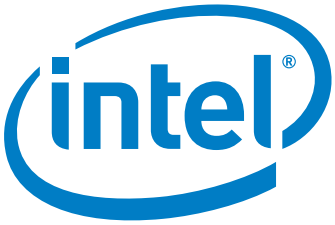
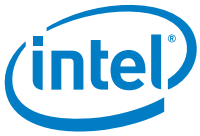


Intel® Data Plane Development Kit (Intel® DPDK)

Programmer's Guide

October 2013

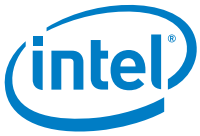
Reference Number: 326003-005

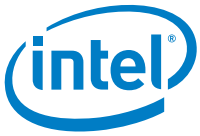


Revision History

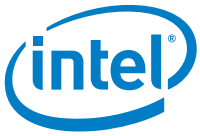
Date	Revision	Description
October 2013	-005	Supports public software release 1.5.1
September 2013	-004	Supports public software release 1.5.0 • Added Chapter 11.0, "Poll Mode Driver for Emulated Virtio NIC" • Added Chapter 12.0, "Libpcap and Ring Based Poll Mode Drivers" • Updated Section 13.1, "Implementation Details" on page 64 • Updated Chapter 16.0, "Multi-process Support" • Added Section 17.8, "KNI Working as a Kernel vHost Backend" on page 78
August 2013	-003	Supports public software release 1.4.1
June 2013	-002	Supports public software release 1.3.1
November 2012	-001	Supports public software release 1.2.3 Minor updates to Section 5.5.3.2 and Section 5.5.3.3 since last limited distribution release.

Contents





Figures



§ §

1.0 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 *Intel® DPDK IPL Sample Application's User Guide* for details.

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

1.1 Documentation Roadmap

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

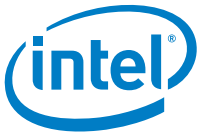
- **Release Notes**: Provides release-specific information, including supported features, limitations, fixed issues, known issues and so on. Also, provides the answers to frequently asked questions in FAQ format.
- **Getting Started Guide**: Describes how to install and configure the Intel® DPDK software; designed to get users up and running quickly with the software.
- **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 Intel® DPDK, the build system (including the commands that can be used in the root Intel® DPDK Makefile to build the development kit and applications) 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 Intel® 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 follow documents provide information that is relevant to the development of applications using the Intel® DPDK:

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



Part 1: Architecture Overview

2.0 Overview

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

The main goal of the Intel® 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 Intel® 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 Intel® DPDK.

