packets before the NIC actually needs those packets, so the NIC TX is well supplied with packets;
2. If NIC time is in the past (NIC time < current time), then NIC time should be adjusted by setting it to the current time. This means that the scheduler is not able to keep up with the speed of the NIC byte conveyor belt, so NIC bandwidth is wasted due to poor packet supply to the NIC TX.

Scheduler Accuracy and Granularity

The scheduler round trip delay (SRTD) is the time (number of CPU cycles) between two consecutive examinations of the same pipe by the scheduler.

To keep up with the output port (that is, avoid bandwidth loss), the scheduler should be able to schedule n packets faster than the same n packets are transmitted by NIC TX.

The scheduler needs to keep up with the rate of each individual pipe, as configured for the pipe token bucket, assuming that no port oversubscription is taking place. This means that the size of the pipe token bucket should be set high enough to prevent it from overflowing due to big SRTD, as this would result in credit loss (and therefore bandwidth loss) for the pipe.

Credit Logic

Scheduling Decision

The scheduling decision to send next packet from (subport S, pipe P, traffic class TC, queue Q) is favorable (packet is sent) when all the conditions below are met:

- Pipe P of subport S is currently selected by one of the port grinders;
- Traffic class TC is the highest priority active traffic class of pipe P;
- Queue Q is the next queue selected by WRR within traffic class TC of pipe P;
- Subport S has enough credits to send the packet;
- Subport S has enough credits for traffic class TC to send the packet;
- Pipe P has enough credits to send the packet;
- Pipe P has enough credits for traffic class TC to send the packet.

If all the above conditions are met, then the packet is selected for transmission and the necessary credits are subtracted from subport S, subport S traffic class TC, pipe P, pipe P traffic class TC.

Framing Overhead

As the greatest common divisor for all packet lengths is one byte, the unit of credit is selected as one byte. The number of credits required for the transmission of a packet of n bytes is equal to (n+h), where h is equal to the number of framing overhead bytes per packet.

Table 21.5: Ethernet Frame Overhead Fields

#	Packet field	Length (bytes)	Comments
1	Preamble	7	
2	Start of Frame Delimiter (SFD)	1	
3	Frame Check Sequence (FCS)	4	Considered overhead only if not included in the mbuf packet length field.
4	Inter Frame Gap (IFG)	12	
5	Total	24	

Traffic Shaping

The traffic shaping for subport and pipe is implemented using a token bucket per subport/per pipe. Each token bucket is implemented using one saturated counter that keeps track of the number of available credits.

The token bucket generic parameters and operations are presented in Table 21.6 and Table 21.7.

Table 21.6: Token Bucket Generic Operations

#	Token Bucket Parameter	Unit	Description
1	bucket_rate	Credits per second	Rate of adding credits to the bucket.
2	bucket_size	Credits	Max number of credits that can be stored in the bucket.

Table 21.7: Token Bucket Generic Parameters

#	Token Bucket Operation	Description
1	Initialization	Bucket set to a predefined value, e.g. zero or half of the bucket size.
2	Credit update	Credits are added to the bucket on top of existing ones, either periodically or on demand, based on the bucket_rate. Credits cannot exceed the upper limit defined by the bucket_size, so any credits to be added to the bucket while the bucket is full are dropped.
3	Credit consumption	As result of packet scheduling, the necessary number of credits is removed from the bucket. The packet can only be sent if enough credits are in the bucket to send the full packet (packet bytes and framing overhead for the packet).

To implement the token bucket generic operations described above, the current design uses the persistent data structure presented in Table 21.8, while the implementation of the token bucket operations is described in Table 21.9.

Table 21.8: Token Bucket Persistent Data Structure

#	Token bucket field	Unit	Description
1	tb_time	Bytes	Time of the last credit update. Measured in bytes instead of seconds or CPU cycles for ease of credit consumption operation (as the current time is also maintained in bytes). See Section 26.2.4.5.1 "Internal Time Reference" for an explanation of why the time is maintained in byte units.
2	tb_period	Bytes	Time period that should elapse since the last credit update in order for the bucket to be awarded tb_credits_per_period worth or credits.
3	tb_credits_per_period	Bytes	Credit allowance per tb_period.
4	tb_size	Bytes	Bucket size, i.e. upper limit for the tb_credits.
5	tb_credits	Bytes	Number of credits currently in the bucket.

The bucket rate (in bytes per second) can be computed with the following formula:

*bucket_rate = (tb_credits_per_period / tb_period) * r*

where, r = port line rate (in bytes per second).

Table 21.9: Token Bucket Operations

#	Token bucket operation	Description
1	Initialization	*tb_credits = 0; or tb_credits = tb_size / 2;*
2	Credit update	Credit update options: • Every time a packet is sent for a port, update the credits of all the the subports and pipes of that port. Not feasible. • Every time a packet is sent, update the credits for the pipe and subport. Very accurate, but not needed (a lot of calculations). • Every time a pipe is selected (that is, picked by one of the grinders), update the credits for the pipe and its subport. The current implementation is using option 3. According to Section *Dequeue State Machine*, the pipe and subport credits are updated every time a pipe is selected by the dequeue process before the pipe and subport credits are actually used. The implementation uses a tradeoff between accuracy and speed by updating the bucket credits only when at least a full *tb_period* has elapsed since the last update. • Full accuracy can be achieved by selecting the value for *tb_period* for which *tb_credits_per_period = 1*. • When full accuracy is not required, better performance is achieved by setting *tb_credits* to a larger value. Update operations: • n_periods = (time - tb_time) / tb_period; • tb_credits += n_periods * tb_credits_per_period; • tb_credits = min(tb_credits, tb_size); • tb_time += n_periods * tb_period;

Traffic Classes

Implementation of Strict Priority Scheduling Strict priority scheduling of traffic classes within the same pipe is implemented by the pipe dequeue state machine, which selects the queues in ascending order. Therefore, queues 0..3 (associated with TC 0, highest priority TC) are handled before queues 4..7 (TC 1, lower priority than TC 0), which are handled before queues 8..11 (TC 2), which are handled before queues 12..15 (TC 3, lowest priority TC).

Upper Limit Enforcement The traffic classes at the pipe and subport levels are not traffic shaped, so there is no token bucket maintained in this context. The upper limit for the traffic classes at the subport and pipe levels is enforced by periodically refilling the subport / pipe traffic class credit counter, out of which credits are consumed every time a packet is scheduled for that subport / pipe, as described in Table 21.10 and Table 21.11.

Table 21.10: Subport/Pipe Traffic Class Upper Limit Enforcement Persistent Data Structure

#	Subport or pipe field	Unit	Description
1	tc_time	Bytes	Time of the next update (upper limit refill) for the 4 TCs of the current subport / pipe. See Section *Internal Time Reference* for the explanation of why the time is maintained in byte units.
2	tc_period	Bytes	Time between two consecutive updates for the 4 TCs of the current subport / pipe. This is expected to be many times bigger than the typical value of the token bucket tb_period.
3	tc_credits_per_period	Bytes	Upper limit for the number of credits allowed to be consumed by the current TC during each enforcement period tc_period.
4	tc_credits	Bytes	Current upper limit for the number of credits that can be consumed by the current traffic class for the remainder of the current enforcement period.

Table 21.11: Subport/Pipe Traffic Class Upper Limit Enforcement Operations

#	Traffic Class Operation	Description
1	Initialization	tc_credits = tc_credits_per_period; tc_time = tc_period;

2	Credit update	Update operations:
		if (time >= tc_time) {
		tc_credits = tc_credits_per_period;
		tc_time = time + tc_period;
		}

3	Credit consumption (on packet scheduling)	As result of packet scheduling, the TC limit is decreased with the necessary number of credits. The packet can only be sent if enough credits are currently available in the TC limit to send the full packet (packet bytes and framing overhead for the packet).
		Scheduling operations:
		pkt_credits = pk_len + frame_overhead;
		if (tc_credits >= pkt_credits) {tc_credits -= pkt_credits;}

Weighted Round Robin (WRR)

The evolution of the WRR design solution from simple to complex is shown in Table 21.12.

Table 21.12: Weighted Round Robin (WRR)

#	All Queues Active?	Equal Weights for All Queues?	All Packets Equal?	Strategy
1	Yes	Yes	Yes	**Byte level round robin** *Next queue* queue #i, i = *(i + 1) % n*
2	Yes	Yes	No	**Packet level round robin** Consuming one byte from queue #i requires consuming exactly one token for queue #i. T(i) = Accumulated number of tokens previously consumed from queue #i. Every time a packet is consumed from queue #i, T(i) is updated as: T(i) += *pkt_len*. *Next queue* : queue with the smallest T.
3	Yes	No	No	**Packet level weighted round robin** This case can be reduced to the previous case by introducing a cost per byte that is different for each queue. Queues with lower weights have a higher cost per byte. This way, it is still meaningful to compare the consumption amongst different queues in order to select the next queue. w(i) = Weight of queue #i t(i) = Tokens per byte for queue #i, defined as the inverse weight of queue #i. For example, if w[0..3] = [1:2:4:8], then t[0..3] = [8:4:2:1]; if w[0..3] = [1:4:15:20], then t[0..3] = [60:15:4:3]. Consuming one byte from queue #i requires consuming t(i) tokens for queue #i. T(i) = Accumulated number of tokens previously consumed from queue #i. Every time a packet is consumed from queue #i, T(i) is updated as: *T(i) += pkt_len * t(i)*. *Next queue* : queue with the smallest T.
4	No	No	No	**Packet level weighted round robin with variable queue status** Reduce this case to the previous case by setting the consumption of inactive queues to a high number, so that the inactive queues will never be selected by the smallest T logic. To prevent T from overflowing as result of successive accumulations, T(i) is truncated after each packet consumption for all queues. For example, T[0..3] = [1000, 1100, 1200, 1300] is truncated to T[0..3] = [0, 100, 200, 300] by subtracting the min T from T(i), i = 0..n. This requires having at least one active queue in the set of input queues, which is guaranteed by the dequeue state machine never selecting an inactive traffic class. *mask(i) = Saturation mask for queue #i, defined as:* mask(i) = (queue #i is active)? 0 : 0xFFFFFFFF; w(i) = Weight of queue #i t(i) = Tokens per byte for queue #i, defined as the inverse weight of queue #i. T(i) = Accumulated numbers of tokens previously consumed from queue #i.

Subport Traffic Class Oversubscription

Problem Statement Oversubscription for subport traffic class X is a configuration-time event that occurs when more bandwidth is allocated for traffic class X at the level of subport member pipes than allocated for the same traffic class at the parent subport level.

The existence of the oversubscription for a specific subport and traffic class is solely the result of pipe and subport-level configuration as opposed to being created due to dynamic evolution of the traffic load at run-time (as congestion is).

When the overall demand for traffic class X for the current subport is low, the existence of the oversubscription condition does not represent a problem, as demand for traffic class X is completely satisfied for all member pipes. However, this can no longer be achieved when the aggregated demand for traffic class X for all subport member pipes exceeds the limit configured at the subport level.

Solution Space summarizes some of the possible approaches for handling this problem, with the third approach selected for implementation.

Table 21.13: Subport Traffic Class Oversubscription

No.	Approach	Description
1	Don't care	First come, first served. This approach is not fair amongst subport member pipes, as pipes that are served first will use up as much bandwidth for TC X as they need, while pipes that are served later will receive poor service due to bandwidth for TC X at the subport level being scarce.
2	Scale down all pipes	All pipes within the subport have their bandwidth limit for TC X scaled down by the same factor. This approach is not fair among subport member pipes, as the low end pipes (that is, pipes configured with low bandwidth) can potentially experience severe service degradation that might render their service unusable (if available bandwidth for these pipes drops below the minimum requirements for a workable service), while the service degradation for high end pipes might not be noticeable at all.
3	Cap the high demand pipes	Each subport member pipe receives an equal share of the bandwidth available at run-time for TC X at the subport level. Any bandwidth left unused by the low-demand pipes is redistributed in equal portions to the high-demand pipes. This way, the high-demand pipes are truncated while the low-demand pipes are not impacted.

Typically, the subport TC oversubscription feature is enabled only for the lowest priority traffic class (TC 3), which is typically used for best effort traffic, with the management plane preventing this condition from occurring for the other (higher priority) traffic classes.

To ease implementation, it is also assumed that the upper limit for subport TC 3 is set to 100% of the subport rate, and that the upper limit for pipe TC 3 is set to 100% of pipe rate for all subport member pipes.

Implementation Overview The algorithm computes a watermark, which is periodically updated based on the current demand experienced by the subport member pipes, whose purpose is to limit the amount of traffic that each pipe is allowed to send for TC 3. The watermark is

computed at the subport level at the beginning of each traffic class upper limit enforcement period and the same value is used by all the subport member pipes throughout the current enforcement period. illustrates how the watermark computed as subport level at the beginning of each period is propagated to all subport member pipes.

At the beginning of the current enforcement period (which coincides with the end of the previous enforcement period), the value of the watermark is adjusted based on the amount of bandwidth allocated to TC 3 at the beginning of the previous period that was not left unused by the subport member pipes at the end of the previous period.

If there was subport TC 3 bandwidth left unused, the value of the watermark for the current period is increased to encourage the subport member pipes to consume more bandwidth. Otherwise, the value of the watermark is decreased to enforce equality of bandwidth consumption among subport member pipes for TC 3.

The increase or decrease in the watermark value is done in small increments, so several enforcement periods might be required to reach the equilibrium state. This state can change at any moment due to variations in the demand experienced by the subport member pipes for TC 3, for example, as a result of demand increase (when the watermark needs to be lowered) or demand decrease (when the watermark needs to be increased).

When demand is low, the watermark is set high to prevent it from impeding the subport member pipes from consuming more bandwidth. The highest value for the watermark is picked as the highest rate configured for a subport member pipe. Table 21.14 and Table 21.15 illustrates the watermark operation.

Table 21.14: Watermark Propagation from Subport Level to Member Pipes at the Beginning of Each Traffic Class Upper Limit Enforcement Period

No.	Subport Traffic Class Operation	Description
1	Initialization	**Subport level**: subport_period_id= 0 **Pipe level**: pipe_period_id = 0

2	Credit update	**Subport Level**:
		if (time>=subport_tc_time)
		{ subport_wm = water_mark_update();
		subport_tc_time = time + subport_tc_period;
		subport_period_id++;
		}
		Pipelevel:
		if(pipe_period_id != subport_period_id)
		{
		pipe_ov_credits = subport_wm * pipe_weight;
		pipe_period_id = subport_period_id;
		}

3	Credit consumption (on packet scheduling)	**Pipe level:**
		pkt_credits = pk_len + frame_overhead;
		if(pipe_ov_credits >= pkt_credits{
		pipe_ov_credits - = pkt_credits;
		}

Table 21.15: Watermark Calculation

No.	Subport Traffic Class Operation	Description
1	Initialization	**Subport level:** wm = WM_MAX

2	Credit update	**Subport level (water_mark_update):**
		tc0_cons = subport_tc0_credits_per_period - subport_tc0_credits;
		tc1_cons = subport_tc1_credits_per_period - subport_tc1_credits;
		tc2_cons = subport_tc2_credits_per_period - subport_tc2_credits;
		tc3_cons = subport_tc3_credits_per_period - subport_tc3_credits;
		tc3_cons_max = subport_tc3_credits_per_period - (tc0_cons + tc1_cons + tc2_cons);
		if(tc3_consumption > (tc3_consumption_max - MTU)){
		wm -= wm >> 7;
		if(wm < WM_MIN)
		wm = WM_MIN;
		} else {
		wm += (wm >> 7) + 1;
		if(wm > WM_MAX) wm = WM_MAX;
		}

21.2.5 Worst Case Scenarios for Performance

Lots of Active Queues with Not Enough Credits

The more queues the scheduler has to examine for packets and credits in order to select one packet, the lower the performance of the scheduler is.

The scheduler maintains the bitmap of active queues, which skips the non-active queues, but in order to detect whether a specific pipe has enough credits, the pipe has to be drilled down using the pipe dequeue state machine, which consumes cycles regardless of the scheduling result (no packets are produced or at least one packet is produced).

This scenario stresses the importance of the policer for the scheduler performance: if the pipe

does not have enough credits, its packets should be dropped as soon as possible (before they reach the hierarchical scheduler), thus rendering the pipe queues as not active, which allows the dequeue side to skip that pipe with no cycles being spent on investigating the pipe credits that would result in a "not enough credits" status.

Single Queue with 100% Line Rate

The port scheduler performance is optimized for a large number of queues. If the number of queues is small, then the performance of the port scheduler for the same level of active traffic is expected to be worse than the performance of a small set of message passing queues.

21.3 Dropper

The purpose of the DPDK dropper is to drop packets arriving at a packet scheduler to avoid congestion. The dropper supports the Random Early Detection (RED), Weighted Random Early Detection (WRED) and tail drop algorithms. Fig. 21.7 illustrates how the dropper integrates with the scheduler. The DPDK currently does not support congestion management so the dropper provides the only method for congestion avoidance.

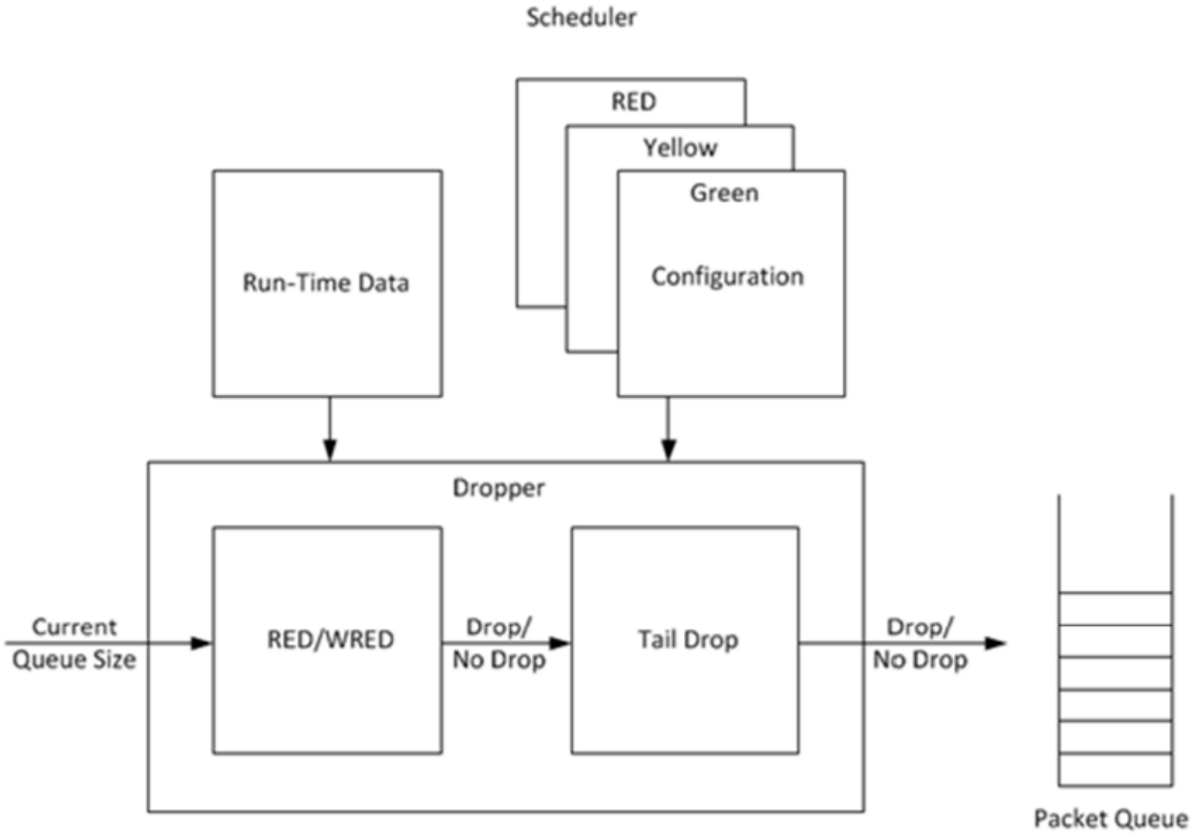


Fig. 21.7: High-level Block Diagram of the DPDK Dropper

The dropper uses the Random Early Detection (RED) congestion avoidance algorithm as documented in the reference publication. The purpose of the RED algorithm is to monitor a packet queue, determine the current congestion level in the queue and decide whether an arriving packet should be enqueued or dropped. The RED algorithm uses an Exponential Weighted

Moving Average (EWMA) filter to compute average queue size which gives an indication of the current congestion level in the queue.