The Intel® 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 Intel® 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 Intel® 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 Intel® 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-default-linuxapp-gcc
```

See the *Intel® DPDK IPL 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:

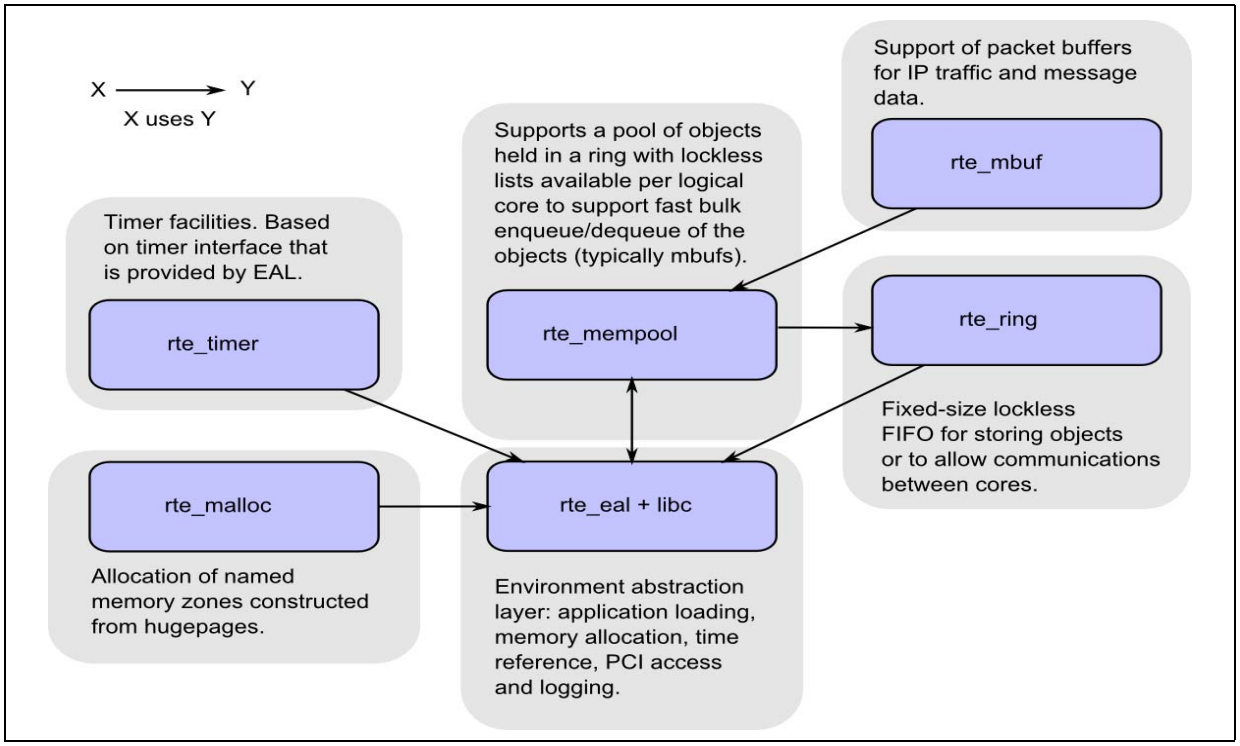
- Intel® DPDK loading and launching
- Support for multi-process and multi-thread execution types
- Core affinity/assignment procedures
- System memory allocation/de-allocation
- Atomic/lock operations
- Time reference
- PCI bus access
- Trace and debug functions
- CPU feature identification
- Interrupt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

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.

Figure 1. Core Components Architecture



2.3.1 Memory Manager (librte_malloc)

The librte_malloc library provides an API to allocate memory from the memzones created from the hugepages instead of the heap. This helps when allocating large numbers of items that may become susceptible to TLB misses when using typical 4k heap pages in the Linux user space environment.

This memory allocator is fully described in Malloc Library.

2.3.2 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.

2.3.3 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 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.

2.3.4 Network Packet Buffer Management (librte_mbuf)

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

This library provide 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.

2.3.5 Timer Manager (librte_timer)

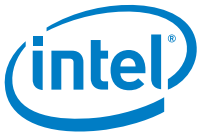
This library provides a timer service to Intel® DPDK execution units, providing the ability to execute a function asynchronously. It can be periodic function calls, or just a one-shot call. It uses the timer interface provided by the Environment Abstraction Layer (EAL) to get a precise time reference 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 Intel® DPDK includes Poll Mode Drivers (PMDs) for 1 GbE and 10 GbE, and para virtualized virtio Ethernet controllers which are designed to work without asynchronous, interrupt-based signalling mechanisms.

See Poll Mode Driver.



2.5 Packet Forwarding Algorithm Support

The Intel® 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.

§ §

3.0 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:

- Intel® DPDK Loading and Launching: The Intel® 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 Intel® 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 a module called `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 Intel® DPDK service layers such as the Mempool Library.

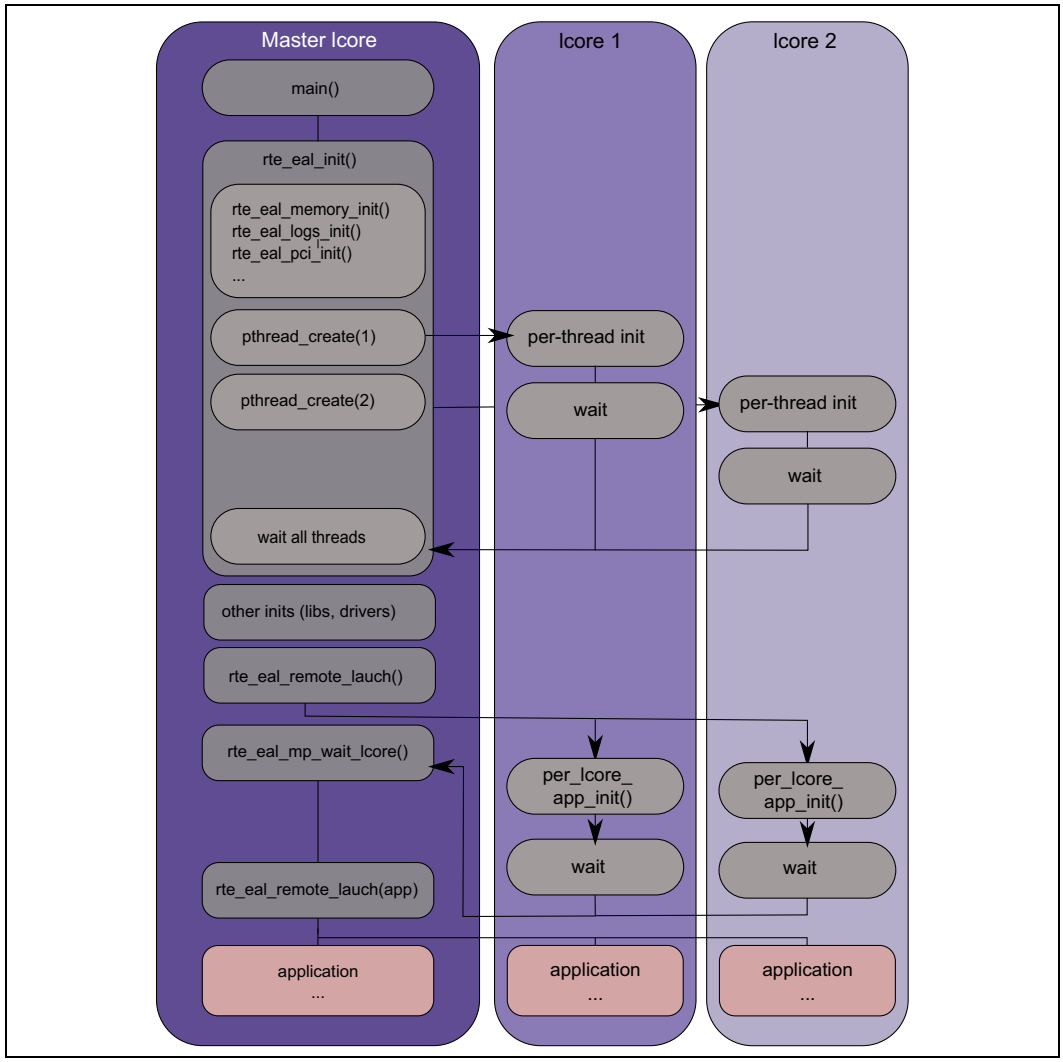
At this point, the Intel® DPDK services layer will be initialized, then through pthread `setaffinity` calls, each execution unit will be assigned to a specific logical core to run as a user-level thread.

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()`).

Figure 2. EAL Initialization in a Linux Application Environment



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 16.0, "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 the rte_malloc library are also backed by pages from the `hugetlbfs` filesystem. However, physical address information is not available for the blocks of memory allocated in this way.

3.1.4 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 `igb_uio` provides a `/dev/uioX` device file that can be `mmap`'d to obtain access to PCI address space from the application. It uses the `uio` kernel feature (userland driver).

3.1.5 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.

3.1.6 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.

3.1.6.1 Trace and Debug Functions

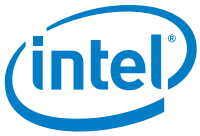
There are some debug functions to dump the stack in glibc. The `rte_panic()` function can voluntarily provoke a SIG_ABORT, which can trigger the generation of a core file, readable by gdb.

3.1.7 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.

3.1.8 User Space Interrupt and Alarm Handling

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: The only interrupts supported by the Intel® DPDK Poll-Mode Drivers are those for link status change, i.e. link up and link down notification.

3.1.9 Blacklisting

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

3.1.10 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.

§ §

4.0 Malloc Library

The `librte_malloc` library provides an API to allocate any-sized memory.

The objective of this library is to provide malloc-like functions to allow allocation from hugepage memory and to facilitate application porting. The *Intel® DPDK IPL 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 *Intel® DPDK IPL API Reference* manual for more information.

4.1 Cookies

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

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).

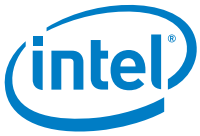
4.3 Use Cases

This library is needed by an application that requires malloc-like functions at initialization time, and does not require the physical address information for the individual memory blocks.

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

If a block of memory with a known physical address is needed, e.g. for use by a hardware device, a memory zone should be used.

§ §



5.0 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:

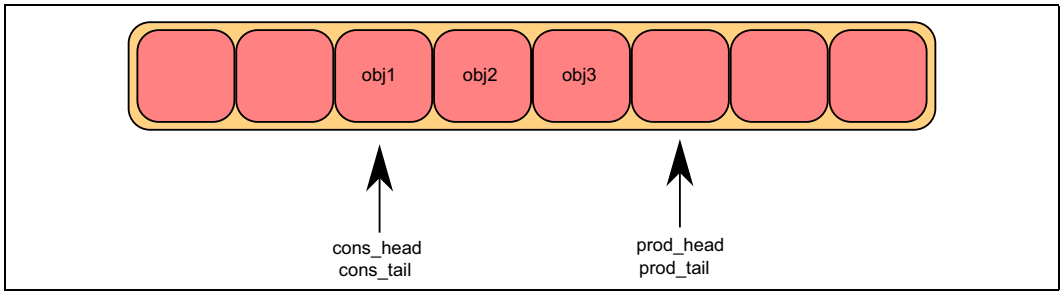
- Faster; only requires a single Compare-And-Swap instruction of `sizeof(void *)` instead of several double-Compare-And-Swap instructions.
- Simpler than a full lockless queue.
- Adapted to bulk enqueue/dequeue operations. As pointers are stored in a table, a dequeue of several objects will not produce 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 Figure 3 with consumer and producer head and tail pointers to objects stored in the data structure.

Figure 3. Ring Structure



5.1 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.c in FreeBSD
- bufring.h in FreeBSD

5.2 Lockless Ring Buffer in Linux*

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

5.3 Additional Features

5.3.1 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).

5.3.2 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.

5.3.3 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.

5.4 Use Cases

Use cases for the Ring library include:

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

5.5 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.

5.5.1 Single Producer Enqueue

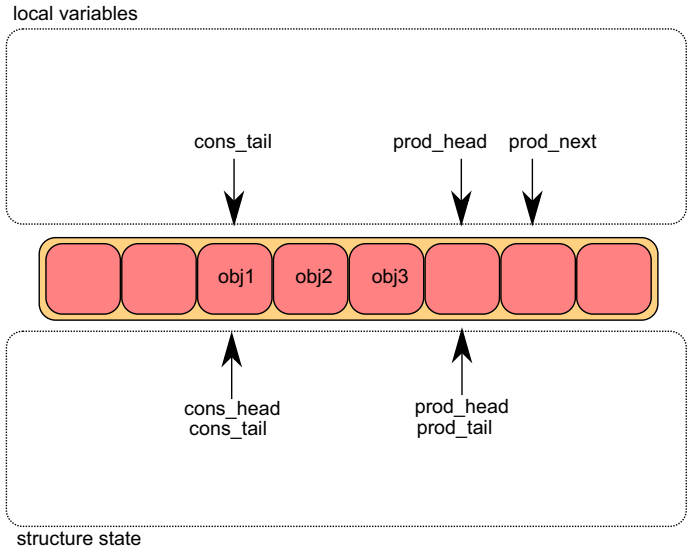
This section explains what occurs when a producer adds an object to the ring. In this example, only the producer head and tail (`prod_head` and `prod_tail`) are modified, and there is only one producer.

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

5.5.1.1 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.

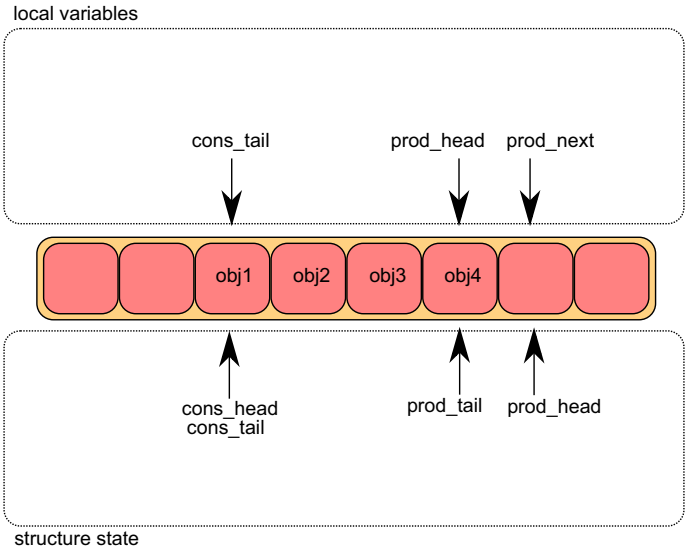
If there is not enough room in the ring (this is detected by checking `cons_tail`), it returns an error.



5.5.1.2 Enqueue Second Step

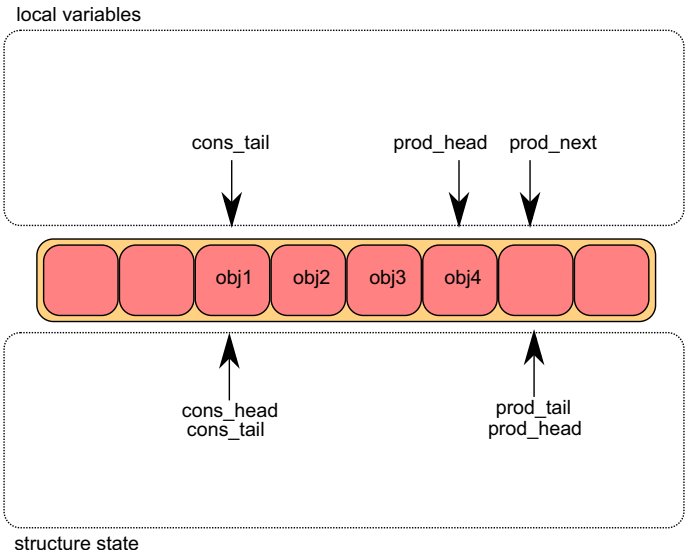
The second step is to modify `ring->prod_head` in ring structure to point to the same location as `prod_next`.

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



5.5.1.3 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.



5.5.2 Single Consumer Dequeue

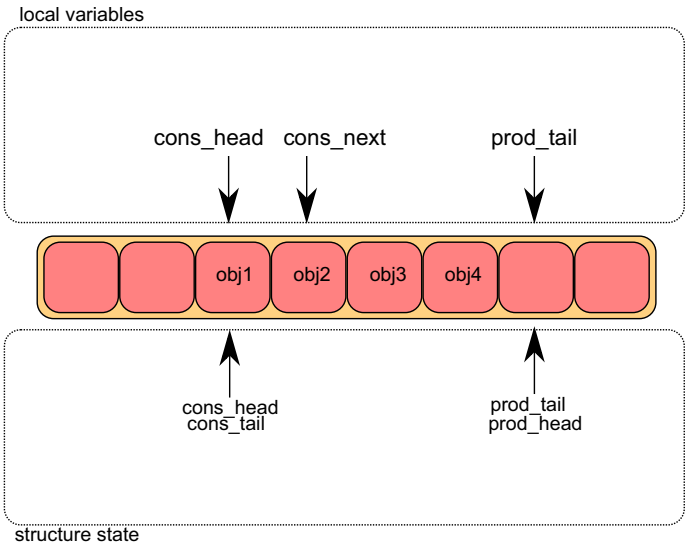
This section explains what occurs when a consumer dequeues an object from the ring. In this example, only the consumer head and tail (`cons_head` and `cons_tail`) are modified and there is only one consumer.