For each enqueue operation, the RED algorithm compares the average queue size to minimum and maximum thresholds. Depending on whether the average queue size is below, above or in between these thresholds, the RED algorithm calculates the probability that an arriving packet should be dropped and makes a random decision based on this probability.

The dropper also supports Weighted Random Early Detection (WRED) by allowing the scheduler to select different RED configurations for the same packet queue at run-time. In the case of severe congestion, the dropper resorts to tail drop. This occurs when a packet queue has reached maximum capacity and cannot store any more packets. In this situation, all arriving packets are dropped.

The flow through the dropper is illustrated in Fig. 21.8. The RED/WRED algorithm is exercised first and tail drop second.

The use cases supported by the dropper are:

- – Initialize configuration data
- – Initialize run-time data
- – Enqueue (make a decision to enqueue or drop an arriving packet)
- – Mark empty (record the time at which a packet queue becomes empty)

The configuration use case is explained in *Section2.23.3.1*, the enqueue operation is explained in *Section 2.23.3.2* and the mark empty operation is explained in *Section 2.23.3.3*.

21.3.1 Configuration

A RED configuration contains the parameters given in Table 21.16.

Table 21.16: RED Configuration Parameters

Parameter	Minimum	Maximum	Typical
Minimum Threshold	0	1022	1/4 x queue size
Maximum Threshold	1	1023	1/2 x queue size
Inverse Mark Probability	1	255	10
EWMA Filter Weight	1	12	9

The meaning of these parameters is explained in more detail in the following sections. The format of these parameters as specified to the dropper module API corresponds to the format used by Cisco* in their RED implementation. The minimum and maximum threshold parameters are specified to the dropper module in terms of number of packets. The mark probability parameter is specified as an inverse value, for example, an inverse mark probability parameter value of 10 corresponds to a mark probability of 1/10 (that is, 1 in 10 packets will be dropped). The EWMA filter weight parameter is specified as an inverse log value, for example, a filter weight parameter value of 9 corresponds to a filter weight of 1/29.

21.3.2 Enqueue Operation

In the example shown in Fig. 21.9, q (actual queue size) is the input value, avg (average queue size) and count (number of packets since the last drop) are run-time values, decision is

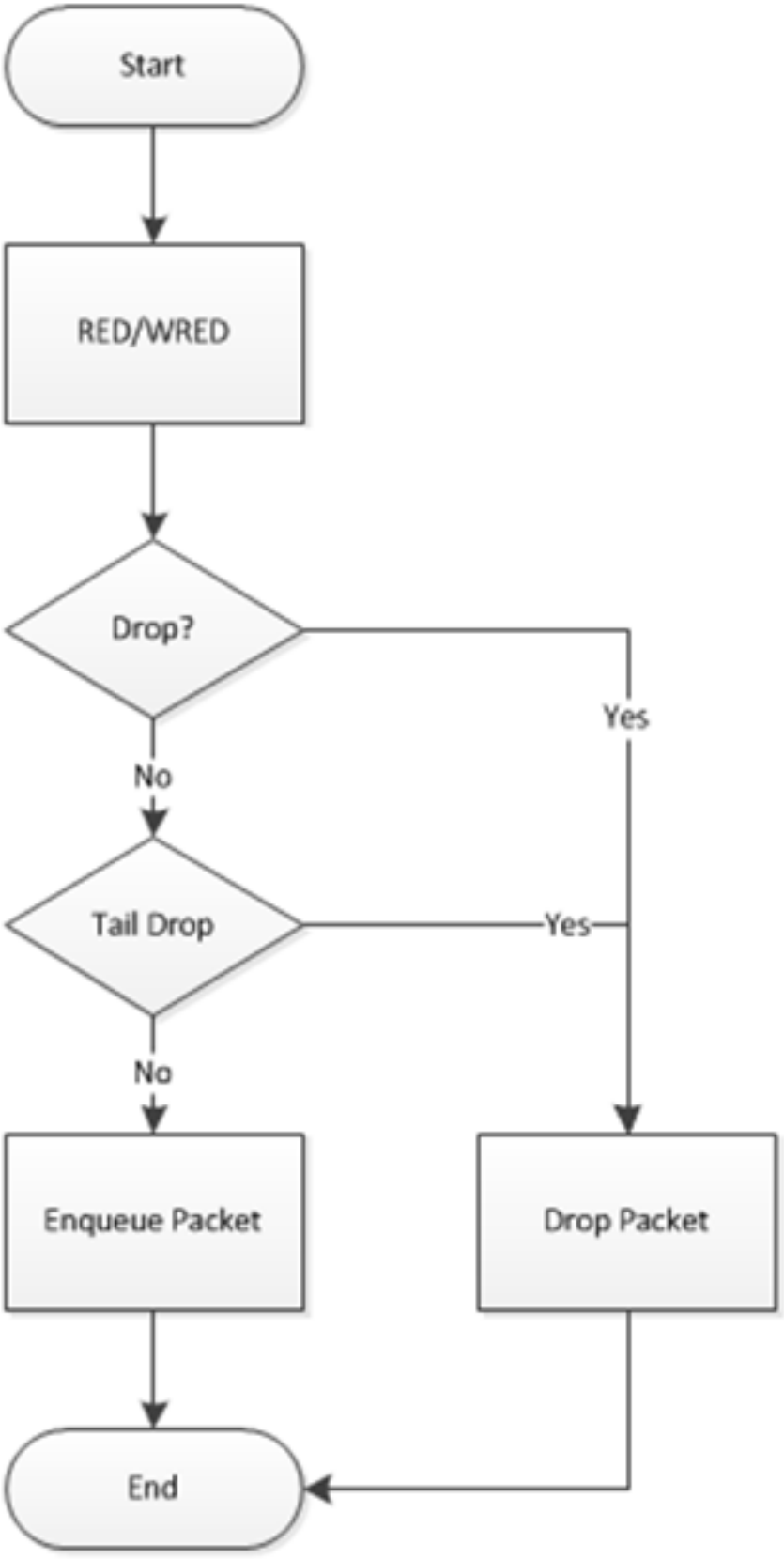


Fig. 21.8: Flow Through the Dropper

the output value and the remaining values are configuration parameters.

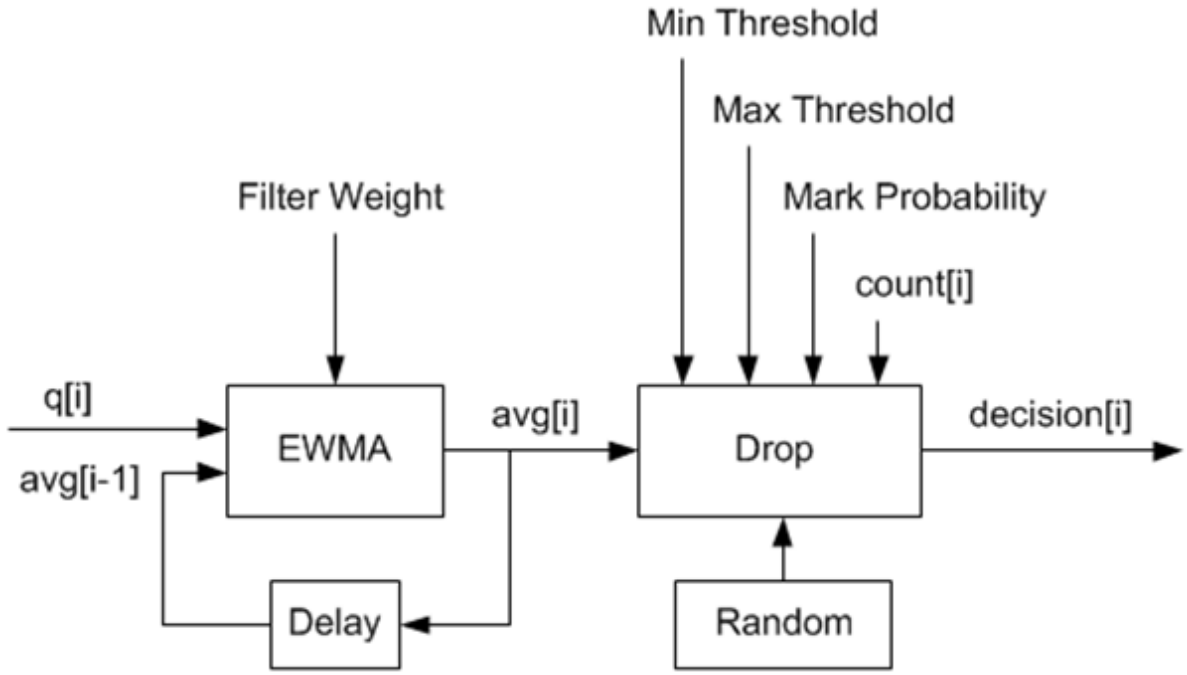


Fig. 21.9: Example Data Flow Through Dropper

EWMA Filter Microblock

The purpose of the EWMA Filter microblock is to filter queue size values to smooth out transient changes that result from "bursty" traffic. The output value is the average queue size which gives a more stable view of the current congestion level in the queue.

The EWMA filter has one configuration parameter, filter weight, which determines how quickly or slowly the average queue size output responds to changes in the actual queue size input. Higher values of filter weight mean that the average queue size responds more quickly to changes in actual queue size.

Average Queue Size Calculation when the Queue is not Empty

The definition of the EWMA filter is given in the following equation.

$$avg[i] = (1 - w_q) \times avg[i-1] + w_q \times q[i]$$

Where:

- *avg* = average queue size
- *wq* = filter weight
- *q* = actual queue size

Note:

The filter weight, wq = 1/2^n, where n is the filter weight parameter value passed to the dropper modu
on configuration (see *Section2.23.3.1*).

Average Queue Size Calculation when the Queue is Empty

The EWMA filter does not read time stamps and instead assumes that enqueue operations will happen quite regularly. Special handling is required when the queue becomes empty as the queue could be empty for a short time or a long time. When the queue becomes empty, average queue size should decay gradually to zero instead of dropping suddenly to zero or remaining stagnant at the last computed value. When a packet is enqueued on an empty queue, the average queue size is computed using the following formula:

$$avg[i] = \mathrm{avg}[i-1] \times \left(1 - \mathrm{w_q}\right)^m$$

Where:

- *m* = the number of enqueue operations that could have occurred on this queue while the queue was empty

In the dropper module, *m* is defined as:

$$m = \left(\frac{time - qtime}{s}\right)$$

Where:

- *time* = current time
- *qtime* = time the queue became empty
- *s* = typical time between successive enqueue operations on this queue

The time reference is in units of bytes, where a byte signifies the time duration required by the physical interface to send out a byte on the transmission medium (see Section *Internal Time Reference*). The parameter s is defined in the dropper module as a constant with the value: s=2^22. This corresponds to the time required by every leaf node in a hierarchy with 64K leaf nodes to transmit one 64-byte packet onto the wire and represents the worst case scenario. For much smaller scheduler hierarchies, it may be necessary to reduce the parameter s, which is defined in the red header source file (rte_red.h) as:

```
#define RTE_RED_S
```

Since the time reference is in bytes, the port speed is implied in the expression: *time-qtime*. The dropper does not have to be configured with the actual port speed. It adjusts automatically to low speed and high speed links.

Implementation

A numerical method is used to compute the factor (1-wq)^m that appears in Equation 2.

This method is based on the following identity:

$$a \equiv 2^{(b \times \log_2(a))}$$

This allows us to express the following:

$$(1-w_q)^m = 2^{(m \times \log_2(1-w_q))}$$

In the dropper module, a look-up table is used to compute log2(1-wq) for each value of wq supported by the dropper module. The factor (1-wq)^m can then be obtained by multiplying the table value by *m* and applying shift operations. To avoid overflow in the multiplication, the value, *m*, and the look-up table values are limited to 16 bits. The total size of the look-up table is 56 bytes. Once the factor (1-wq)^m is obtained using this method, the average queue size can be calculated from Equation 2.

Alternative Approaches

Other methods for calculating the factor (1-wq)^m in the expression for computing average queue size when the queue is empty (Equation 2) were considered. These approaches include:

- Floating-point evaluation
- Fixed-point evaluation using a small look-up table (512B) and up to 16 multiplications (this is the approach used in the FreeBSD* ALTQ RED implementation)
- Fixed-point evaluation using a small look-up table (512B) and 16 SSE multiplications (SSE optimized version of the approach used in the FreeBSD* ALTQ RED implementation)
- Large look-up table (76 KB)

The method that was finally selected (described above in Section 26.3.2.2.1) out performs all of these approaches in terms of run-time performance and memory requirements and also achieves accuracy comparable to floating-point evaluation. Table 21.17 lists the performance of each of these alternative approaches relative to the method that is used in the dropper. As can be seen, the floating-point implementation achieved the worst performance.

Table 21.17: Relative Performance of A

Method	Relative Performance
Current dropper method (see *Section 23.3.2.1.3*)	100%
Fixed-point method with small (512B) look-up table	148%
SSE method with small (512B) look-up table	114%
Large (76KB) look-up table	118%
Floating-point	595%
Note: In this case, since performance is expressed as time spent executing the operation in a specific condi	

Drop Decision Block

The Drop Decision block:

- Compares the average queue size with the minimum and maximum thresholds
- Calculates a packet drop probability
- Makes a random decision to enqueue or drop an arriving packet

The calculation of the drop probability occurs in two stages. An initial drop probability is calculated based on the average queue size, the minimum and maximum thresholds and the mark probability. An actual drop probability is then computed from the initial drop probability. The

actual drop probability takes the count run-time value into consideration so that the actual drop probability increases as more packets arrive to the packet queue since the last packet was dropped.

Initial Packet Drop Probability

The initial drop probability is calculated using the following equation.

$$p_b = \begin{cases} 0, & avg < min_{th} \\ max_p \left(\frac{avg - min_{th}}{max_{th} - min_{th}} \right), & min_{th} \leq avg < max_{th} \\ 1, & avg \geq max_{th} \end{cases}$$

Where:

- *maxp* = mark probability
- *avg* = average queue size
- *minth* = minimum threshold
- *maxth* = maximum threshold

The calculation of the packet drop probability using Equation 3 is illustrated in Fig. 21.10. If the average queue size is below the minimum threshold, an arriving packet is enqueued. If the average queue size is at or above the maximum threshold, an arriving packet is dropped. If the average queue size is between the minimum and maximum thresholds, a drop probability is calculated to determine if the packet should be enqueued or dropped.

Actual Drop Probability

If the average queue size is between the minimum and maximum thresholds, then the actual drop probability is calculated from the following equation.

$$p_a = \frac{p_b}{(2 - count \times p_b)}$$

Where:

- *Pb* = initial drop probability (from Equation 3)
- *count* = number of packets that have arrived since the last drop

The constant 2, in Equation 4 is the only deviation from the drop probability formulae given in the reference document where a value of 1 is used instead. It should be noted that the value pa computed from can be negative or greater than 1. If this is the case, then a value of 1 should be used instead.

The initial and actual drop probabilities are shown in Fig. 21.11. The actual drop probability is shown for the case where the formula given in the reference document1 is used (blue curve) and also for the case where the formula implemented in the dropper module, is used (red curve). The formula in the reference document results in a significantly higher drop rate compared to the mark probability configuration parameter specified by the user. The choice to deviate from the reference document is simply a design decision and one that has been taken by other RED implementations, for example, FreeBSD* ALTQ RED.

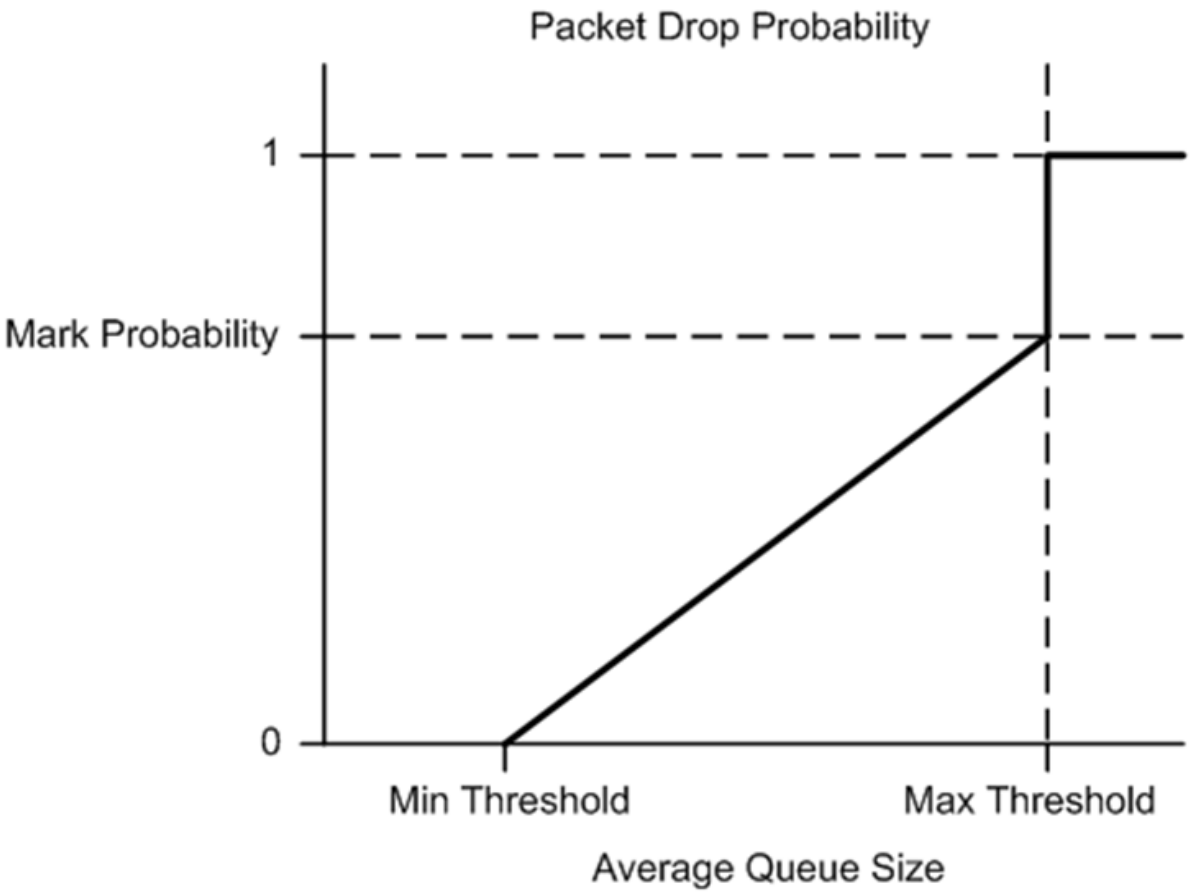


Fig. 21.10: Packet Drop Probability for a Given RED Configuration

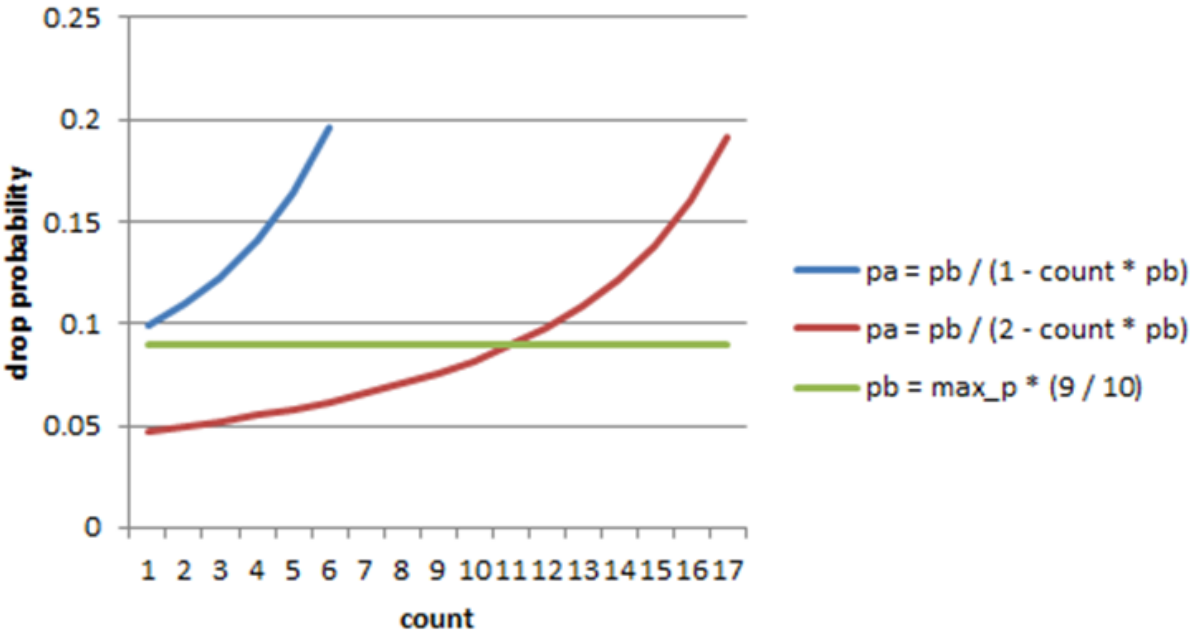


Fig. 21.11: Initial Drop Probability (pb), Actual Drop probability (pa) Computed Using a Factor 1 (Blue Curve) and a Factor 2 (Red Curve)

21.3.3 Queue Empty Operation

The time at which a packet queue becomes empty must be recorded and saved with the RED run-time data so that the EWMA filter block can calculate the average queue size on the next enqueue operation. It is the responsibility of the calling application to inform the dropper module through the API that a queue has become empty.

21.3.4 Source Files Location

The source files for the DPDK dropper are located at:

- DPDK/lib/librte_sched/rte_red.h
- DPDK/lib/librte_sched/rte_red.c

21.3.5 Integration with the DPDK QoS Scheduler

RED functionality in the DPDK QoS scheduler is disabled by default. To enable it, use the DPDK configuration parameter:

```
CONFIG_RTE_SCHED_RED=y
```

This parameter must be set to y. The parameter is found in the build configuration files in the DPDK/config directory, for example, DPDK/config/common_linuxapp. RED configuration parameters are specified in the rte_red_params structure within the rte_sched_port_params structure that is passed to the scheduler on initialization. RED parameters are specified separately for four traffic classes and three packet colors (green, yellow and red) allowing the scheduler to implement Weighted Random Early Detection (WRED).

21.3.6 Integration with the DPDK QoS Scheduler Sample Application

The DPDK QoS Scheduler Application reads a configuration file on start-up. The configuration file includes a section containing RED parameters. The format of these parameters is described in *Section2.23.3.1*. A sample RED configuration is shown below. In this example, the queue size is 64 packets.

Note: For correct operation, the same EWMA filter weight parameter (wred weight) should be used for each packet color (green, yellow, red) in the same traffic class (tc).

```
 ; RED params per traffic class and color (Green / Yellow / Red)

[red]
tc 0 wred min = 28 22 16
tc 0 wred max = 32 32 32
tc 0 wred inv prob = 10 10 10
tc 0 wred weight = 9 9 9

tc 1 wred min = 28 22 16
tc 1 wred max = 32 32 32
tc 1 wred inv prob = 10 10 10
tc 1 wred weight = 9 9 9

tc 2 wred min = 28 22 16
tc 2 wred max = 32 32 32
```

```
tc 2 wred inv prob = 10 10 10
tc 2 wred weight = 9 9 9

tc 3 wred min = 28 22 16
tc 3 wred max = 32 32 32
tc 3 wred inv prob = 10 10 10
tc 3 wred weight = 9 9 9
```

With this configuration file, the RED configuration that applies to green, yellow and red packets in traffic class 0 is shown in Table 21.18.

Table 21.18: RED Configuration Corresponding to RED Configuration File

RED Parameter	Configuration Name	Green	Yellow	Red
Minimum Threshold	tc 0 wred min	28	22	16
Maximum Threshold	tc 0 wred max	32	32	32
Mark Probability	tc 0 wred inv prob	10	10	10
EWMA Filter Weight	tc 0 wred weight	9	9	9

21.3.7 Application Programming Interface (API)

Enqueue API

The syntax of the enqueue API is as follows:

```
int rte_red_enqueue(const struct rte_red_config *red_cfg, struct rte_red *red, const unsigned q
```

The arguments passed to the enqueue API are configuration data, run-time data, the current size of the packet queue (in packets) and a value representing the current time. The time reference is in units of bytes, where a byte signifies the time duration required by the physical interface to send out a byte on the transmission medium (see Section 26.2.4.5.1 "Internal Time Reference"). The dropper reuses the scheduler time stamps for performance reasons.

Empty API

The syntax of the empty API is as follows:

```
void rte_red_mark_queue_empty(struct rte_red *red, const uint64_t time)
```

The arguments passed to the empty API are run-time data and the current time in bytes.

21.4 Traffic Metering

The traffic metering component implements the Single Rate Three Color Marker (srTCM) and Two Rate Three Color Marker (trTCM) algorithms, as defined by IETF RFC 2697 and 2698 respectively. These algorithms meter the stream of incoming packets based on the allowance defined in advance for each traffic flow. As result, each incoming packet is tagged as green, yellow or red based on the monitored consumption of the flow the packet belongs to.

21.4.1 Functional Overview

The srTCM algorithm defines two token buckets for each traffic flow, with the two buckets sharing the same token update rate:

- Committed (C) bucket: fed with tokens at the rate defined by the Committed Information Rate (CIR) parameter (measured in IP packet bytes per second). The size of the C bucket is defined by the Committed Burst Size (CBS) parameter (measured in bytes);
- Excess (E) bucket: fed with tokens at the same rate as the C bucket. The size of the E bucket is defined by the Excess Burst Size (EBS) parameter (measured in bytes).

The trTCM algorithm defines two token buckets for each traffic flow, with the two buckets being updated with tokens at independent rates:

- Committed (C) bucket: fed with tokens at the rate defined by the Committed Information Rate (CIR) parameter (measured in bytes of IP packet per second). The size of the C bucket is defined by the Committed Burst Size (CBS) parameter (measured in bytes);
- Peak (P) bucket: fed with tokens at the rate defined by the Peak Information Rate (PIR) parameter (measured in IP packet bytes per second). The size of the P bucket is defined by the Peak Burst Size (PBS) parameter (measured in bytes).

Please refer to RFC 2697 (for srTCM) and RFC 2698 (for trTCM) for details on how tokens are consumed from the buckets and how the packet color is determined.

Color Blind and Color Aware Modes

For both algorithms, the color blind mode is functionally equivalent to the color aware mode with input color set as green. For color aware mode, a packet with red input color can only get the red output color, while a packet with yellow input color can only get the yellow or red output colors.

The reason why the color blind mode is still implemented distinctly than the color aware mode is that color blind mode can be implemented with fewer operations than the color aware mode.

21.4.2 Implementation Overview

For each input packet, the steps for the srTCM / trTCM algorithms are:

- Update the C and E / P token buckets. This is done by reading the current time (from the CPU timestamp counter), identifying the amount of time since the last bucket update and computing the associated number of tokens (according to the pre-configured bucket rate). The number of tokens in the bucket is limited by the pre-configured bucket size;
- Identify the output color for the current packet based on the size of the IP packet and the amount of tokens currently available in the C and E / P buckets; for color aware mode only, the input color of the packet is also considered. When the output color is not red, a number of tokens equal to the length of the IP packet are subtracted from the C or E /P or both buckets, depending on the algorithm and the output color of the packet.

CHAPTER

TWENTYTWO

POWER MANAGEMENT

The DPDK Power Management feature allows users space applications to save power by dynamically adjusting CPU frequency or entering into different C-States.

- Adjusting the CPU frequency dynamically according to the utilization of RX queue.
- Entering into different deeper C-States according to the adaptive algorithms to speculate brief periods of time suspending the application if no packets are received.

The interfaces for adjusting the operating CPU frequency are in the power management library. C-State control is implemented in applications according to the different use cases.

22.1 CPU Frequency Scaling

The Linux kernel provides a cpufreq module for CPU frequency scaling for each lcore. For example, for cpuX, /sys/devices/system/cpu/cpuX/cpufreq/ has the following sys files for frequency scaling:

- affected_cpus
- bios_limit
- cpuinfo_cur_freq
- cpuinfo_max_freq
- cpuinfo_min_freq
- cpuinfo_transition_latency
- related_cpus
- scaling_available_frequencies
- scaling_available_governors
- scaling_cur_freq
- scaling_driver
- scaling_governor
- scaling_max_freq
- scaling_min_freq
- scaling_setspeed

In the DPDK, scaling_governor is configured in user space. Then, a user space application can prompt the kernel by writing scaling_setspeed to adjust the CPU frequency according to the strategies defined by the user space application.

22.2 Core-load Throttling through C-States

Core state can be altered by speculative sleeps whenever the specified lcore has nothing to do. In the DPDK, if no packet is received after polling, speculative sleeps can be triggered according the strategies defined by the user space application.

22.3 API Overview of the Power Library

The main methods exported by power library are for CPU frequency scaling and include the following:

- **Freq up**: Prompt the kernel to scale up the frequency of the specific lcore.
- **Freq down**: Prompt the kernel to scale down the frequency of the specific lcore.
- **Freq max**: Prompt the kernel to scale up the frequency of the specific lcore to the maximum.
- **Freq min**: Prompt the kernel to scale down the frequency of the specific lcore to the minimum.
- **Get available freqs**: Read the available frequencies of the specific lcore from the sys file.
- **Freq get**: Get the current frequency of the specific lcore.
- **Freq set**: Prompt the kernel to set the frequency for the specific lcore.

22.4 User Cases

The power management mechanism is used to save power when performing L3 forwarding.

22.5 References

- l3fwd-power: The sample application in DPDK that performs L3 forwarding with power management.
- The "L3 Forwarding with Power Management Sample Application" chapter in the *DPDK Sample Application's User Guide*.

CHAPTER

TWENTYTHREE

PACKET CLASSIFICATION AND ACCESS CONTROL

The DPDK provides an Access Control library that gives the ability to classify an input packet based on a set of classification rules.

The ACL library is used to perform an N-tuple search over a set of rules with multiple categories and find the best match (highest priority) for each category. The library API provides the following basic operations:

- Create a new Access Control (AC) context.
- Add rules into the context.
- For all rules in the context, build the runtime structures necessary to perform packet classification.
- Perform input packet classifications.
- Destroy an AC context and its runtime structures and free the associated memory.

23.1 Overview

23.1.1 Rule definition

The current implementation allows the user for each AC context to specify its own rule (set of fields) over which packet classification will be performed. Though there are few restrictions on the rule fields layout:

- First field in the rule definition has to be one byte long.
- All subsequent fields has to be grouped into sets of 4 consecutive bytes.

This is done mainly for performance reasons - search function processes the first input byte as part of the flow setup and then the inner loop of the search function is unrolled to process four input bytes at a time.

To define each field inside an AC rule, the following structure is used:

```
struct rte_acl_field_def {
    uint8_t type;         /*< type - ACL_FIELD_TYPE. */
    uint8_t size;         /*< size of field 1,2,4, or 8. */
    uint8_t field_index;  /*< index of field inside the rule. */
    uint8_t input_index;  /*< 0-N input index. */
    uint32_t offset;      /*< offset to start of field. */
};
```

- type The field type is one of three choices:

  - _MASK - for fields such as IP addresses that have a value and a mask defining the number of relevant bits.
  - _RANGE - for fields such as ports that have a lower and upper value for the field.
  - _BITMASK - for fields such as protocol identifiers that have a value and a bit mask.
- size The size parameter defines the length of the field in bytes. Allowable values are 1, 2, 4, or 8 bytes. Note that due to the grouping of input bytes, 1 or 2 byte fields must be defined as consecutive fields that make up 4 consecutive input bytes. Also, it is best to define fields of 8 or more bytes as 4 byte fields so that the build processes can eliminate fields that are all wild.
- field_index A zero-based value that represents the position of the field inside the rule; 0 to N-1 for N fields.
- input_index As mentioned above, all input fields, except the very first one, must be in groups of 4 consecutive bytes. The input index specifies to which input group that field belongs to.
- offset The offset field defines the offset for the field. This is the offset from the beginning of the buffer parameter for the search.

For example, to define classification for the following IPv4 5-tuple structure:

```
struct ipv4_5tuple {
    uint8_t proto;
    uint32_t ip_src;
    uint32_t ip_dst;
    uint16_t port_src;
    uint16_t port_dst;
};
```

The following array of field definitions can be used:

```
struct rte_acl_field_def ipv4_defs[5] = {
    /* first input field - always one byte long. */
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint8_t),
        .field_index = 0,
        .input_index = 0,
        .offset = offsetof (struct ipv4_5tuple, proto),
    },

    /* next input field (IPv4 source address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 1,
        .input_index = 1,
       .offset = offsetof (struct ipv4_5tuple, ip_src),
    },

    /* next input field (IPv4 destination address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 2,
        .input_index = 2,
       .offset = offsetof (struct ipv4_5tuple, ip_dst),
    },
```

```
    /*
     * Next 2 fields (src & dst ports) form 4 consecutive bytes.
     * They share the same input index.
     */
    {
        .type = RTE_ACL_FIELD_TYPE_RANGE,
        .size = sizeof (uint16_t),
        .field_index = 3,
        .input_index = 3,
        .offset = offsetof (struct ipv4_5tuple, port_src),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_RANGE,
        .size = sizeof (uint16_t),
        .field_index = 4,
        .input_index = 3,
        .offset = offsetof (struct ipv4_5tuple, port_dst),
    },
};
```

A typical example of such an IPv4 5-tuple rule is a follows:

```
source addr/mask  destination addr/mask  source ports dest ports protocol/mask
192.168.1.0/24    192.168.2.31/32        0:65535      1234:1234  17/0xff
```

Any IPv4 packets with protocol ID 17 (UDP), source address 192.168.1.[0-255], destination address 192.168.2.31, source port [0-65535] and destination port 1234 matches the above rule.

To define classification for the IPv6 2-tuple: <protocol, IPv6 source address> over the following IPv6 header structure:

```
struct struct ipv6_hdr {
    uint32_t vtc_flow;     /* IP version, traffic class & flow label. */
    uint16_t payload_len;  /* IP packet length - includes sizeof(ip_header). */
    uint8_t proto;         /* Protocol, next header. */
    uint8_t hop_limits;    /* Hop limits. */
    uint8_t src_addr[16];  /* IP address of source host. */
    uint8_t dst_addr[16];  /* IP address of destination host(s). */
} __attribute__((__packed__));
```

The following array of field definitions can be used:

```
struct struct rte_acl_field_def ipv6_2tuple_defs[5] = {
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint8_t),
        .field_index = 0,
        .input_index = 0,
        .offset = offsetof (struct ipv6_hdr, proto),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 1,
        .input_index = 1,
        .offset = offsetof (struct ipv6_hdr, src_addr[0]),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
```

```
        .field_index = 2,
        .input_index = 2,
        .offset = offsetof (struct ipv6_hdr, src_addr[4]),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 3,
        .input_index = 3,
       .offset = offsetof (struct ipv6_hdr, src_addr[8]),
    },

    {
       .type = RTE_ACL_FIELD_TYPE_MASK,
       .size = sizeof (uint32_t),
       .field_index = 4,
       .input_index = 4,
       .offset = offsetof (struct ipv6_hdr, src_addr[12]),
    },
};
```

A typical example of such an IPv6 2-tuple rule is a follows:

```
source addr/mask                                 protocol/mask
2001:db8:1234:0000:0000:0000:0000:0000/48     6/0xff
```

Any IPv6 packets with protocol ID 6 (TCP), and source address inside the range [2001:db8:1234:0000:0000:0000:0000:0000 - 2001:db8:1234:ffff:ffff:ffff:ffff:ffff] matches the above rule.

In the following example the last element of the search key is 8-bit long. So it is a case where the 4 consecutive bytes of an input field are not fully occupied. The structure for the classification is:

```
struct acl_key {
    uint8_t ip_proto;
    uint32_t ip_src;
    uint32_t ip_dst;
    uint8_t tos;      /*< This is partially using a 32-bit input element */
};
```

The following array of field definitions can be used:

```
struct rte_acl_field_def ipv4_defs[4] = {
    /* first input field - always one byte long. */
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint8_t),
        .field_index = 0,
        .input_index = 0,
        .offset = offsetof (struct acl_key, ip_proto),
    },

    /* next input field (IPv4 source address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 1,
        .input_index = 1,
       .offset = offsetof (struct acl_key, ip_src),
    },

    /* next input field (IPv4 destination address) - 4 consecutive bytes. */
```

```
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 2,
        .input_index = 2,
       .offset = offsetof (struct acl_key, ip_dst),
    },

    /*
     * Next element of search key (Type of Service) is indeed 1 byte long.
     * Anyway we need to allocate all the 4 consecutive bytes for it.
     */
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint32_t), /* All the 4 consecutive bytes are allocated */
        .field_index = 3,
        .input_index = 3,
        .offset = offsetof (struct acl_key, tos),
    },
};
```

A typical example of such an IPv4 4-tuple rule is as follows:

```
source addr/mask  destination addr/mask  tos/mask protocol/mask
192.168.1.0/24    192.168.2.31/32        1/0xff   6/0xff
```

Any IPv4 packets with protocol ID 6 (TCP), source address 192.168.1.[0-255], destination address 192.168.2.31, ToS 1 matches the above rule.