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

5.5.2.1 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.

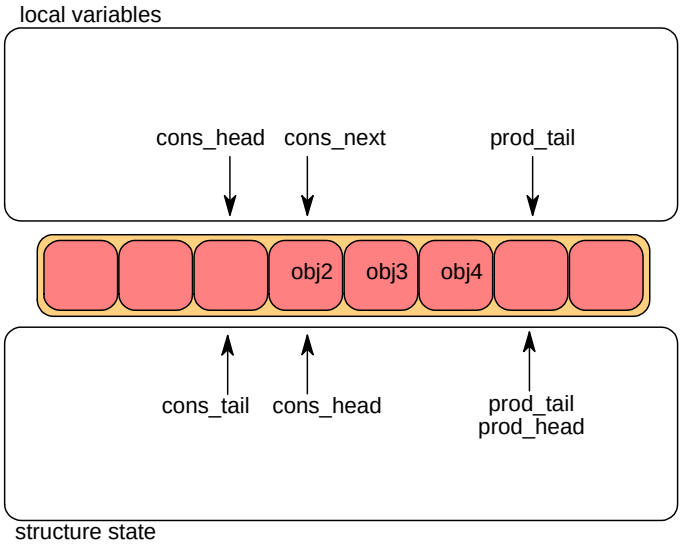
If there are not enough objects in the ring (this is detected by checking `prod_tail`), it returns an error.



5.5.2.2 Dequeue Second Step

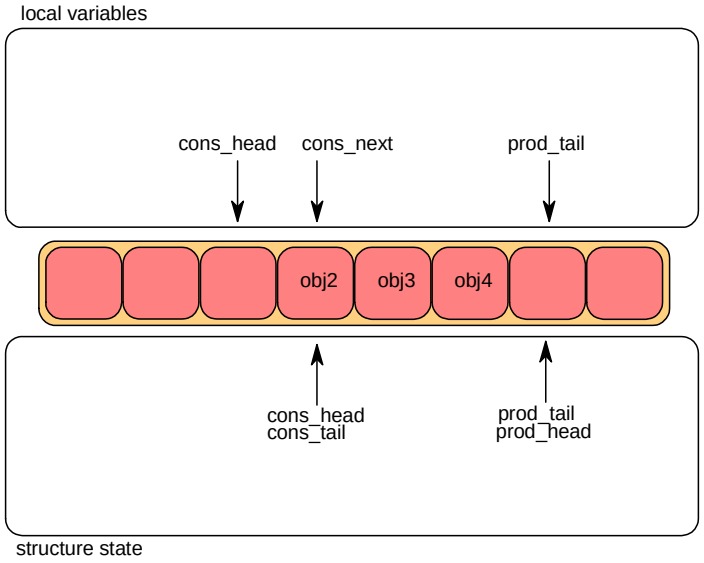
The second step is to modify `ring->cons_head` in the ring structure to point to the same location as `cons_next`.

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



5.5.2.3 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.



5.5.3 Multiple Producers Enqueue

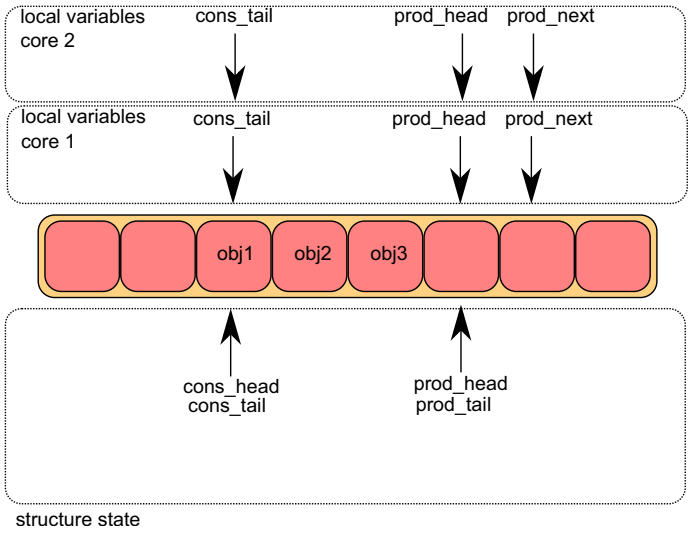
This section explains what occurs when two producers concurrently add an object to the ring. In this example, only the producer head and tail (`prod_head` and `prod_tail`) are modified.

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

5.5.3.1 MC 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 are not enough objects in the ring (this is detected by checking `cons_tail`), it returns an error.

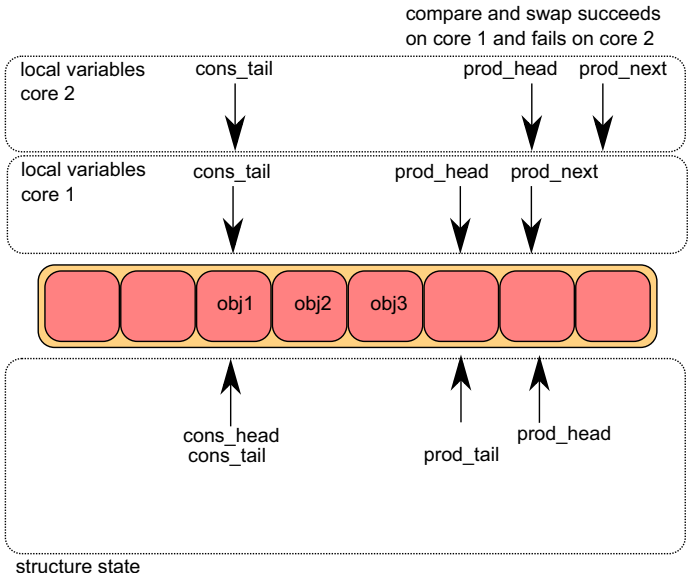


5.5.3.2 MC Enqueue Second Step

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.

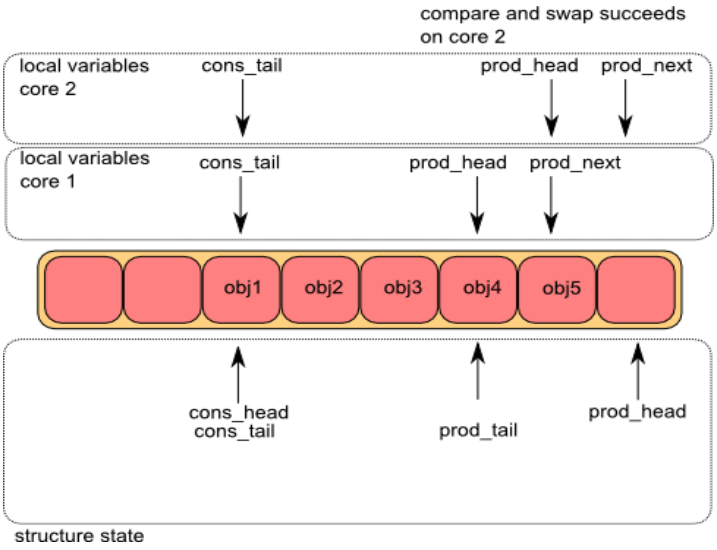
In the figure, the operation succeeded on core 1, and step one restarted on core 2.



5.5.3.3 MC Enqueue Third Step

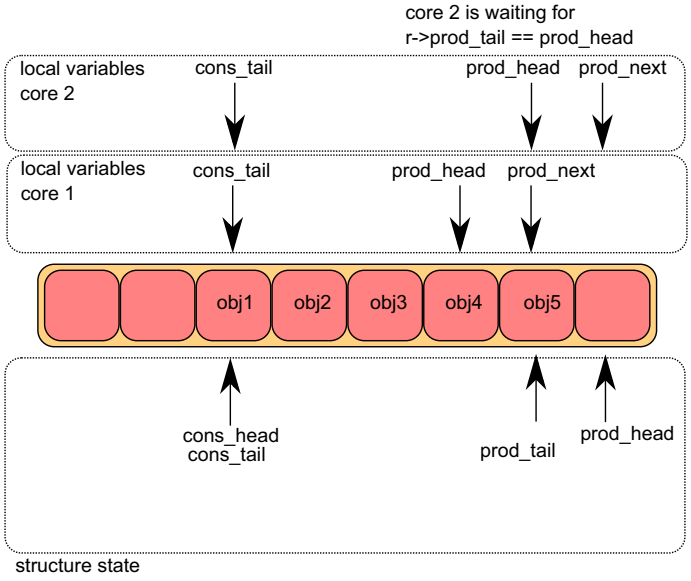
The CAS operation is retried on core 2 with success.

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



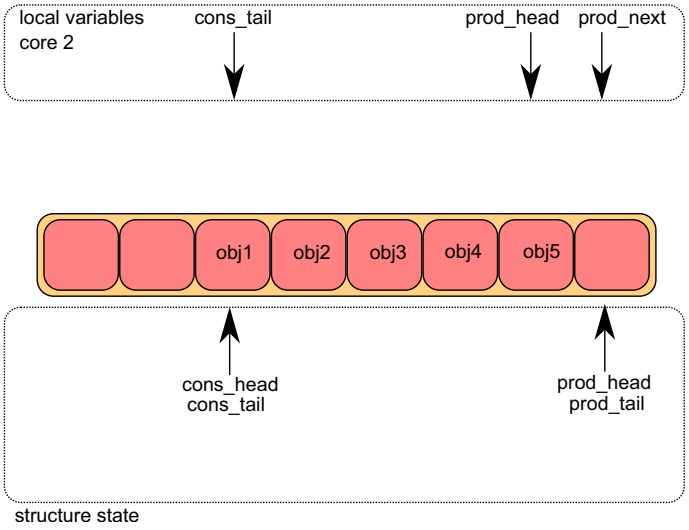
5.5.3.4 MC Enqueue Fourth Step

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.



5.5.3.5 MC 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.

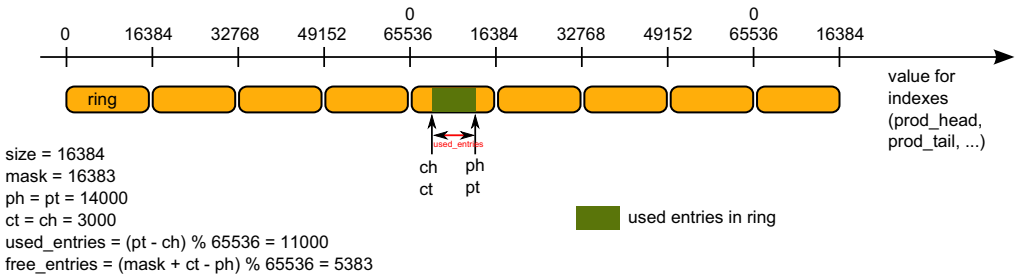


5.5.4 Modulo 32-bit Indexes

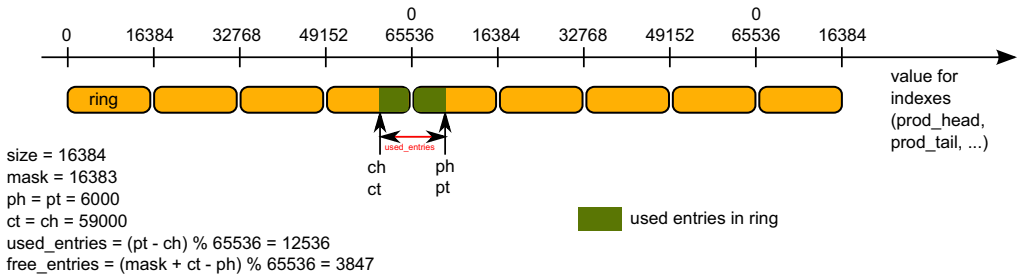
In the preceding figures, the `prod_head`, `prod_tail`, `cons_head` and `cons_tail` indexes are represented by arrows. In the actual implementation, these values are not between 0 and `size(ring)-1` as would be assumed. The indexes are between 0 and 2^{32}-1, and we mask their value when we access the 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.

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.

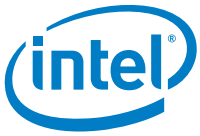
The code always maintains a distance between producer and consumer between 0 and `size(ring)-1`. Thanks to this property, we can do subtractions between 2 index values in a modulo-32bit base: that's why the overflow of the indexes is not a problem.

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);
```