When creating a set of rules, for each rule, additional information must be supplied also:

- **priority**: A weight to measure the priority of the rules (higher is better). If the input tuple matches more than one rule, then the rule with the higher priority is returned. Note that if the input tuple matches more than one rule and these rules have equal priority, it is undefined which rule is returned as a match. It is recommended to assign a unique priority for each rule.
- **category_mask**: Each rule uses a bit mask value to select the relevant category(s) for the rule. When a lookup is performed, the result for each category is returned. This effectively provides a "parallel lookup" by enabling a single search to return multiple results if, for example, there were four different sets of ACL rules, one for access control, one for routing, and so on. Each set could be assigned its own category and by combining them into a single database, one lookup returns a result for each of the four sets.
- **userdata**: A user-defined field that could be any value except zero. For each category, a successful match returns the userdata field of the highest priority matched rule.

Note: When adding new rules into an ACL context, all fields must be in host byte order (LSB). When the search is performed for an input tuple, all fields in that tuple must be in network byte order (MSB).

23.1.2 RT memory size limit

Build phase (rte_acl_build()) creates for a given set of rules internal structure for further runtime traversal. With current implementation it is a set of multi-bit tries (with stride == 8). Depending on the rules set, that could consume significant amount of memory. In attempt to conserve some space ACL build process tries to split the given rule-set into several non-intersecting subsets and construct a separate trie for each of them. Depending on the rule-set,

it might reduce RT memory requirements but might increase classification time. There is a possibility at build-time to specify maximum memory limit for internal RT structures for given AC context. It could be done via **max_size** field of the **rte_acl_config** structure. Setting it to the value greater than zero, instructs rte_acl_build() to:

- attempt to minimize number of tries in the RT table, but
- make sure that size of RT table wouldn't exceed given value.

Setting it to zero makes rte_acl_build() to use the default behavior: try to minimize size of the RT structures, but doesn't expose any hard limit on it.

That gives the user the ability to decisions about performance/space trade-off. For example:

```
struct rte_acl_ctx * acx;
struct rte_acl_config cfg;
int ret;

/*
 * assuming that acx points to already created and
 * populated with rules AC context and cfg filled properly.
 */

 /* try to build AC context, with RT structures less then 8MB. */
 cfg.max_size = 0x800000;
 ret = rte_acl_build(acx, &cfg);

 /*
  * RT structures can't fit into 8MB for given context.
  * Try to build without exposing any hard limit.
  */
 if (ret == -ERANGE) {
    cfg.max_size = 0;
    ret = rte_acl_build(acx, &cfg);
 }
```

23.1.3 Classification methods

After rte_acl_build() over given AC context has finished successfully, it can be used to perform classification - search for a rule with highest priority over the input data. There are several implementations of classify algorithm:

- **RTE_ACL_CLASSIFY_SCALAR**: generic implementation, doesn't require any specific HW support.
- **RTE_ACL_CLASSIFY_SSE**: vector implementation, can process up to 8 flows in parallel. Requires SSE 4.1 support.
- **RTE_ACL_CLASSIFY_AVX2**: vector implementation, can process up to 16 flows in parallel. Requires AVX2 support.

It is purely a runtime decision which method to choose, there is no build-time difference. All implementations operates over the same internal RT structures and use similar principles. The main difference is that vector implementations can manually exploit IA SIMD instructions and process several input data flows in parallel. At startup ACL library determines the highest available classify method for the given platform and sets it as default one. Though the user has an ability to override the default classifier function for a given ACL context or perform particular search using non-default classify method. In that case it is user responsibility to make sure that given platform supports selected classify implementation.

23.2 Application Programming Interface (API) Usage

Note: For more details about the Access Control API, please refer to the *DPDK API Reference*.

The following example demonstrates IPv4, 5-tuple classification for rules defined above with multiple categories in more detail.

23.2.1 Classify with Multiple Categories

```
struct rte_acl_ctx * acx;
struct rte_acl_config cfg;
int ret;

/* define a structure for the rule with up to 5 fields. */

RTE_ACL_RULE_DEF(acl_ipv4_rule, RTE_DIM(ipv4_defs));

/* AC context creation parameters. */

struct rte_acl_param prm = {
    .name = "ACL_example",
    .socket_id = SOCKET_ID_ANY,
    .rule_size = RTE_ACL_RULE_SZ(RTE_DIM(ipv4_defs)),

    /* number of fields per rule. */

    .max_rule_num = 8, /* maximum number of rules in the AC context. */
};

struct acl_ipv4_rule acl_rules[] = {

    /* matches all packets traveling to 192.168.0.0/16, applies for categories: 0,1 */
    {
        .data = {.userdata = 1, .category_mask = 3, .priority = 1},

        /* destination IPv4 */
        .field[2] = {.value.u32 = IPv4(192,168,0,0),. mask_range.u32 = 16,},

        /* source port */
        .field[3] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},

        /* destination port */
       .field[4] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},
    },

    /* matches all packets traveling to 192.168.1.0/24, applies for categories: 0 */
    {
        .data = {.userdata = 2, .category_mask = 1, .priority = 2},

        /* destination IPv4 */
        .field[2] = {.value.u32 = IPv4(192,168,1,0),. mask_range.u32 = 24,},

        /* source port */
        .field[3] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},

        /* destination port */
        .field[4] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},
    },
```

```
    /* matches all packets traveling from 10.1.1.1, applies for categories: 1 */
    {
        .data = {.userdata = 3, .category_mask = 2, .priority = 3},

        /* source IPv4 */
        .field[1] = {.value.u32 = IPv4(10,1,1,1),. mask_range.u32 = 32,},

        /* source port */
        .field[3] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},

        /* destination port */
        .field[4] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},
    },

};


/* create an empty AC context  */

if ((acx = rte_acl_create(&prm)) == NULL) {

    /* handle context create failure. */

}

/* add rules to the context */

ret = rte_acl_add_rules(acx, acl_rules, RTE_DIM(acl_rules));
if (ret != 0) {
   /* handle error at adding ACL rules. */
}

/* prepare AC build config. */

cfg.num_categories = 2;
cfg.num_fields = RTE_DIM(ipv4_defs);

memcpy(cfg.defs, ipv4_defs, sizeof (ipv4_defs));

/* build the runtime structures for added rules, with 2 categories. */

ret = rte_acl_build(acx, &cfg);
if (ret != 0) {
   /* handle error at build runtime structures for ACL context. */
}
```

For a tuple with source IP address: 10.1.1.1 and destination IP address: 192.168.1.15, once the following lines are executed:

```
uint32_t results[4]; /* make classify for 4 categories. */

rte_acl_classify(acx, data, results, 1, 4);
```

then the results[] array contains:

```
results[4] = {2, 3, 0, 0};
```

- For category 0, both rules 1 and 2 match, but rule 2 has higher priority, therefore results[0] contains the userdata for rule 2.
- For category 1, both rules 1 and 3 match, but rule 3 has higher priority, therefore results[1] contains the userdata for rule 3.

- For categories 2 and 3, there are no matches, so results[2] and results[3] contain zero, which indicates that no matches were found for those categories.

For a tuple with source IP address: 192.168.1.1 and destination IP address: 192.168.2.11, once the following lines are executed:

```
uint32_t results[4]; /* make classify by 4 categories. */

rte_acl_classify(acx, data, results, 1, 4);
```

the results[] array contains:

```
results[4] = {1, 1, 0, 0};
```

- For categories 0 and 1, only rule 1 matches.
- For categories 2 and 3, there are no matches.

For a tuple with source IP address: 10.1.1.1 and destination IP address: 201.212.111.12, once the following lines are executed:

```
uint32_t results[4]; /* make classify by 4 categories. */
rte_acl_classify(acx, data, results, 1, 4);
```

the results[] array contains:

```
results[4] = {0, 3, 0, 0};
```

- For category 1, only rule 3 matches.
- For categories 0, 2 and 3, there are no matches.

CHAPTER

TWENTYFOUR

PACKET FRAMEWORK

24.1 Design Objectives

The main design objectives for the DPDK Packet Framework are:

- Provide standard methodology to build complex packet processing pipelines. Provide reusable and extensible templates for the commonly used pipeline functional blocks;
- Provide capability to switch between pure software and hardware-accelerated implementations for the same pipeline functional block;
- Provide the best trade-off between flexibility and performance. Hardcoded pipelines usually provide the best performance, but are not flexible, while developing flexible frameworks is never a problem, but performance is usually low;
- Provide a framework that is logically similar to Open Flow.

24.2 Overview

Packet processing applications are frequently structured as pipelines of multiple stages, with the logic of each stage glued around a lookup table. For each incoming packet, the table defines the set of actions to be applied to the packet, as well as the next stage to send the packet to.

The DPDK Packet Framework minimizes the development effort required to build packet processing pipelines by defining a standard methodology for pipeline development, as well as providing libraries of reusable templates for the commonly used pipeline blocks.

The pipeline is constructed by connecting the set of input ports with the set of output ports through the set of tables in a tree-like topology. As result of lookup operation for the current packet in the current table, one of the table entries (on lookup hit) or the default table entry (on lookup miss) provides the set of actions to be applied on the current packet, as well as the next hop for the packet, which can be either another table, an output port or packet drop.

An example of packet processing pipeline is presented in Fig. 24.1:

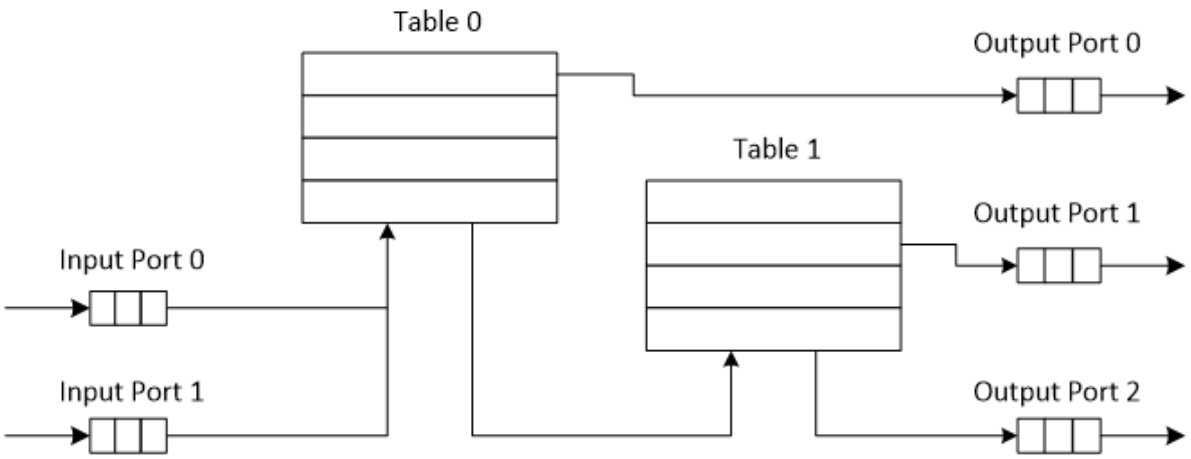


Fig. 24.1: Example of Packet Processing Pipeline where Input Ports 0 and 1 are Connected with Output Ports 0, 1 and 2 through Tables 0 and 1

24.3 Port Library Design

24.3.1 Port Types

Table 24.1 is a non-exhaustive list of ports that can be implemented with the Packet Framework.

Table 24.1: Port Types

#	Port type	Description
1	SW ring	SW circular buffer used for message passing between the application threads. Uses the DPDK rte_ring primitive. Expected to be the most commonly used type of port.
2	HW ring	Queue of buffer descriptors used to interact with NIC, switch or accelerator ports. For NIC ports, it uses the DPDK rte_eth_rx_queue or rte_eth_tx_queue primitives.
3	IP re-assembly	Input packets are either IP fragments or complete IP datagrams. Output packets are complete IP datagrams.
4	IP fragmentation	Input packets are jumbo (IP datagrams with length bigger than MTU) or non-jumbo packets. Output packets are non-jumbo packets.
5	Traffic manager	Traffic manager attached to a specific NIC output port, performing congestion management and hierarchical scheduling according to pre-defined SLAs.
6	KNI	Send/receive packets to/from Linux kernel space.
7	Source	Input port used as packet generator. Similar to Linux kernel /dev/zero character device.
8	Sink	Output port used to drop all input packets. Similar to Linux kernel /dev/null character device.

24.3.2 Port Interface

Each port is unidirectional, i.e. either input port or output port. Each input/output port is required to implement an abstract interface that defines the initialization and run-time operation of the port. The port abstract interface is described in.

Table 24.2: 20 Port Abstract Interface

#	Port Operation	Description
1	Create	Create the low-level port object (e.g. queue). Can internally allocate memory.
2	Free	Free the resources (e.g. memory) used by the low-level port object.
3	RX	Read a burst of input packets. Non-blocking operation. Only defined for input ports.
4	TX	Write a burst of input packets. Non-blocking operation. Only defined for output ports.
5	Flush	Flush the output buffer. Only defined for output ports.

24.4 Table Library Design

24.4.1 Table Types

Table 24.3 is a non-exhaustive list of types of tables that can be implemented with the Packet Framework.

Table 24.3: Table Types

#	Table Type	Description
1	Hash table	Lookup key is n-tuple based. Typically, the lookup key is hashed to produce a signature that is used to identify a bucket of entries where the lookup key is searched next. The signature associated with the lookup key of each input packet is either read from the packet descriptor (pre-computed signature) or computed at table lookup time. The table lookup, add entry and delete entry operations, as well as any other pipeline block that pre-computes the signature all have to use the same hashing algorithm to generate the signature. Typically used to implement flow classification tables, ARP caches, routing table for tunnelling protocols, etc.
2	Longest Prefix Match (LPM)	Lookup key is the IP address. Each table entries has an associated IP prefix (IP and depth). The table lookup operation selects the IP prefix that is matched by the lookup key; in case of multiple matches, the entry with the longest prefix depth wins. Typically used to implement IP routing tables.
3	Access Control List (ACLs)	Lookup key is 7-tuple of two VLAN/MPLS labels, IP destination address, IP source addresses, L4 protocol, L4 destination port, L4 source port. Each table entry has an associated ACL and priority. The ACL contains bit masks for the VLAN/MPLS labels, IP prefix for IP destination address, IP prefix for IP source addresses, L4 protocol and bitmask, L4 destination port and bit mask, L4 source port and bit mask. The table lookup operation selects the ACL that is matched by the lookup key; in case of multiple matches, the entry with the highest priority wins. Typically used to implement rule databases for firewalls, etc.
4	Pattern matching search	Lookup key is the packet payload. Table is a database of patterns, with each pattern having a priority assigned. The table lookup operation selects the patterns that is matched by the input packet; in case of multiple matches, the matching pattern with the highest priority wins.
5	Array	Lookup key is the table entry index itself.

24.4.2 Table Interface

Each table is required to implement an abstract interface that defines the initialization and run-time operation of the table. The table abstract interface is described in Table 24.4.

Table 24.4: Table Abstract Interface

#	Table operation	Description
1	Create	Create the low-level data structures of the lookup table. Can internally allocate memory.
2	Free	Free up all the resources used by the lookup table.
3	Add entry	Add new entry to the lookup table.
4	Delete entry	Delete specific entry from the lookup table.
5	Lookup	Look up a burst of input packets and return a bit mask specifying the result of the lookup operation for each packet: a set bit signifies lookup hit for the corresponding packet, while a cleared bit a lookup miss. For each lookup hit packet, the lookup operation also returns a pointer to the table entry that was hit, which contains the actions to be applied on the packet and any associated metadata. For each lookup miss packet, the actions to be applied on the packet and any associated metadata are specified by the default table entry preconfigured for lookup miss.

24.4.3 Hash Table Design

Hash Table Overview

Hash tables are important because the key lookup operation is optimized for speed: instead of having to linearly search the lookup key through all the keys in the table, the search is limited to only the keys stored in a single table bucket.

Associative Arrays

An associative array is a function that can be specified as a set of (key, value) pairs, with each key from the possible set of input keys present at most once. For a given associative array, the possible operations are:

1. *add (key, value)*: When no value is currently associated with *key*, then the (key, *value*) association is created. When *key* is already associated value *value0*, then the association (*key*, *value0*) is removed and association *(key, value)* is created;
2. *delete key*: When no value is currently associated with *key*, this operation has no effect. When *key* is already associated *value*, then association *(key, value)* is removed;
3. *lookup key*: When no value is currently associated with *key*, then this operation returns void value (lookup miss). When *key* is associated with *value*, then this operation returns *value*. The *(key, value)* association is not changed.

The matching criterion used to compare the input key against the keys in the associative array is *exact match*, as the key size (number of bytes) and the key value (array of bytes) have to match exactly for the two keys under comparison.

Hash Function

A hash function deterministically maps data of variable length (key) to data of fixed size (hash value or key signature). Typically, the size of the key is bigger than the size of the key signature. The hash function basically compresses a long key into a short signature. Several keys can share the same signature (collisions).

High quality hash functions have uniform distribution. For large number of keys, when dividing the space of signature values into a fixed number of equal intervals (buckets), it is desirable to have the key signatures evenly distributed across these intervals (uniform distribution), as opposed to most of the signatures going into only a few of the intervals and the rest of the intervals being largely unused (non-uniform distribution).

Hash Table

A hash table is an associative array that uses a hash function for its operation. The reason for using a hash function is to optimize the performance of the lookup operation by minimizing the number of table keys that have to be compared against the input key.

Instead of storing the (key, value) pairs in a single list, the hash table maintains multiple lists (buckets). For any given key, there is a single bucket where that key might exist, and this bucket is uniquely identified based on the key signature. Once the key signature is computed and the hash table bucket identified, the key is either located in this bucket or it is not present in the hash table at all, so the key search can be narrowed down from the full set of keys currently in the table to just the set of keys currently in the identified table bucket.

The performance of the hash table lookup operation is greatly improved, provided that the table keys are evenly distributed among the hash table buckets, which can be achieved by using a hash function with uniform distribution. The rule to map a key to its bucket can simply be to use the key signature (modulo the number of table buckets) as the table bucket ID:

bucket_id = f_hash(key) % n_buckets;

By selecting the number of buckets to be a power of two, the modulo operator can be replaced by a bitwise AND logical operation:

bucket_id = f_hash(key) & (n_buckets - 1);

considering *n_bits* as the number of bits set in *bucket_mask = n_buckets - 1*, this means that all the keys that end up in the same hash table bucket have the lower *n_bits* of their signature identical. In order to reduce the number of keys in the same bucket (collisions), the number of hash table buckets needs to be increased.

In packet processing context, the sequence of operations involved in hash table operations is described in Fig. 24.2:

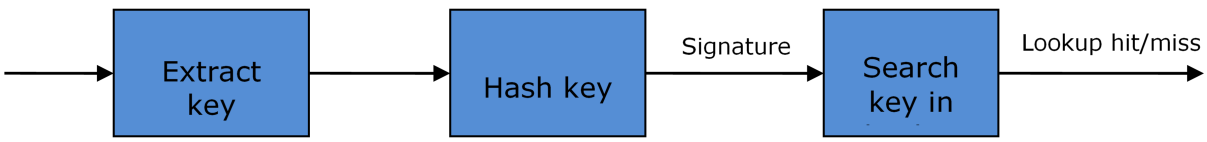


Fig. 24.2: Sequence of Steps for Hash Table Operations in a Packet Processing Context

Hash Table Use Cases

Flow Classification

Description: The flow classification is executed at least once for each input packet. This operation maps each incoming packet against one of the known traffic flows in the flow database that typically contains millions of flows.

Hash table name: Flow classification table

Number of keys: Millions

Key format: n-tuple of packet fields that uniquely identify a traffic flow/connection. Example: DiffServ 5-tuple of (Source IP address, Destination IP address, L4 protocol, L4 protocol source port, L4 protocol destination port). For IPv4 protocol and L4 protocols like TCP, UDP or SCTP, the size of the DiffServ 5-tuple is 13 bytes, while for IPv6 it is 37 bytes.

Key value (key data): actions and action meta-data describing what processing to be applied for the packets of the current flow. The size of the data associated with each traffic flow can vary from 8 bytes to kilobytes.

Address Resolution Protocol (ARP)

Description: Once a route has been identified for an IP packet (so the output interface and the IP address of the next hop station are known), the MAC address of the next hop station is needed in order to send this packet onto the next leg of the journey towards its destination (as identified by its destination IP address). The MAC address of the next hop station becomes the destination MAC address of the outgoing Ethernet frame.

Hash table name: ARP table

Number of keys: Thousands

Key format: The pair of (Output interface, Next Hop IP address), which is typically 5 bytes for IPv4 and 17 bytes for IPv6.

Key value (key data): MAC address of the next hop station (6 bytes).

Hash Table Types

Table 24.5 lists the hash table configuration parameters shared by all different hash table types.

Table 24.5: Configuration Parameters Common for All Hash Table Types

#	Parameter	Details
1	Key size	Measured as number of bytes. All keys have the same size.
2	Key value (key data) size	Measured as number of bytes.
3	Number of buckets	Needs to be a power of two.
4	Maximum number of keys	Needs to be a power of two.
5	Hash function	Examples: jhash, CRC hash, etc.
6	Hash function seed	Parameter to be passed to the hash function.
7	Key offset	Offset of the lookup key byte array within the packet meta-data stored in the packet buffer.

Bucket Full Problem

On initialization, each hash table bucket is allocated space for exactly 4 keys. As keys are added to the table, it can happen that a given bucket already has 4 keys when a new key has to be added to this bucket. The possible options are:

1. **Least Recently Used (LRU) Hash Table.** One of the existing keys in the bucket is deleted and the new key is added in its place. The number of keys in each bucket never grows bigger than 4. The logic to pick the key to be dropped from the bucket is LRU. The hash table lookup operation maintains the order in which the keys in the same bucket are hit, so every time a key is hit, it becomes the new Most Recently Used (MRU) key, i.e. the last candidate for drop. When a key is added to the bucket, it also becomes the new MRU key. When a key needs to be picked and dropped, the first candidate for drop, i.e. the current LRU key, is always picked. The LRU logic requires maintaining specific data structures per each bucket.
2. **Extendable Bucket Hash Table.** The bucket is extended with space for 4 more keys. This is done by allocating additional memory at table initialization time, which is used to create a pool of free keys (the size of this pool is configurable and always a multiple of 4). On key add operation, the allocation of a group of 4 keys only happens successfully within the limit of free keys, otherwise the key add operation fails. On key delete operation, a group of 4 keys is freed back to the pool of free keys when the key to be deleted is the only key that was used within its group of 4 keys at that time. On key lookup operation, if the current bucket is in extended state and a match is not found in the first group of 4 keys, the search continues beyond the first group of 4 keys, potentially until all keys in this bucket are examined. The extendable bucket logic requires maintaining specific data structures per table and per each bucket.

Table 24.6: Configuration Parameters Specific to Extendable Bucket Hash Table

#	Parameter	Details
1	Number of additional keys	Needs to be a power of two, at least equal to 4.

Signature Computation

The possible options for key signature computation are:

1. **Pre-computed key signature.** The key lookup operation is split between two CPU cores. The first CPU core (typically the CPU core that performs packet RX) extracts the key from the input packet, computes the key signature and saves both the key and the key signature in the packet buffer as packet meta-data. The second CPU core reads both the key and the key signature from the packet meta-data and performs the bucket search step of the key lookup operation.
2. **Key signature computed on lookup ("do-sig" version).** The same CPU core reads the key from the packet meta-data, uses it to compute the key signature and also performs the bucket search step of the key lookup operation.

Table 24.7: Configuration Parameters Specific to Pre-computed Key Signature Hash Table

#	Parameter	Details
1	Signature offset	Offset of the pre-computed key signature within the packet meta-data.

Key Size Optimized Hash Tables

For specific key sizes, the data structures and algorithm of key lookup operation can be specially handcrafted for further performance improvements, so following options are possible:

1. **Implementation supporting configurable key size.**
2. **Implementation supporting a single key size.** Typical key sizes are 8 bytes and 16 bytes.

Bucket Search Logic for Configurable Key Size Hash Tables

The performance of the bucket search logic is one of the main factors influencing the performance of the key lookup operation. The data structures and algorithm are designed to make the best use of Intel CPU architecture resources like: cache memory space, cache memory bandwidth, external memory bandwidth, multiple execution units working in parallel, out of order instruction execution, special CPU instructions, etc.

The bucket search logic handles multiple input packets in parallel. It is built as a pipeline of several stages (3 or 4), with each pipeline stage handling two different packets from the burst of input packets. On each pipeline iteration, the packets are pushed to the next pipeline stage: for the 4-stage pipeline, two packets (that just completed stage 3) exit the pipeline, two packets (that just completed stage 2) are now executing stage 3, two packets (that just completed stage 1) are now executing stage 2, two packets (that just completed stage 0) are now executing stage 1 and two packets (next two packets to read from the burst of input packets) are entering the pipeline to execute stage 0. The pipeline iterations continue until all packets from the burst of input packets execute the last stage of the pipeline.

The bucket search logic is broken into pipeline stages at the boundary of the next memory access. Each pipeline stage uses data structures that are stored (with high probability) into the L1 or L2 cache memory of the current CPU core and breaks just before the next memory access required by the algorithm. The current pipeline stage finalizes by prefetching the data structures required by the next pipeline stage, so given enough time for the prefetch to complete, when the next pipeline stage eventually gets executed for the same packets, it will read the data structures it needs from L1 or L2 cache memory and thus avoid the significant penalty incurred by L2 or L3 cache memory miss.

By prefetching the data structures required by the next pipeline stage in advance (before they are used) and switching to executing another pipeline stage for different packets, the number of L2 or L3 cache memory misses is greatly reduced, hence one of the main reasons for improved performance. This is because the cost of L2/L3 cache memory miss on memory read accesses is high, as usually due to data dependency between instructions, the CPU execution units have to stall until the read operation is completed from L3 cache memory or external DRAM memory. By using prefetch instructions, the latency of memory read accesses is hidden, provided that it is preformed early enough before the respective data structure is actually used.

By splitting the processing into several stages that are executed on different packets (the packets from the input burst are interlaced), enough work is created to allow the prefetch instructions to complete successfully (before the prefetched data structures are actually accessed) and also the data dependency between instructions is loosened. For example, for the 4-stage pipeline, stage 0 is executed on packets 0 and 1 and then, before same packets 0 and 1 are used (i.e. before stage 1 is executed on packets 0 and 1), different packets are used: packets

2 and 3 (executing stage 1), packets 4 and 5 (executing stage 2) and packets 6 and 7 (executing stage 3). By executing useful work while the data structures are brought into the L1 or L2 cache memory, the latency of the read memory accesses is hidden. By increasing the gap between two consecutive accesses to the same data structure, the data dependency between instructions is loosened; this allows making the best use of the super-scalar and out-of-order execution CPU architecture, as the number of CPU core execution units that are active (rather than idle or stalled due to data dependency constraints between instructions) is maximized.

The bucket search logic is also implemented without using any branch instructions. This avoids the important cost associated with flushing the CPU core execution pipeline on every instance of branch misprediction.

Configurable Key Size Hash Table

Fig. 24.3, Table 24.8 and Table 24.9 detail the main data structures used to implement configurable key size hash tables (either LRU or extendable bucket, either with pre-computed signature or "do-sig").

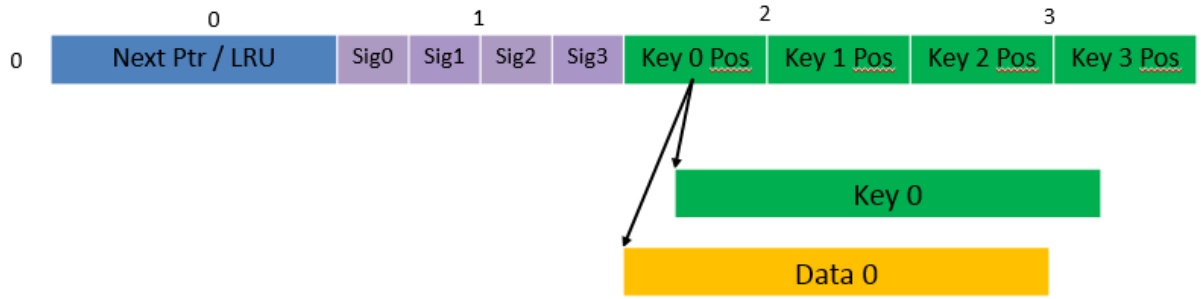


Fig. 24.3: Data Structures for Configurable Key Size Hash Tables

Table 24.8: Main Large Data Structures (Arrays) used for Configurable Key Size Hash Tables

#	Array name	Number of entries	Entry size (bytes)	Description
1	Bucket array	n_buckets (configurable)	32	Buckets of the hash table.
2	Bucket extensions array	n_buckets_ext (configurable)	32	This array is only created for extendable bucket tables.
3	Key array	n_keys	key_size (configurable)	Keys added to the hash table.
4	Data array	n_keys	entry_size (configurable)	Key values (key data) associated with the hash table keys.

Table 24.9: Field Description for Bucket Array Entry (Configurable Key Size Hash Tables)

#	Field name	Field size (bytes)	Description
1	Next Ptr/LRU	8	For LRU tables, this fields represents the LRU list for the current bucket stored as array of 4 entries of 2 bytes each. Entry 0 stores the index (0 .. 3) of the MRU key, while entry 3 stores the index of the LRU key. For extendable bucket tables, this field represents the next pointer (i.e. the pointer to the next group of 4 keys linked to the current bucket). The next pointer is not NULL if the bucket is currently extended or NULL otherwise. To help the branchless implementation, bit 0 (least significant bit) of this field is set to 1 if the next pointer is not NULL and to 0 otherwise.
2	Sig[0 .. 3]	4 x 2	If key X (X = 0 .. 3) is valid, then sig X bits 15 .. 1 store the most significant 15 bits of key X signature and sig X bit 0 is set to 1. If key X is not valid, then sig X is set to zero.
3	Key Pos [0 .. 3]	4 x 4	If key X is valid (X = 0 .. 3), then Key Pos X represents the index into the key array where key X is stored, as well as the index into the data array where the value associated with key X is stored. If key X is not valid, then the value of Key Pos X is undefined.

Fig. 24.4 and Table 24.10 detail the bucket search pipeline stages (either LRU or extendable bucket, either with pre-computed signature or "do-sig"). For each pipeline stage, the described operations are applied to each of the two packets handled by that stage.

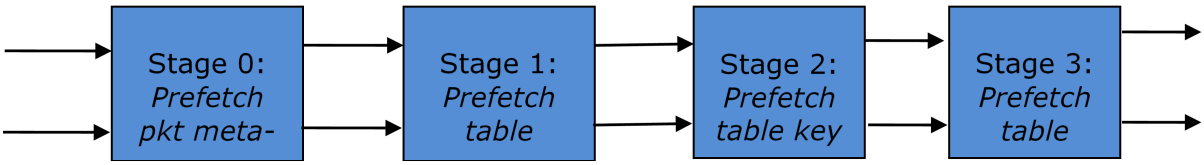


Fig. 24.4: Bucket Search Pipeline for Key Lookup Operation (Configurable Key Size Hash Tables)

Table 24.10: Description of the Bucket Search Pipeline Stages (Configurable Key Size Hash Tables)

#	Stage name	Description
0	Prefetch packet meta-data	Select next two packets from the burst of input packets. Prefetch packet meta-data containing the key and key signature.
1	Prefetch table bucket	Read the key signature from the packet meta-data (for extendable bucket hash tables) or read the key from the packet meta-data and compute key signature (for LRU tables). Identify the bucket ID using the key signature. Set bit 0 of the signature to 1 (to match only signatures of valid keys from the table). Prefetch the bucket.
2	Prefetch table key	Read the key signatures from the bucket. Compare the signature of the input key against the 4 key signatures from the packet. As result, the following is obtained: *match* = equal to TRUE if there was at least one signature match and to FALSE in the case of no signature match; *match_many* = equal to TRUE is there were more than one signature matches (can be up to 4 signature matches in the worst case scenario) and to FALSE otherwise; *match_pos* = the index of the first key that produced signature match (only valid if match is true). For extendable bucket hash tables only, set *match_many* to TRUE if next pointer is valid. Prefetch the bucket key indicated by *match_pos* (even if *match_pos* does not point to valid key valid).
3	Prefetch table data	Read the bucket key indicated by *match_pos*. Compare the bucket key against the input key. As result, the following is obtained: *match_key* = equal to TRUE if the two keys match and to FALSE otherwise. Report input key as lookup hit only when both *match* and *match_key* are equal to TRUE and as lookup miss otherwise. For LRU tables only, use branchless logic to update the bucket LRU list (the current key becomes the new MRU) only on lookup hit. Prefetch the key value (key data) associated with the current key (to avoid branches, this is done on both lookup hit and miss).

Additional notes:

1. The pipelined version of the bucket search algorithm is executed only if there are at least 7 packets in the burst of input packets. If there are less than 7 packets in the burst of input packets, a non-optimized implementation of the bucket search algorithm is executed.

2. Once the pipelined version of the bucket search algorithm has been executed for all the packets in the burst of input packets, the non-optimized implementation of the bucket search algorithm is also executed for any packets that did not produce a lookup hit, but have the *match_many* flag set. As result of executing the non-optimized version, some of these packets may produce a lookup hit or lookup miss. This does not impact the performance of the key lookup operation, as the probability of matching more than one signature in the same group of 4 keys or of having the bucket in extended state (for extendable bucket hash tables only) is relatively small.

Key Signature Comparison Logic

The key signature comparison logic is described in Table 24.11.

Table 24.11: Lookup Tables for Match, Match_Many and Match_Pos

#	mask	match (1 bit)	match_many (1 bit)	match_pos (2 bits)
0	0000	0	0	00
1	0001	1	0	00
2	0010	1	0	01
3	0011	1	1	00
4	0100	1	0	10
5	0101	1	1	00
6	0110	1	1	01
7	0111	1	1	00
8	1000	1	0	11
9	1001	1	1	00
10	1010	1	1	01
11	1011	1	1	00
12	1100	1	1	10
13	1101	1	1	00
14	1110	1	1	01
15	1111	1	1	00

The input *mask* hash bit X (X = 0 .. 3) set to 1 if input signature is equal to bucket signature X and set to 0 otherwise. The outputs *match*, *match_many* and *match_pos* are 1 bit, 1 bit and 2 bits in size respectively and their meaning has been explained above.

As displayed in Table 24.12, the lookup tables for *match* and *match_many* can be collapsed into a single 32-bit value and the lookup table for *match_pos* can be collapsed into a 64-bit value. Given the input *mask*, the values for *match*, *match_many* and *match_pos* can be obtained by indexing their respective bit array to extract 1 bit, 1 bit and 2 bits respectively with branchless logic.

Table 24.12: Collapsed Lookup Tables for Match, Match_Many and Match_Pos

	Bit array	Hexadecimal value
match	1111_1111_1111_1110	0xFFFELLU
match_many	1111_1110_1110_1000	0xFEE8LLU
match_pos	0001_0010_0001_0011__0001_0010_0001_0000	0x12131210LLU

The pseudo-code for match, match_many and match_pos is:

```
match = (0xFFFELLU >> mask) & 1;

match_many = (0xFEE8LLU >> mask) & 1;

match_pos = (0x12131210LLU >> (mask << 1)) & 3;
```

Single Key Size Hash Tables

Fig. 24.5, Fig. 24.6, Table 24.13 and Table 24.14 detail the main data structures used to implement 8-byte and 16-byte key hash tables (either LRU or extendable bucket, either with pre-computed signature or "do-sig").

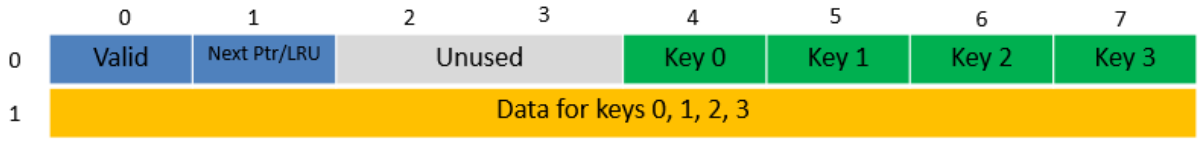


Fig. 24.5: Data Structures for 8-byte Key Hash Tables

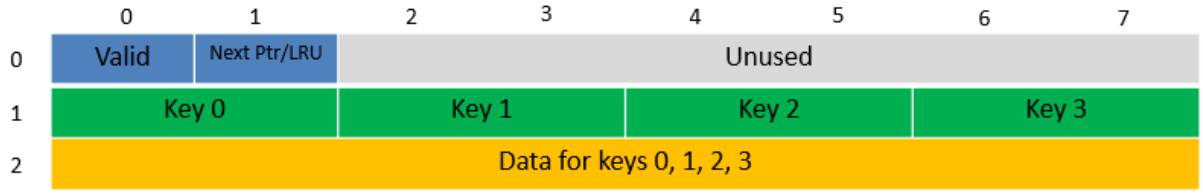


Fig. 24.6: Data Structures for 16-byte Key Hash Tables

Table 24.13: Main Large Data Structures (Arrays) used for 8-byte and 16-byte Key Size Hash Tables

#	Array name	Number of entries	Entry size (bytes)	Description
1	Bucket array	n_buckets (configurable)	*8-byte key size:* 64 + 4 x entry_size *16-byte key size:* 128 + 4 x entry_size	Buckets of the hash table.
2	Bucket extensions array	n_buckets_ext (configurable)	*8-byte key size:* 64 + 4 x entry_size *16-byte key size:* 128 + 4 x entry_size	This array is only created for extendable bucket tables.

Table 24.14: Field Description for Bucket Array Entry (8-byte and 16-byte Key Hash Tables)

#	Field name	Field size (bytes)	Description
1	Valid	8	Bit X (X = 0 .. 3) is set to 1 if key X is valid or to 0 otherwise. Bit 4 is only used for extendable bucket tables to help with the implementation of the branchless logic. In this case, bit 4 is set to 1 if next pointer is valid (not NULL) or to 0 otherwise.
2	Next Ptr/LRU	8	For LRU tables, this fields represents the LRU list for the current bucket stored as array of 4 entries of 2 bytes each. Entry 0 stores the index (0 .. 3) of the MRU key, while entry 3 stores the index of the LRU key. For extendable bucket tables, this field represents the next pointer (i.e. the pointer to the next group of 4 keys linked to the current bucket). The next pointer is not NULL if the bucket is currently extended or NULL otherwise.
3	Key [0 .. 3]	4 x key_size	Full keys.
4	Data [0 .. 3]	4 x entry_size	Full key values (key data) associated with keys 0 .. 3.

and detail the bucket search pipeline used to implement 8-byte and 16-byte key hash tables (either LRU or extendable bucket, either with pre-computed signature or "do-sig"). For each pipeline stage, the described operations are applied to each of the two packets handled by that stage.