5.6 References

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



6.0 Mempool Library

A memory pool is an allocator of a fixed-sized object. In the Intel® DPDK, it is identified by name and uses a ring to store free objects. 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 and the Environment Abstraction Layer (for logging history).

6.1 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.

6.2 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.

6.3 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.

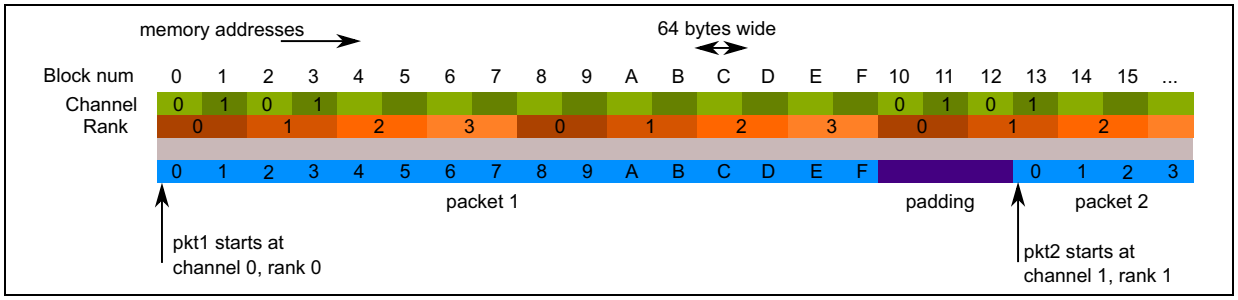
The number of ranks on any DIMM is the number of independent sets of DRAMs that can be accessed for the full data bit-width of the DIMM. The ranks cannot be accessed simultaneously since they share the same data path. The physical layout of the DRAM chips on the DIMM itself does not necessarily relate 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 Figure 4 and Figure 5.