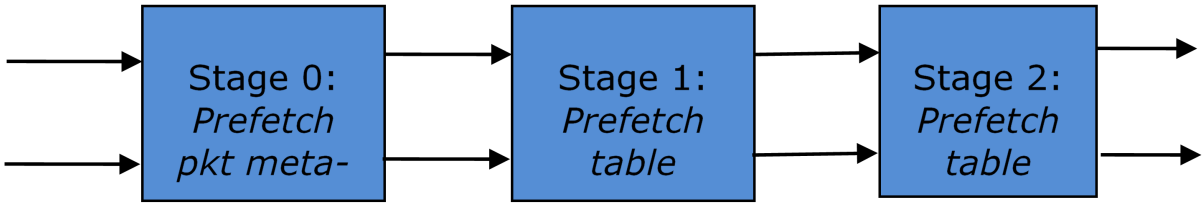


Fig. 24.7: Bucket Search Pipeline for Key Lookup Operation (Single Key Size Hash Tables)

Table 24.15: Description of the Bucket Search Pipeline Stages (8-byte and 16-byte Key Hash Tables)

#	Stage name	Description
0	Prefetch packet meta-data	1. Select next two packets from the burst of input packets. 2. Prefetch packet meta-data containing the key and key signature.
1	Prefetch table bucket	1. Read the key signature from the packet meta-data (for extendable bucket hash tables) or read the key from the packet meta-data and compute key signature (for LRU tables). 2. Identify the bucket ID using the key signature. 3. Prefetch the bucket.
2	Prefetch table data	1. Read the bucket. 2. Compare all 4 bucket keys against the input key. 3. Report input key as lookup hit only when a match is identified (more than one key match is not possible) 4. For LRU tables only, use branchless logic to update the bucket LRU list (the current key becomes the new MRU) only on lookup hit. 5. Prefetch the key value (key data) associated with the matched key (to avoid branches, this is done on both lookup hit and miss).

Additional notes:

1. The pipelined version of the bucket search algorithm is executed only if there are at least 5 packets in the burst of input packets. If there are less than 5 packets in the burst of input

packets, a non-optimized implementation of the bucket search algorithm is executed.

2. For extendable bucket hash tables only, once the pipelined version of the bucket search algorithm has been executed for all the packets in the burst of input packets, the non-optimized implementation of the bucket search algorithm is also executed for any packets that did not produce a lookup hit, but have the bucket in extended state. As result of executing the non-optimized version, some of these packets may produce a lookup hit or lookup miss. This does not impact the performance of the key lookup operation, as the probability of having the bucket in extended state is relatively small.

24.5 Pipeline Library Design

A pipeline is defined by:

1. The set of input ports;
2. The set of output ports;
3. The set of tables;
4. The set of actions.

The input ports are connected with the output ports through tree-like topologies of interconnected tables. The table entries contain the actions defining the operations to be executed on the input packets and the packet flow within the pipeline.

24.5.1 Connectivity of Ports and Tables

To avoid any dependencies on the order in which pipeline elements are created, the connectivity of pipeline elements is defined after all the pipeline input ports, output ports and tables have been created.

General connectivity rules:

1. Each input port is connected to a single table. No input port should be left unconnected;
2. The table connectivity to other tables or to output ports is regulated by the next hop actions of each table entry and the default table entry. The table connectivity is fluid, as the table entries and the default table entry can be updated during run-time.
    - A table can have multiple entries (including the default entry) connected to the same output port. A table can have different entries connected to different output ports. Different tables can have entries (including default table entry) connected to the same output port.
    - A table can have multiple entries (including the default entry) connected to another table, in which case all these entries have to point to the same table. This constraint is enforced by the API and prevents tree-like topologies from being created (allowing table chaining only), with the purpose of simplifying the implementation of the pipeline run-time execution engine.

24.5.2 Port Actions

Port Action Handler

An action handler can be assigned to each input/output port to define actions to be executed on each input packet that is received by the port. Defining the action handler for a specific input/output port is optional (i.e. the action handler can be disabled).

For input ports, the action handler is executed after RX function. For output ports, the action handler is executed before the TX function.

The action handler can decide to drop packets.

24.5.3 Table Actions

Table Action Handler

An action handler to be executed on each input packet can be assigned to each table. Defining the action handler for a specific table is optional (i.e. the action handler can be disabled).

The action handler is executed after the table lookup operation is performed and the table entry associated with each input packet is identified. The action handler can only handle the user-defined actions, while the reserved actions (e.g. the next hop actions) are handled by the Packet Framework. The action handler can decide to drop the input packet.

Reserved Actions

The reserved actions are handled directly by the Packet Framework without the user being able to change their meaning through the table action handler configuration. A special category of the reserved actions is represented by the next hop actions, which regulate the packet flow between input ports, tables and output ports through the pipeline. Table 24.16 lists the next hop actions.

Table 24.16: Next Hop Actions (Reserved)

#	Next hop action	Description
1	Drop	Drop the current packet.
2	Send to output port	Send the current packet to specified output port. The output port ID is metadata stored in the same table entry.
3	Send to table	Send the current packet to specified table. The table ID is metadata stored in the same table entry.

User Actions

For each table, the meaning of user actions is defined through the configuration of the table action handler. Different tables can be configured with different action handlers, therefore the meaning of the user actions and their associated meta-data is private to each table. Within the same table, all the table entries (including the table default entry) share the same definition for the user actions and their associated meta-data, with each table entry having its own set

of enabled user actions and its own copy of the action meta-data. Table 24.17 contains a non-exhaustive list of user action examples.

Table 24.17: User Action Examples

#	User action	Description
1	Metering	Per flow traffic metering using the srTCM and trTCM algorithms.
2	Statistics	Update the statistics counters maintained per flow.
3	App ID	Per flow state machine fed by variable length sequence of packets at the flow initialization with the purpose of identifying the traffic type and application.
4	Push/pop labels	Push/pop VLAN/MPLS labels to/from the current packet.
5	Network Address Translation (NAT)	Translate between the internal (LAN) and external (WAN) IP destination/source address and/or L4 protocol destination/source port.
6	TTL update	Decrement IP TTL and, in case of IPv4 packets, update the IP checksum.

24.6 Multicore Scaling

A complex application is typically split across multiple cores, with cores communicating through SW queues. There is usually a performance limit on the number of table lookups and actions that can be fitted on the same CPU core due to HW constraints like: available CPU cycles, cache memory size, cache transfer BW, memory transfer BW, etc.

As the application is split across multiple CPU cores, the Packet Framework facilitates the creation of several pipelines, the assignment of each such pipeline to a different CPU core and the interconnection of all CPU core-level pipelines into a single application-level complex pipeline. For example, if CPU core A is assigned to run pipeline P1 and CPU core B pipeline P2, then the interconnection of P1 with P2 could be achieved by having the same set of SW queues act like output ports for P1 and input ports for P2.

This approach enables the application development using the pipeline, run-to-completion (clustered) or hybrid (mixed) models.

It is allowed for the same core to run several pipelines, but it is not allowed for several cores to run the same pipeline.

24.6.1 Shared Data Structures

The threads performing table lookup are actually table writers rather than just readers. Even if the specific table lookup algorithm is thread-safe for multiple readers (e. g. read-only access of the search algorithm data structures is enough to conduct the lookup operation), once the table entry for the current packet is identified, the thread is typically expected to update the action meta-data stored in the table entry (e.g. increment the counter tracking the number of packets that hit this table entry), and thus modify the table entry. During the time this thread is accessing this table entry (either writing or reading; duration is application specific), for data consistency reasons, no other threads (threads performing table lookup or entry add/delete operations) are allowed to modify this table entry.

Mechanisms to share the same table between multiple threads:

1. **Multiple writer threads.** Threads need to use synchronization primitives like semaphores (distinct semaphore per table entry) or atomic instructions. The cost of semaphores is usually high, even when the semaphore is free. The cost of atomic instructions is normally higher than the cost of regular instructions.
2. **Multiple writer threads, with single thread performing table lookup operations and multiple threads performing table entry add/delete operations.** The threads performing table entry add/delete operations send table update requests to the reader (typically through message passing queues), which does the actual table updates and then sends the response back to the request initiator.
3. **Single writer thread performing table entry add/delete operations and multiple reader threads that perform table lookup operations with read-only access to the table entries.** The reader threads use the main table copy while the writer is updating the mirror copy. Once the writer update is done, the writer can signal to the readers and busy wait until all readers swaps between the mirror copy (which now becomes the main copy) and the mirror copy (which now becomes the main copy).

24.7 Interfacing with Accelerators

The presence of accelerators is usually detected during the initialization phase by inspecting the HW devices that are part of the system (e.g. by PCI bus enumeration). Typical devices with acceleration capabilities are:

- Inline accelerators: NICs, switches, FPGAs, etc;
- Look-aside accelerators: chipsets, FPGAs, etc.

Usually, to support a specific functional block, specific implementation of Packet Framework tables and/or ports and/or actions has to be provided for each accelerator, with all the implementations sharing the same API: pure SW implementation (no acceleration), implementation using accelerator A, implementation using accelerator B, etc. The selection between these implementations could be done at build time or at run-time (recommended), based on which accelerators are present in the system, with no application changes required.

CHAPTER

TWENTYFIVE

VHOST LIBRARY

The vhost library implements a user space virtio net server allowing the user to manipulate the virtio ring directly. In another words, it allows the user to fetch/put packets from/to the VM virtio net device. To achieve this, a vhost library should be able to:

- Access the guest memory:

  For QEMU, this is done by using the `-object memory-backend-file,share=on,...` option. Which means QEMU will create a file to serve as the guest RAM. The `share=on` option allows another process to map that file, which means it can access the guest RAM.

- Know all the necessary information about the vring:

  Information such as where the available ring is stored. Vhost defines some messages (passed through a Unix domain socket file) to tell the backend all the information it needs to know how to manipulate the vring.

25.1 Vhost API Overview

The following is an overview of the Vhost API functions:

- `rte_vhost_driver_register(path, flags)`

  This function registers a vhost driver into the system. `path` specifies the Unix domain socket file path.

  Currently supported flags are:

  - `RTE_VHOST_USER_CLIENT`

    DPDK vhost-user will act as the client when this flag is given. See below for an explanation.

  - `RTE_VHOST_USER_NO_RECONNECT`

    When DPDK vhost-user acts as the client it will keep trying to reconnect to the server (QEMU) until it succeeds. This is useful in two cases:

    * When QEMU is not started yet.
    * When QEMU restarts (for example due to a guest OS reboot).

    This reconnect option is enabled by default. However, it can be turned off by setting this flag.

  - `RTE_VHOST_USER_DEQUEUE_ZERO_COPY`

    Dequeue zero copy will be enabled when this flag is set. It is disabled by default.

    There are some truths (including limitations) you might want to know while setting this flag:

    * zero copy is not good for small packets (typically for packet size below 512).
    * zero copy is really good for VM2VM case. For iperf between two VMs, the boost could be above 70% (when TSO is enableld).
    * for VM2NIC case, the `nb_tx_desc` has to be small enough: <= 64 if virtio indirect feature is not enabled and <= 128 if it is enabled.

      The is because when dequeue zero copy is enabled, guest Tx used vring will be updated only when corresponding mbuf is freed. Thus, the nb_tx_desc has to be small enough so that the PMD driver will run out of available Tx descriptors and free mbufs timely. Otherwise, guest Tx vring would be starved.
    * Guest memory should be backended with huge pages to achieve better performance. Using 1G page size is the best.

      When dequeue zero copy is enabled, the guest phys address and host phys address mapping has to be established. Using non-huge pages means far more page segments. To make it simple, DPDK vhost does a linear search of those segments, thus the fewer the segments, the quicker we will get the mapping. NOTE: we may speed it by using tree searching in future.

- `rte_vhost_driver_session_start()`

  This function starts the vhost session loop to handle vhost messages. It starts an infinite loop, therefore it should be called in a dedicated thread.

- `rte_vhost_driver_callback_register(virtio_net_device_ops)`

  This function registers a set of callbacks, to let DPDK applications take the appropriate action when some events happen. The following events are currently supported:

  - `new_device(int vid)`

    This callback is invoked when a virtio net device becomes ready. `vid` is the virtio net device ID.

  - `destroy_device(int vid)`

    This callback is invoked when a virtio net device shuts down (or when the vhost connection is broken).

  - `vring_state_changed(int vid, uint16_t queue_id, int enable)`

    This callback is invoked when a specific queue's state is changed, for example to enabled or disabled.

- `rte_vhost_enqueue_burst(vid, queue_id, pkts, count)`

  Transmits (enqueues) `count` packets from host to guest.

- `rte_vhost_dequeue_burst(vid, queue_id, mbuf_pool, pkts, count)`

  Receives (dequeues) `count` packets from guest, and stored them at `pkts`.

- `rte_vhost_feature_disable/rte_vhost_feature_enable(feature_mask)`

  This function disables/enables some features. For example, it can be used to disable mergeable buffers and TSO features, which both are enabled by default.

25.2 Vhost-user Implementations

Vhost-user uses Unix domain sockets for passing messages. This means the DPDK vhost-user implementation has two options:

- DPDK vhost-user acts as the server.

  DPDK will create a Unix domain socket server file and listen for connections from the frontend.

  Note, this is the default mode, and the only mode before DPDK v16.07.

- DPDK vhost-user acts as the client.

  Unlike the server mode, this mode doesn't create the socket file; it just tries to connect to the server (which responses to create the file instead).

  When the DPDK vhost-user application restarts, DPDK vhost-user will try to connect to the server again. This is how the "reconnect" feature works.

  Note:

  – The "reconnect" feature requires **QEMU v2.7** (or above).

  – The vhost supported features must be exactly the same before and after the restart. For example, if TSO is disabled and then enabled, nothing will work and issues undefined might happen.

No matter which mode is used, once a connection is established, DPDK vhost-user will start receiving and processing vhost messages from QEMU.

For messages with a file descriptor, the file descriptor can be used directly in the vhost process as it is already installed by the Unix domain socket.

The supported vhost messages are:

- `VHOST_SET_MEM_TABLE`
- `VHOST_SET_VRING_KICK`
- `VHOST_SET_VRING_CALL`
- `VHOST_SET_LOG_FD`
- `VHOST_SET_VRING_ERR`

For `VHOST_SET_MEM_TABLE` message, QEMU will send information for each memory region and its file descriptor in the ancillary data of the message. The file descriptor is used to map that region.

`VHOST_SET_VRING_KICK` is used as the signal to put the vhost device into the data plane, and `VHOST_GET_VRING_BASE` is used as the signal to remove the vhost device from the data plane.

When the socket connection is closed, vhost will destroy the device.

25.3 Vhost supported vSwitch reference

For more vhost details and how to support vhost in vSwitch, please refer to the vhost example in the DPDK Sample Applications Guide.

CHAPTER

TWENTYSIX

PORT HOTPLUG FRAMEWORK

The Port Hotplug Framework provides DPDK applications with the ability to attach and detach ports at runtime. Because the framework depends on PMD implementation, the ports that PMDs cannot handle are out of scope of this framework. Furthermore, after detaching a port from a DPDK application, the framework doesn't provide a way for removing the devices from the system. For the ports backed by a physical NIC, the kernel will need to support PCI Hotplug feature.

26.1 Overview

The basic requirements of the Port Hotplug Framework are:

- DPDK applications that use the Port Hotplug Framework must manage their own ports.

  The Port Hotplug Framework is implemented to allow DPDK applications to manage ports. For example, when DPDK applications call the port attach function, the attached port number is returned. DPDK applications can also detach the port by port number.

- Kernel support is needed for attaching or detaching physical device ports.

  To attach new physical device ports, the device will be recognized by userspace driver I/O framework in kernel at first. Then DPDK applications can call the Port Hotplug functions to attach the ports. For detaching, steps are vice versa.

- Before detaching, they must be stopped and closed.

  DPDK applications must call "rte_eth_dev_stop()" and "rte_eth_dev_close()" APIs before detaching ports. These functions will start finalization sequence of the PMDs.

- The framework doesn't affect legacy DPDK applications behavior.

  If the Port Hotplug functions aren't called, all legacy DPDK apps can still work without modifications.

26.2 Port Hotplug API overview

- Attaching a port

  "rte_eth_dev_attach()" API attaches a port to DPDK application, and returns the attached port number. Before calling the API, the device should be recognized by an userspace driver I/O framework. The API receives a pci address like "0000:01:00.0" or a virtual

device name like "net_pcap0,iface=eth0". In the case of virtual device name, the format is the same as the general "–vdev" option of DPDK.

- Detaching a port

  "rte_eth_dev_detach()" API detaches a port from DPDK application, and returns a pci address of the detached device or a virtual device name of the device.

26.3 Reference

"testpmd" supports the Port Hotplug Framework.

26.4 Limitations

- The Port Hotplug APIs are not thread safe.
- The framework can only be enabled with Linux. BSD is not supported.
- To detach a port, the port should be backed by a device that igb_uio manages. VFIO is not supported.
- Not all PMDs support detaching feature. To know whether a PMD can support detaching, search for the "RTE_PCI_DRV_DETACHABLE" flag in PMD implementation. If the flag is defined in the PMD, detaching is supported.

Part 2: Development Environment

CHAPTER

TWENTYSEVEN

SOURCE ORGANIZATION

This section describes the organization of sources in the DPDK framework.

27.1 Makefiles and Config

Note: In the following descriptions, `RTE_SDK` is the environment variable that points to the base directory into which the tarball was extracted. See *Useful Variables Provided by the Build System* for descriptions of other variables.

Makefiles that are provided by the DPDK libraries and applications are located in `$(RTE_SDK)/mk`.

Config templates are located in `$(RTE_SDK)/config`. The templates describe the options that are enabled for each target. The config file also contains items that can be enabled and disabled for many of the DPDK libraries, including debug options. The user should look at the config file and become familiar with these options. The config file is also used to create a header file, which will be located in the new build directory.

27.2 Libraries

Libraries are located in subdirectories of `$(RTE_SDK)/lib`. By convention a library refers to any code that provides an API to an application. Typically, it generates an archive file (`.a`), but a kernel module would also go in the same directory.

The lib directory contains:

```
lib
+-- librte_cmdline      # Command line interface helper
+-- librte_distributor  # Packet distributor
+-- librte_eal          # Environment abstraction layer
+-- librte_ether        # Generic interface to poll mode driver
+-- librte_hash         # Hash library
+-- librte_ip_frag      # IP fragmentation library
+-- librte_kni          # Kernel NIC interface
+-- librte_kvargs       # Argument parsing library
+-- librte_lpm          # Longest prefix match library
+-- librte_mbuf         # Packet and control mbuf manipulation
+-- librte_mempool      # Memory pool manager (fixed sized objects)
+-- librte_meter        # QoS metering library
+-- librte_net          # Various IP-related headers
+-- librte_power        # Power management library
+-- librte_ring         # Software rings (act as lockless FIFOs)
```

```
+-- librte_sched        # QoS scheduler and dropper library
+-- librte_timer        # Timer library
```

27.3 Drivers

Drivers are special libraries which provide poll-mode driver implementations for devices: either hardware devices or pseudo/virtual devices. They are contained in the *drivers* subdirectory, classified by type, and each compiles to a library with the format `librte_pmd_X.a` where `X` is the driver name.