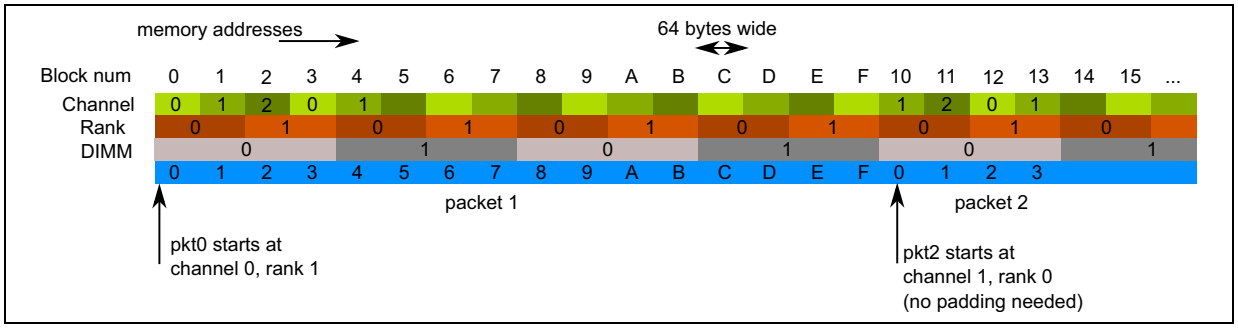
Figure 4. Two Channels and Quad-ranked DIMM Example



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).

Figure 5. Three Channels and Two Dual-ranked DIMM Example



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

6.4 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.

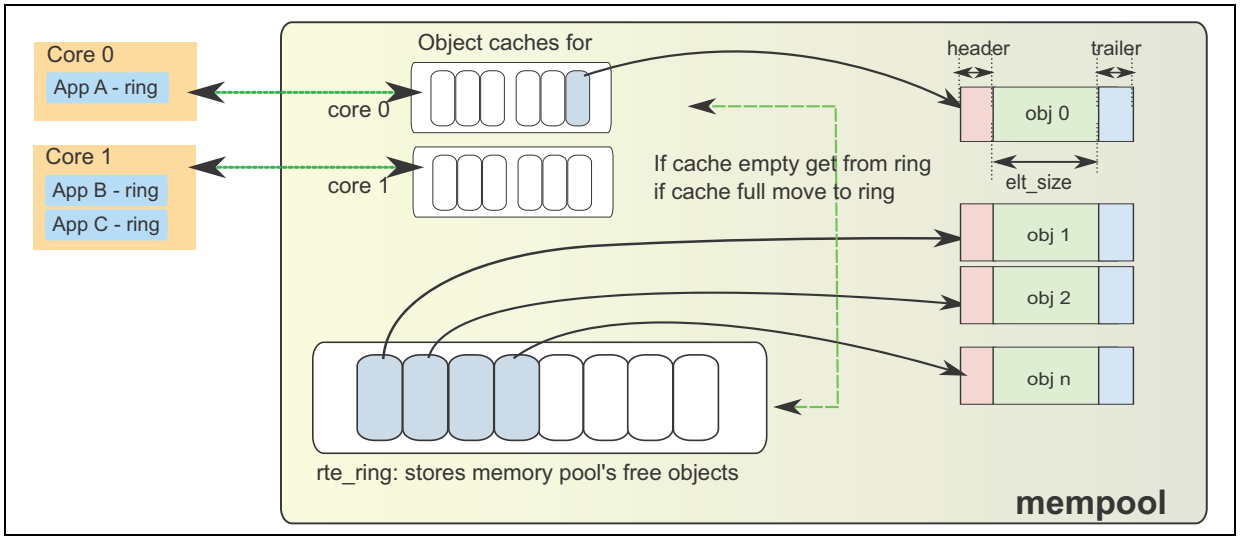
While this may mean a number of buffers may sit idle on some core's cache, the speed at which a core can access its own cache for a specific memory pool without locks provides performance gains.

The cache is composed of a small, per-core table of pointers and its length (used as a stack). This 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`).

Figure 6 shows a cache in operation.

Figure 6. A mempool in Memory with its Associated Ring



6.5 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.

§ §

7.0 Mbuf Library

The mbuf library provides the ability to allocate and free buffers (*mbufs*) that may be used by the Intel® 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 (type is `RTE_MBUF_PKT`) or generic control buffers (type is `RTE_MBUF_CTRL`). This can be extended to other types. The `rte_mbuf` is kept as small as possible (one cache line if possible).

7.1 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 Intel® DPDK. The metadata contains control information such as message type, length, pointer 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 `pkt.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 data pointers to point to other message buffer data sections or other structures.

Figure 7 and Figure 8 show some of these scenarios.