The drivers directory has a *net* subdirectory which contains:

```
drivers/net
+-- af_packet          # Poll mode driver based on Linux af_packet
+-- bonding            # Bonding poll mode driver
+-- cxgbe              # Chelsio Terminator 10GbE/40GbE poll mode driver
+-- e1000              # 1GbE poll mode drivers (igb and em)
+-- enic               # Cisco VIC Ethernet NIC Poll-mode Driver
+-- fm10k              # Host interface PMD driver for FM10000 Series
+-- i40e               # 40GbE poll mode driver
+-- ixgbe              # 10GbE poll mode driver
+-- mlx4               # Mellanox ConnectX-3 poll mode driver
+-- null               # NULL poll mode driver for testing
+-- pcap               # PCAP poll mode driver
+-- ring               # Ring poll mode driver
+-- szedata2           # SZEDATA2 poll mode driver
+-- virtio             # Virtio poll mode driver
+-- vmxnet3            # VMXNET3 poll mode driver
+-- xenvirt            # Xen virtio poll mode driver
```

Note: Several of the `driver/net` directories contain a `base` sub-directory. The `base` directory generally contains code the shouldn't be modified directly by the user. Any enhancements should be done via the `X_osdep.c` and/or `X_osdep.h` files in that directory. Refer to the local README in the base directories for driver specific instructions.

27.4 Applications

Applications are source files that contain a `main()` function. They are located in the `$(RTE_SDK)/app` and `$(RTE_SDK)/examples` directories.

The app directory contains sample applications that are used to test DPDK (such as autotests) or the Poll Mode Drivers (test-pmd):

```
app
+-- chkincs            # Test program to check include dependencies
+-- cmdline_test       # Test the commandline library
+-- test               # Autotests to validate DPDK features
+-- test-acl           # Test the ACL library
+-- test-pipeline      # Test the IP Pipeline framework
+-- test-pmd           # Test and benchmark poll mode drivers
```

The examples directory contains sample applications that show how libraries can be used:

```
examples
+-- cmdline            # Example of using the cmdline library
+-- dpdk_qat           # Sample integration with Intel QuickAssist
+-- exception_path     # Sending packets to and from Linux TAP device
```

```
+-- helloworld          # Basic Hello World example
+-- ip_reassembly       # Example showing IP reassembly
+-- ip_fragmentation    # Example showing IPv4 fragmentation
+-- ipv4_multicast      # Example showing IPv4 multicast
+-- kni                 # Kernel NIC Interface (KNI) example
+-- l2fwd               # L2 forwarding with and without SR-IOV
+-- l3fwd               # L3 forwarding example
+-- l3fwd-power         # L3 forwarding example with power management
+-- l3fwd-vf            # L3 forwarding example with SR-IOV
+-- link_status_interrupt # Link status change interrupt example
+-- load_balancer       # Load balancing across multiple cores/sockets
+-- multi_process       # Example apps using multiple DPDK processes
+-- qos_meter           # QoS metering example
+-- qos_sched           # QoS scheduler and dropper example
+-- timer               # Example of using librte_timer library
+-- vmdq_dcb            # Example of VMDQ and DCB receiving
+-- vmdq                # Example of VMDQ receiving
+-- vhost               # Example of userspace vhost and switch
```

Note: The actual examples directory may contain additional sample applications to those shown above. Check the latest DPDK source files for details.

CHAPTER

TWENTYEIGHT

DEVELOPMENT KIT BUILD SYSTEM

The DPDK requires a build system for compilation activities and so on. This section describes the constraints and the mechanisms used in the DPDK framework.

There are two use-cases for the framework:

- Compilation of the DPDK libraries and sample applications; the framework generates specific binary libraries, include files and sample applications
- Compilation of an external application or library, using an installed binary DPDK

28.1 Building the Development Kit Binary

The following provides details on how to build the DPDK binary.

28.1.1 Build Directory Concept

After installation, a build directory structure is created. Each build directory contains include files, libraries, and applications.

A build directory is specific to a configuration that includes architecture + execution environment + toolchain. It is possible to have several build directories sharing the same sources with different configurations.

For instance, to create a new build directory called my_sdk_build_dir using the default configuration template config/defconfig_x86_64-linuxapp, we use:

```
cd ${RTE_SDK}
make config T=x86_64-native-linuxapp-gcc O=my_sdk_build_dir
```

This creates a new my_sdk_build_dir directory. After that, we can compile by doing:

```
cd my_sdk_build_dir
make
```

which is equivalent to:

```
make O=my_sdk_build_dir
```

The content of the my_sdk_build_dir is then:

```
-- .config                          # used configuration

-- Makefile                         # wrapper that calls head Makefile
                                    # with $PWD as build directory
```

```
    -- build                                #All temporary files used during build
    +--app                                  # process, including . o, .d, and .cmd files.
        |  +-- test                        # For libraries, we have the .a file.
        |  +-- test.o                      # For applications, we have the elf file.
        |  `-- ...
        +-- lib
            +-- librte_eal
            |   `-- ...
            +-- librte_mempool
            |  +--  mempool-file1.o
            |  +--  .mempool-file1.o.cmd
            |  +--  .mempool-file1.o.d
            |  +--   mempool-file2.o
            |  +--  .mempool-file2.o.cmd
            |  +--  .mempool-file2.o.d
            |  `--  mempool.a
            `-- ...

-- include                # All include files installed by libraries
    +-- librte_mempool.h  # and applications are located in this
    +-- rte_eal.h         # directory. The installed files can depend
    +-- rte_spinlock.h    # on configuration if needed (environment,
    +-- rte_atomic.h      # architecture, ..)
    `-- \*.h ...

-- lib                    # all compiled libraries are copied in this
    +-- librte_eal.a      # directory
    +-- librte_mempool.a
    `-- \*.a ...

-- app                    # All compiled applications are installed
+ --test                  # here. It includes the binary in elf format
```

Refer to *Development Kit Root Makefile Help* for details about make commands that can be used from the root of DPDK.

28.2 Building External Applications

Since DPDK is in essence a development kit, the first objective of end users will be to create an application using this SDK. To compile an application, the user must set the RTE_SDK and RTE_TARGET environment variables.

```
export RTE_SDK=/opt/DPDK
export RTE_TARGET=x86_64-native-linuxapp-gcc
cd /path/to/my_app
```

For a new application, the user must create their own Makefile that includes some .mk files, such as ${RTE_SDK}/mk/rte.vars.mk, and ${RTE_SDK}/mk/ rte.app.mk. This is described in *Building Your Own Application*.

Depending on the chosen target (architecture, machine, executive environment, toolchain) defined in the Makefile or as an environment variable, the applications and libraries will compile using the appropriate .h files and will link with the appropriate .a files. These files are located in ${RTE_SDK}/arch-machine-execenv-toolchain, which is referenced internally by ${RTE_BIN_SDK}.

To compile their application, the user just has to call make. The compilation result will be located in /path/to/my_app/build directory.

Sample applications are provided in the examples directory.

28.3 Makefile Description

28.3.1 General Rules For DPDK Makefiles

In the DPDK, Makefiles always follow the same scheme:

1. Include $(RTE_SDK)/mk/rte.vars.mk at the beginning.
2. Define specific variables for RTE build system.
3. Include a specific $(RTE_SDK)/mk/rte.XYZ.mk, where XYZ can be app, lib, extapp, extlib, obj, gnuconfigure, and so on, depending on what kind of object you want to build. *See Makefile Types* below.
4. Include user-defined rules and variables.

   The following is a very simple example of an external application Makefile:

```
include $(RTE_SDK)/mk/rte.vars.mk

# binary name
APP = helloworld

# all source are stored in SRCS-y
SRCS-y := main.c

CFLAGS += -O3
CFLAGS += $(WERROR_FLAGS)

include $(RTE_SDK)/mk/rte.extapp.mk
```

28.3.2 Makefile Types

Depending on the .mk file which is included at the end of the user Makefile, the Makefile will have a different role. Note that it is not possible to build a library and an application in the same Makefile. For that, the user must create two separate Makefiles, possibly in two different directories.

In any case, the rte.vars.mk file must be included in the user Makefile as soon as possible.

Application

These Makefiles generate a binary application.

- rte.app.mk: Application in the development kit framework
- rte.extapp.mk: External application
- rte.hostapp.mk: prerequisite tool to build dpdk

Library

Generate a .a library.

- rte.lib.mk: Library in the development kit framework
- rte.extlib.mk: external library
- rte.hostlib.mk: host library in the development kit framework

Install

- rte.install.mk: Does not build anything, it is only used to create links or copy files to the installation directory. This is useful for including files in the development kit framework.

Kernel Module

- rte.module.mk: Build a kernel module in the development kit framework.

Objects

- rte.obj.mk: Object aggregation (merge several .o in one) in the development kit framework.
- rte.extobj.mk: Object aggregation (merge several .o in one) outside the development kit framework.

Misc

- rte.doc.mk: Documentation in the development kit framework
- rte.gnuconfigure.mk: Build an application that is configure-based.
- rte.subdir.mk: Build several directories in the development kit framework.

28.3.3 Internally Generated Build Tools

`app/dpdk-pmdinfogen`

`dpdk-pmdinfogen` scans an object (.o) file for various well known symbol names. These well known symbol names are defined by various macros and used to export important information about hardware support and usage for pmd files. For instance the macro:

```
RTE_PMD_REGISTER_PCI(name, drv)
```

Creates the following symbol:

```
static char this_pmd_name0[] __attribute__((used)) = "<name>";
```

Which `dpdk-pmdinfogen` scans for. Using this information other relevant bits of data can be exported from the object file and used to produce a hardware support description, that `dpdk-pmdinfogen` then encodes into a json formatted string in the following format:

```
static char <name_pmd_string>="PMD_INFO_STRING=\"{'name' : '<name>', ...}\"";
```

These strings can then be searched for by external tools to determine the hardware support of a given library or application.

28.3.4 Useful Variables Provided by the Build System

- RTE_SDK: The absolute path to the DPDK sources. When compiling the development kit, this variable is automatically set by the framework. It has to be defined by the user as an environment variable if compiling an external application.
- RTE_SRCDIR: The path to the root of the sources. When compiling the development kit, RTE_SRCDIR = RTE_SDK. When compiling an external application, the variable points to the root of external application sources.
- RTE_OUTPUT: The path to which output files are written. Typically, it is $(RTE_SRCDIR)/build, but it can be overridden by the O= option in the make command line.
- RTE_TARGET: A string identifying the target for which we are building. The format is arch-machine-execenv-toolchain. When compiling the SDK, the target is deduced by the build system from the configuration (.config). When building an external application, it must be specified by the user in the Makefile or as an environment variable.
- RTE_SDK_BIN: References $(RTE_SDK)/$(RTE_TARGET).
- RTE_ARCH: Defines the architecture (i686, x86_64). It is the same value as CONFIG_RTE_ARCH but without the double-quotes around the string.
- RTE_MACHINE: Defines the machine. It is the same value as CONFIG_RTE_MACHINE but without the double-quotes around the string.
- RTE_TOOLCHAIN: Defines the toolchain (gcc , icc). It is the same value as CONFIG_RTE_TOOLCHAIN but without the double-quotes around the string.
- RTE_EXEC_ENV: Defines the executive environment (linuxapp). It is the same value as CONFIG_RTE_EXEC_ENV but without the double-quotes around the string.
- RTE_KERNELDIR: This variable contains the absolute path to the kernel sources that will be used to compile the kernel modules. The kernel headers must be the same as the ones that will be used on the target machine (the machine that will run the application). By default, the variable is set to /lib/modules/$(shell uname -r)/build, which is correct when the target machine is also the build machine.
- RTE_DEVEL_BUILD: Stricter options (stop on warning). It defaults to y in a git tree.

28.3.5 Variables that Can be Set/Overridden in a Makefile Only

- VPATH: The path list that the build system will search for sources. By default, RTE_SRCDIR will be included in VPATH.
- CFLAGS: Flags to use for C compilation. The user should use += to append data in this variable.
- LDFLAGS: Flags to use for linking. The user should use += to append data in this variable.

- ASFLAGS: Flags to use for assembly. The user should use += to append data in this variable.
- CPPFLAGS: Flags to use to give flags to C preprocessor (only useful when assembling .S files). The user should use += to append data in this variable.
- LDLIBS: In an application, the list of libraries to link with (for example, -L /path/to/libfoo -lfoo). The user should use += to append data in this variable.
- SRC-y: A list of source files (.c, .S, or .o if the source is a binary) in case of application, library or object Makefiles. The sources must be available from VPATH.
- INSTALL-y-$(INSTPATH): A list of files to be installed in $(INSTPATH). The files must be available from VPATH and will be copied in $(RTE_OUTPUT)/$(INSTPATH). Can be used in almost any RTE Makefile.
- SYMLINK-y-$(INSTPATH): A list of files to be installed in $(INSTPATH). The files must be available from VPATH and will be linked (symbolically) in $(RTE_OUTPUT)/$(INSTPATH). This variable can be used in almost any DPDK Makefile.
- PREBUILD: A list of prerequisite actions to be taken before building. The user should use += to append data in this variable.
- POSTBUILD: A list of actions to be taken after the main build. The user should use += to append data in this variable.
- PREINSTALL: A list of prerequisite actions to be taken before installing. The user should use += to append data in this variable.
- POSTINSTALL: A list of actions to be taken after installing. The user should use += to append data in this variable.
- PRECLEAN: A list of prerequisite actions to be taken before cleaning. The user should use += to append data in this variable.
- POSTCLEAN: A list of actions to be taken after cleaning. The user should use += to append data in this variable.
- DEPDIR-y: Only used in the development kit framework to specify if the build of the current directory depends on build of another one. This is needed to support parallel builds correctly.

28.3.6 Variables that can be Set/Overridden by the User on the Command Line Only

Some variables can be used to configure the build system behavior. They are documented in *Development Kit Root Makefile Help* and *External Application/Library Makefile Help*

- WERROR_CFLAGS: By default, this is set to a specific value that depends on the compiler. Users are encouraged to use this variable as follows:

```
CFLAGS += $(WERROR_CFLAGS)
```

This avoids the use of different cases depending on the compiler (icc or gcc). Also, this variable can be overridden from the command line, which allows bypassing of the flags for testing purposes.

28.3.7 Variables that Can be Set/Overridden by the User in a Makefile or Command Line

- CFLAGS_my_file.o: Specific flags to add for C compilation of my_file.c.
- LDFLAGS_my_app: Specific flags to add when linking my_app.
- EXTRA_CFLAGS: The content of this variable is appended after CFLAGS when compiling.
- EXTRA_LDFLAGS: The content of this variable is appended after LDFLAGS when linking.
- EXTRA_LDLIBS: The content of this variable is appended after LDLIBS when linking.
- EXTRA_ASFLAGS: The content of this variable is appended after ASFLAGS when assembling.
- EXTRA_CPPFLAGS: The content of this variable is appended after CPPFLAGS when using a C preprocessor on assembly files.

CHAPTER

TWENTYNINE

DEVELOPMENT KIT ROOT MAKEFILE HELP

The DPDK provides a root level Makefile with targets for configuration, building, cleaning, testing, installation and others. These targets are explained in the following sections.

29.1 Configuration Targets

The configuration target requires the name of the target, which is specified using T=mytarget and it is mandatory. The list of available targets are in $(RTE_SDK)/config (remove the defconfig _ prefix).

Configuration targets also support the specification of the name of the output directory, using O=mybuilddir. This is an optional parameter, the default output directory is build.

- Config

  This will create a build directory, and generates a configuration from a template. A Makefile is also created in the new build directory.

  Example:

  ```
  make config O=mybuild T=x86_64-native-linuxapp-gcc
  ```

29.2 Build Targets

Build targets support the optional specification of the name of the output directory, using O=mybuilddir. The default output directory is build.

- all, build or just make

  Build the DPDK in the output directory previously created by a make config.

  Example:

  ```
  make O=mybuild
  ```

- clean

  Clean all objects created using make build.

  Example:

  ```
  make clean O=mybuild
  ```

- %_sub

  Build a subdirectory only, without managing dependencies on other directories.

  Example:

  ```
  make lib/librte_eal_sub O=mybuild
  ```

- %_clean

  Clean a subdirectory only.

  Example:

  ```
  make lib/librte_eal_clean O=mybuild
  ```

29.3 Install Targets

- Install

  The list of available targets are in $(RTE_SDK)/config (remove the defconfig_ prefix).

  The GNU standards variables may be used: http://gnu.org/prep/standards/html_node/Directory-Variables.html and http://gnu.org/prep/standards/html_node/DESTDIR.html

  Example:

  ```
  make install DESTDIR=myinstall prefix=/usr
  ```

29.4 Test Targets

- test

  Launch automatic tests for a build directory specified using O=mybuilddir. It is optional, the default output directory is build.

  Example:

  ```
  make test O=mybuild
  ```

29.5 Documentation Targets

- doc

  Generate the documentation (API and guides).

- doc-api-html

  Generate the Doxygen API documentation in html.

- doc-guides-html

  Generate the guides documentation in html.

- doc-guides-pdf

  Generate the guides documentation in pdf.

29.6 Deps Targets

- depdirs

  This target is implicitly called by make config. Typically, there is no need for a user to call it, except if DEPDIRS-y variables have been updated in Makefiles. It will generate the file $(RTE_OUTPUT)/.depdirs.

  Example:

  ```
  make depdirs O=mybuild
  ```

- depgraph

  This command generates a dot graph of dependencies. It can be displayed to debug circular dependency issues, or just to understand the dependencies.

  Example:

  ```
  make depgraph O=mybuild > /tmp/graph.dot && dotty /tmp/ graph.dot
  ```

29.7 Misc Targets

- help

  Show a quick help.

29.8 Other Useful Command-line Variables

The following variables can be specified on the command line:

- V=

  Enable verbose build (show full compilation command line, and some intermediate commands).

- D=

  Enable dependency debugging. This provides some useful information about why a target is built or not.

- EXTRA_CFLAGS=, EXTRA_LDFLAGS=, EXTRA_LDLIBS=, EXTRA_ASFLAGS=, EXTRA_CPPFLAGS=

  Append specific compilation, link or asm flags.

- CROSS=

  Specify a cross toolchain header that will prefix all gcc/binutils applications. This only works when using gcc.

29.9 Make in a Build Directory

All targets described above are called from the SDK root $(RTE_SDK). It is possible to run the same Makefile targets inside the build directory. For instance, the following command:

```
cd $(RTE_SDK)
make config O=mybuild T=x86_64-native-linuxapp-gcc
make O=mybuild
```

is equivalent to:

```
cd $(RTE_SDK)
make config O=mybuild T=x86_64-native-linuxapp-gcc
cd mybuild

# no need to specify O= now
make
```

29.10 Compiling for Debug

To compile the DPDK and sample applications with debugging information included and the optimization level set to 0, the EXTRA_CFLAGS environment variable should be set before compiling as follows:

```
export EXTRA_CFLAGS='-O0 -g'
```

CHAPTER

THIRTY

EXTENDING THE DPDK

This chapter describes how a developer can extend the DPDK to provide a new library, a new target, or support a new target.

30.1 Example: Adding a New Library libfoo

To add a new library to the DPDK, proceed as follows:

1. Add a new configuration option:

```
for f in config/\*; do \
    echo CONFIG_RTE_LIBFOO=y >> $f; done
```

1. Create a new directory with sources:

```
mkdir ${RTE_SDK}/lib/libfoo
touch ${RTE_SDK}/lib/libfoo/foo.c
touch ${RTE_SDK}/lib/libfoo/foo.h
```

1. Add a foo() function in libfoo.

   Definition is in foo.c:

```
void foo(void)
{
}
```

   Declaration is in foo.h:

```
extern void foo(void);
```

2. Update lib/Makefile:

```
vi ${RTE_SDK}/lib/Makefile
# add:
# DIRS-$(CONFIG_RTE_LIBFOO) += libfoo
```

3. Create a new Makefile for this library, for example, derived from mempool Makefile:

```
cp ${RTE_SDK}/lib/librte_mempool/Makefile ${RTE_SDK}/lib/libfoo/

vi ${RTE_SDK}/lib/libfoo/Makefile
# replace:
# librte_mempool -> libfoo
# rte_mempool -> foo
```

4. Update mk/DPDK.app.mk, and add -lfoo in LDLIBS variable when the option is enabled. This will automatically add this flag when linking a DPDK application.

5. Build the DPDK with the new library (we only show a specific target here):

```
cd ${RTE_SDK}
make config T=x86_64-native-linuxapp-gcc
make
```

6. Check that the library is installed:

```
ls build/lib
ls build/include
```

30.1.1 Example: Using libfoo in the Test Application

The test application is used to validate all functionality of the DPDK. Once you have added a library, a new test case should be added in the test application.

- A new test_foo.c file should be added, that includes foo.h and calls the foo() function from test_foo(). When the test passes, the test_foo() function should return 0.
- Makefile, test.h and commands.c must be updated also, to handle the new test case.
- Test report generation: autotest.py is a script that is used to generate the test report that is available in the ${RTE_SDK}/doc/rst/test_report/autotests directory. This script must be updated also. If libfoo is in a new test family, the links in ${RTE_SDK}/doc/rst/test_report/test_report.rst must be updated.
- Build the DPDK with the updated test application (we only show a specific target here):

```
cd ${RTE_SDK}
make config T=x86_64-native-linuxapp-gcc
make
```

CHAPTER

THIRTYONE

BUILDING YOUR OWN APPLICATION

31.1 Compiling a Sample Application in the Development Kit Directory

When compiling a sample application (for example, hello world), the following variables must be exported: RTE_SDK and RTE_TARGET.

```
~/DPDK$ cd examples/helloworld/
~/DPDK/examples/helloworld$ export RTE_SDK=/home/user/DPDK
~/DPDK/examples/helloworld$ export RTE_TARGET=x86_64-native-linuxapp-gcc
~/DPDK/examples/helloworld$ make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map
```

The binary is generated in the build directory by default:

```
~/DPDK/examples/helloworld$ ls build/app
helloworld helloworld.map
```

31.2 Build Your Own Application Outside the Development Kit

The sample application (Hello World) can be duplicated in a new directory as a starting point for your development:

```
~$ cp -r DPDK/examples/helloworld my_rte_app
~$ cd my_rte_app/
~/my_rte_app$ export RTE_SDK=/home/user/DPDK
~/my_rte_app$ export RTE_TARGET=x86_64-native-linuxapp-gcc
~/my_rte_app$ make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map
```

31.3 Customizing Makefiles

31.3.1 Application Makefile

The default makefile provided with the Hello World sample application is a good starting point. It includes:

- $(RTE_SDK)/mk/rte.vars.mk at the beginning
- $(RTE_SDK)/mk/rte.extapp.mk at the end

The user must define several variables:

- APP: Contains the name of the application.
- SRCS-y: List of source files (*.c, *.S).

31.3.2 Library Makefile

It is also possible to build a library in the same way:

- Include $(RTE_SDK)/mk/rte.vars.mk at the beginning.
- Include $(RTE_SDK)/mk/rte.extlib.mk at the end.

The only difference is that APP should be replaced by LIB, which contains the name of the library. For example, libfoo.a.

31.3.3 Customize Makefile Actions

Some variables can be defined to customize Makefile actions. The most common are listed below. Refer to *Makefile Description* section in *Development Kit Build System*

chapter for details.

- VPATH: The path list where the build system will search for sources. By default, RTE_SRCDIR will be included in VPATH.
- CFLAGS_my_file.o: The specific flags to add for C compilation of my_file.c.
- CFLAGS: The flags to use for C compilation.
- LDFLAGS: The flags to use for linking.
- CPPFLAGS: The flags to use to provide flags to the C preprocessor (only useful when assembling .S files)
- LDLIBS: A list of libraries to link with (for example, -L /path/to/libfoo - lfoo)

CHAPTER

THIRTYTWO

EXTERNAL APPLICATION/LIBRARY MAKEFILE HELP

External applications or libraries should include specific Makefiles from RTE_SDK, located in mk directory. These Makefiles are:

- ${RTE_SDK}/mk/rte.extapp.mk: Build an application
- ${RTE_SDK}/mk/rte.extlib.mk: Build a static library
- ${RTE_SDK}/mk/rte.extobj.mk: Build objects (.o)

32.1 Prerequisites

The following variables must be defined:

- ${RTE_SDK}: Points to the root directory of the DPDK.
- ${RTE_TARGET}: Reference the target to be used for compilation (for example, x86_64-native-linuxapp-gcc).

32.2 Build Targets

Build targets support the specification of the name of the output directory, using O=mybuilddir. This is optional; the default output directory is build.

- all, “nothing” (meaning just make)

  Build the application or the library in the specified output directory.

  Example:

  ```
  make O=mybuild
  ```

- clean

  Clean all objects created using make build.

  Example:

  ```
  make clean O=mybuild
  ```

32.3 Help Targets

- help

Show this help.

32.4 Other Useful Command-line Variables

The following variables can be specified at the command line:

- S=

  Specify the directory in which the sources are located. By default, it is the current directory.

- M=

  Specify the Makefile to call once the output directory is created. By default, it uses $(S)/Makefile.

- V=

  Enable verbose build (show full compilation command line and some intermediate commands).

- D=

  Enable dependency debugging. This provides some useful information about why a target must be rebuilt or not.

- EXTRA_CFLAGS=, EXTRA_LDFLAGS=, EXTRA_ASFLAGS=, EXTRA_CPPFLAGS=

  Append specific compilation, link or asm flags.

- CROSS=

  Specify a cross-toolchain header that will prefix all gcc/binutils applications. This only works when using gcc.

32.5 Make from Another Directory

It is possible to run the Makefile from another directory, by specifying the output and the source dir. For example:

```
export RTE_SDK=/path/to/DPDK
export RTE_TARGET=x86_64-native-linuxapp-icc
make -f /path/to/my_app/Makefile S=/path/to/my_app O=/path/to/build_dir
```

Part 3: Performance Optimization

CHAPTER

THIRTYTHREE

PERFORMANCE OPTIMIZATION GUIDELINES

33.1 Introduction

The following sections describe optimizations used in the DPDK and optimizations that should be considered for a new applications.

They also highlight the performance-impacting coding techniques that should, and should not be, used when developing an application using the DPDK.

And finally, they give an introduction to application profiling using a Performance Analyzer from Intel to optimize the software.

CHAPTER

THIRTYFOUR

WRITING EFFICIENT CODE

This chapter provides some tips for developing efficient code using the DPDK. For additional and more general information, please refer to the *Intel® 64 and IA-32 Architectures Optimization Reference Manual* which is a valuable reference to writing efficient code.

34.1 Memory

This section describes some key memory considerations when developing applications in the DPDK environment.

34.1.1 Memory Copy: Do not Use libc in the Data Plane

Many libc functions are available in the DPDK, via the Linux* application environment. This can ease the porting of applications and the development of the configuration plane. However, many of these functions are not designed for performance. Functions such as memcpy() or strcpy() should not be used in the data plane. To copy small structures, the preference is for a simpler technique that can be optimized by the compiler. Refer to the *VTune™ Performance Analyzer Essentials* publication from Intel Press for recommendations.

For specific functions that are called often, it is also a good idea to provide a self-made optimized function, which should be declared as static inline.

The DPDK API provides an optimized rte_memcpy() function.

34.1.2 Memory Allocation

Other functions of libc, such as malloc(), provide a flexible way to allocate and free memory. In some cases, using dynamic allocation is necessary, but it is really not advised to use malloc-like functions in the data plane because managing a fragmented heap can be costly and the allocator may not be optimized for parallel allocation.

If you really need dynamic allocation in the data plane, it is better to use a memory pool of fixed-size objects. This API is provided by librte_mempool. This data structure provides several services that increase performance, such as memory alignment of objects, lockless access to objects, NUMA awareness, bulk get/put and per-lcore cache. The rte_malloc () function uses a similar concept to mempools.

34.1.3 Concurrent Access to the Same Memory Area

Read-Write (RW) access operations by several lcores to the same memory area can generate a lot of data cache misses, which are very costly. It is often possible to use per-lcore variables, for example, in the case of statistics. There are at least two solutions for this:

- Use RTE_PER_LCORE variables. Note that in this case, data on lcore X is not available to lcore Y.
- Use a table of structures (one per lcore). In this case, each structure must be cache-aligned.

Read-mostly variables can be shared among lcores without performance losses if there are no RW variables in the same cache line.

34.1.4 NUMA

On a NUMA system, it is preferable to access local memory since remote memory access is slower. In the DPDK, the memzone, ring, rte_malloc and mempool APIs provide a way to create a pool on a specific socket.

Sometimes, it can be a good idea to duplicate data to optimize speed. For read-mostly variables that are often accessed, it should not be a problem to keep them in one socket only, since data will be present in cache.

34.1.5 Distribution Across Memory Channels

Modern memory controllers have several memory channels that can load or store data in parallel. Depending on the memory controller and its configuration, the number of channels and the way the memory is distributed across the channels varies. Each channel has a bandwidth limit, meaning that if all memory access operations are done on the first channel only, there is a potential bottleneck.

By default, the *Mempool Library* spreads the addresses of objects among memory channels.

34.2 Communication Between lcores

To provide a message-based communication between lcores, it is advised to use the DPDK ring API, which provides a lockless ring implementation.

The ring supports bulk and burst access, meaning that it is possible to read several elements from the ring with only one costly atomic operation (see Ring Library). Performance is greatly improved when using bulk access operations.

The code algorithm that dequeues messages may be something similar to the following:

```
#define MAX_BULK 32

while (1) {
    /* Process as many elements as can be dequeued. */
    count = rte_ring_dequeue_burst(ring, obj_table, MAX_BULK);
    if (unlikely(count == 0))
        continue;
```

```
        my_process_bulk(obj_table, count);
}
```

34.3 PMD Driver

The DPDK Poll Mode Driver (PMD) is also able to work in bulk/burst mode, allowing the factorization of some code for each call in the send or receive function.

Avoid partial writes. When PCI devices write to system memory through DMA, it costs less if the write operation is on a full cache line as opposed to part of it. In the PMD code, actions have been taken to avoid partial writes as much as possible.

34.3.1 Lower Packet Latency

Traditionally, there is a trade-off between throughput and latency. An application can be tuned to achieve a high throughput, but the end-to-end latency of an average packet will typically increase as a result. Similarly, the application can be tuned to have, on average, a low end-to-end latency, at the cost of lower throughput.

In order to achieve higher throughput, the DPDK attempts to aggregate the cost of processing each packet individually by processing packets in bursts.

Using the testpmd application as an example, the burst size can be set on the command line to a value of 16 (also the default value). This allows the application to request 16 packets at a time from the PMD. The testpmd application then immediately attempts to transmit all the packets that were received, in this case, all 16 packets.

The packets are not transmitted until the tail pointer is updated on the corresponding TX queue of the network port. This behavior is desirable when tuning for high throughput because the cost of tail pointer updates to both the RX and TX queues can be spread across 16 packets, effectively hiding the relatively slow MMIO cost of writing to the PCIe* device. However, this is not very desirable when tuning for low latency because the first packet that was received must also wait for another 15 packets to be received. It cannot be transmitted until the other 15 packets have also been processed because the NIC will not know to transmit the packets until the TX tail pointer has been updated, which is not done until all 16 packets have been processed for transmission.

To consistently achieve low latency, even under heavy system load, the application developer should avoid processing packets in bunches. The testpmd application can be configured from the command line to use a burst value of 1. This will allow a single packet to be processed at a time, providing lower latency, but with the added cost of lower throughput.

34.4 Locks and Atomic Operations

Atomic operations imply a lock prefix before the instruction, causing the processor's LOCK# signal to be asserted during execution of the following instruction. This has a big impact on performance in a multicore environment.

Performance can be improved by avoiding lock mechanisms in the data plane. It can often be replaced by other solutions like per-lcore variables. Also, some locking techniques are more

efficient than others. For instance, the Read-Copy-Update (RCU) algorithm can frequently replace simple rwlocks.

34.5 Coding Considerations

34.5.1 Inline Functions

Small functions can be declared as static inline in the header file. This avoids the cost of a call instruction (and the associated context saving). However, this technique is not always efficient; it depends on many factors including the compiler.

34.5.2 Branch Prediction

The Intel® C/C++ Compiler (icc)/gcc built-in helper functions likely() and unlikely() allow the developer to indicate if a code branch is likely to be taken or not. For instance:

```
if (likely(x > 1))
    do_stuff();
```

34.6 Setting the Target CPU Type

The DPDK supports CPU microarchitecture-specific optimizations by means of CONFIG_RTE_MACHINE option in the DPDK configuration file. The degree of optimization depends on the compiler's ability to optimize for a specific microarchitecture, therefore it is preferable to use the latest compiler versions whenever possible.

If the compiler version does not support the specific feature set (for example, the Intel® AVX instruction set), the build process gracefully degrades to whatever latest feature set is supported by the compiler.

Since the build and runtime targets may not be the same, the resulting binary also contains a platform check that runs before the main() function and checks if the current machine is suitable for running the binary.

Along with compiler optimizations, a set of preprocessor defines are automatically added to the build process (regardless of the compiler version). These defines correspond to the instruction sets that the target CPU should be able to support. For example, a binary compiled for any SSE4.2-capable processor will have RTE_MACHINE_CPUFLAG_SSE4_2 defined, thus enabling compile-time code path selection for different platforms.

CHAPTER

THIRTYFIVE

PROFILE YOUR APPLICATION

The following sections describe methods of profiling DPDK applications on different architectures.

35.1 Profiling on x86

Intel processors provide performance counters to monitor events. Some tools provided by Intel, such as VTune, can be used to profile and benchmark an application. See the *VTune Performance Analyzer Essentials* publication from Intel Press for more information.

For a DPDK application, this can be done in a Linux* application environment only.

The main situations that should be monitored through event counters are:

- Cache misses
- Branch mis-predicts
- DTLB misses
- Long latency instructions and exceptions

Refer to the Intel Performance Analysis Guide for details about application profiling.

35.2 Profiling on ARM64

35.2.1 Using Linux perf

The ARM64 architecture provide performance counters to monitor events. The Linux `perf` tool can be used to profile and benchmark an application. In addition to the standard events, `perf` can be used to profile arm64 specific PMU (Performance Monitor Unit) events through raw events (`-e -rXX`).

For more derails refer to the ARM64 specific PMU events enumeration.

35.2.2 High-resolution cycle counter

The default `cntvct_el0` based `rte_rdtsc()` provides a portable means to get a wall clock counter in user space. Typically it runs at <= 100MHz.

The alternative method to enable `rte_rdtsc()` for a high resolution wall clock counter is through the armv8 PMU subsystem. The PMU cycle counter runs at CPU frequency. However, access to the PMU cycle counter from user space is not enabled by default in the arm64 linux kernel. It is possible to enable cycle counter for user space access by configuring the PMU from the privileged mode (kernel space).

By default the `rte_rdtsc()` implementation uses a portable `cntvct_el0` scheme. Application can choose the PMU based implementation with `CONFIG_RTE_ARM_EAL_RDTSC_USE_PMU`.

The example below shows the steps to configure the PMU based cycle counter on an armv8 machine.

```
git clone https://github.com/jerinjacobk/armv8_pmu_cycle_counter_el0
cd armv8_pmu_cycle_counter_el0
make
sudo insmod pmu_el0_cycle_counter.ko
cd $DPDK_DIR
make config T=arm64-armv8a-linuxapp-gcc
echo "CONFIG_RTE_ARM_EAL_RDTSC_USE_PMU=y" >> build/.config
make
```

Warning: The PMU based scheme is useful for high accuracy performance profiling with `rte_rdtsc()`. However, this method can not be used in conjunction with Linux userspace profiling tools like `perf` as this scheme alters the PMU registers state.

CHAPTER

THIRTYSIX

GLOSSARY

ACL Access Control List

API Application Programming Interface

ASLR Linux* kernel Address-Space Layout Randomization

BSD Berkeley Software Distribution

Clr Clear

CIDR Classless Inter-Domain Routing

Control Plane The control plane is concerned with the routing of packets and with providing a start or end point.

Core A core may include several lcores or threads if the processor supports hyperthreading.

Core Components A set of libraries provided by the DPDK, including eal, ring, mempool, mbuf, timers, and so on.

CPU Central Processing Unit

CRC Cyclic Redundancy Check

ctrlmbuf An *mbuf* carrying control data.

Data Plane In contrast to the control plane, the data plane in a network architecture are the layers involved when forwarding packets. These layers must be highly optimized to achieve good performance.

DIMM Dual In-line Memory Module

Doxygen A documentation generator used in the DPDK to generate the API reference.

DPDK Data Plane Development Kit

DRAM Dynamic Random Access Memory

EAL The Environment Abstraction Layer (EAL) provides a generic interface that hides the environment specifics from the applications and libraries. The services expected from the EAL are: development kit loading and launching, core affinity/ assignment procedures, system memory allocation/description, PCI bus access, inter-partition communication.

FIFO First In First Out

FPGA Field Programmable Gate Array

GbE Gigabit Ethernet

HW Hardware

HPET High Precision Event Timer; a hardware timer that provides a precise time reference on x86 platforms.

ID Identifier

IOCTL Input/Output Control

I/O Input/Output

IP Internet Protocol

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

lcore A logical execution unit of the processor, sometimes called a *hardware thread*.

KNI Kernel Network Interface

L1 Layer 1

L2 Layer 2

L3 Layer 3

L4 Layer 4

LAN Local Area Network

LPM Longest Prefix Match

master lcore The execution unit that executes the main() function and that launches other lcores.

mbuf An mbuf is a data structure used internally to carry messages (mainly network packets). The name is derived from BSD stacks. To understand the concepts of packet buffers or mbuf, refer to *TCP/IP Illustrated, Volume 2: The Implementation*.

MESI Modified Exclusive Shared Invalid (CPU cache coherency protocol)

MTU Maximum Transfer Unit

NIC Network Interface Card

OOO Out Of Order (execution of instructions within the CPU pipeline)

NUMA Non-uniform Memory Access

PCI Peripheral Connect Interface

PHY An abbreviation for the physical layer of the OSI model.

pktmbuf An *mbuf* carrying a network packet.

PMD Poll Mode Driver

QoS Quality of Service

RCU Read-Copy-Update algorithm, an alternative to simple rwlocks.

Rd Read

RED Random Early Detection

RSS Receive Side Scaling

RTE Run Time Environment. Provides a fast and simple framework for fast packet processing, in a lightweight environment as a Linux* application and using Poll Mode Drivers (PMDs) to increase speed.

Rx Reception

Slave lcore Any *lcore* that is not the *master lcore*.

Socket A physical CPU, that includes several *cores*.

SLA Service Level Agreement

srTCM Single Rate Three Color Marking

SRTD Scheduler Round Trip Delay

SW Software

Target In the DPDK, the target is a combination of architecture, machine, executive environment and toolchain. For example: i686-native-linuxapp-gcc.

TCP Transmission Control Protocol

TC Traffic Class

TLB Translation Lookaside Buffer

TLS Thread Local Storage

trTCM Two Rate Three Color Marking

TSC Time Stamp Counter

Tx Transmission

TUN/TAP TUN and TAP are virtual network kernel devices.

VLAN Virtual Local Area Network

Wr Write

WRED Weighted Random Early Detection

WRR Weighted Round Robin

Figures

Tables