Figure 7. An mbuf with One Segment

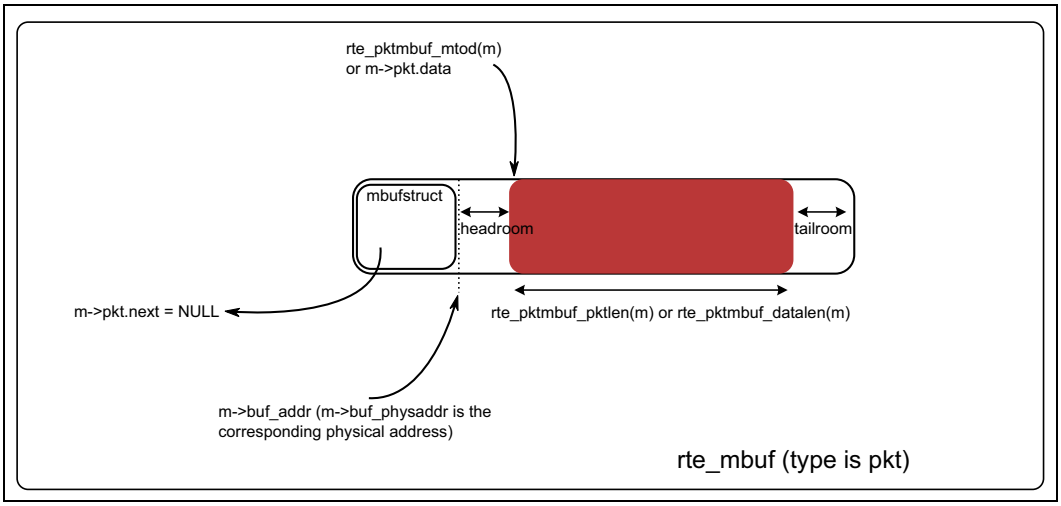
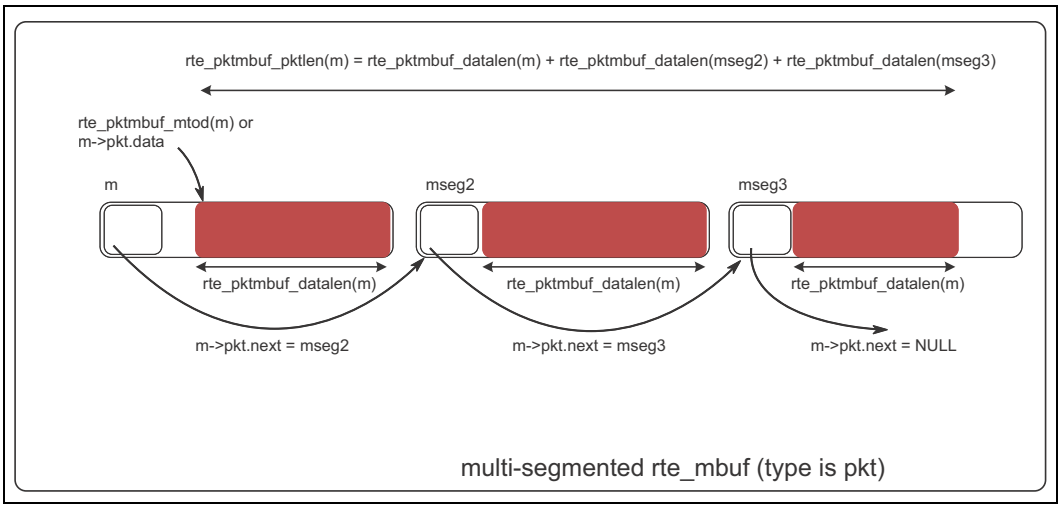


Figure 8. An mbuf with Three Segments



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

7.2 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.

7.3 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.

7.4 Allocating and Freeing mbufs

Allocating a new mbuf requires the user to specify the mempool from which the mbuf should be taken. For a packet mbuf, it contains one segment, with a length of 0. The pointer to data is initialized to have some bytes of headroom in the buffer (`RTE_PKTMBUF_HEADROOM`). For a control mbuf, it is initialized with data pointing to the beginning of the buffer and a length of zero.

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.

7.5 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 *Intel® DPDK IPL API Reference* for details.

7.6 Meta Information

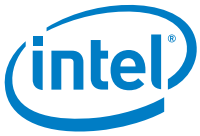
Some information is retrieved by the network driver and stored in an mbuf to make processing easier. For instance, the VLAN, the RSS hash result (see Poll Mode Driver) and a flag indicating that the checksum was computed by hardware.

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.

7.7 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 data pointer it contains points 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.

7.8 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).

7.9 Use Cases

All networking application should use mbufs to transport network packets.

§ §

8.0 Poll Mode Driver

The Intel® DPDK includes 1 Gigabit and 10 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.

8.1 Requirements and Assumptions

The Intel® 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 Intel® 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

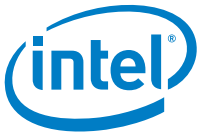
Conversely, in an asynchronous pipe-line model, some logical cores may be dedicated to the retrieval of received packets and other logical cores to the processing of previously received packets. Received packets are exchanged between logical cores through rings. The loop for packet retrieval includes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

8.2 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.

8.3 Logical Cores, Memory and NIC Queues Relationships

The Intel® 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.

8.4 Device Identification and Configuration

8.4.1 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 Intel® 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.

8.4.2 Device Configuration

The configuration of each NIC port includes the following operations:

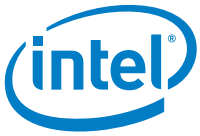
- Allocate PCI resources
- Reset the hardware (issue a Global Reset) to a well-known default state
- Set up the PHY and the link
- Initialize statistics counters

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.

8.4.3 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. Refer to the `testpmd` source code (specifically the `csumonly.c` file) for details.

That being said, the support of some 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.

For instance, this is the case for the IEEE1588 packet timestamp mechanism, the VLAN tagging and the IP checksum computation, as described in the Section 7.6, “Meta Information” on page 36.

8.4.4 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.

8.5 Poll Mode Driver API

8.5.1 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.

8.5.2 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.

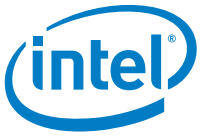
The rte_mbuf data structure includes specific fields to represent, in a generic way, the offload features provided by network controllers. For an input packet, most fields of the rte_mbuf structure are filled in by the PMD receive function with the information contained in the receive descriptor. Conversely,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

8.5.3 Ethernet Device API

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

§ §



9.0 IXGBE/IGB Virtual Function Driver

Supported Intel® Ethernet Controllers (see the *Intel® DPDK IPL Release Notes* for details) support the following modes of operation in a virtualized environment:

- **SR-IOV mode**: Involves direct assignment of part of the port resources to different guest operating systems using the PCI-SIG Single Root I/O Virtualization (SR IOV) standard, also known as "native mode" or "pass-through" mode. In this chapter, this mode is referred to as IOV mode.
- **VMDq mode**: Involves central management of the networking resources by an IO Virtual Machine (IOVM) or a Virtual Machine Monitor (VMM), also known as "software switch acceleration" mode. In this chapter, this mode is referred to as the Next Generation VMDq mode.

9.1 SR-IOV Mode Utilization in an Intel® DPDK Environment

The Intel® DPDK uses the SR-IOV feature for hardware-based I/O sharing in IOV mode. Therefore, it is possible to partition SR-IOV capability on Ethernet controller NIC resources logically and expose them to a virtual machine as a separate PCI function called a "Virtual Function". Refer to Figure 9.

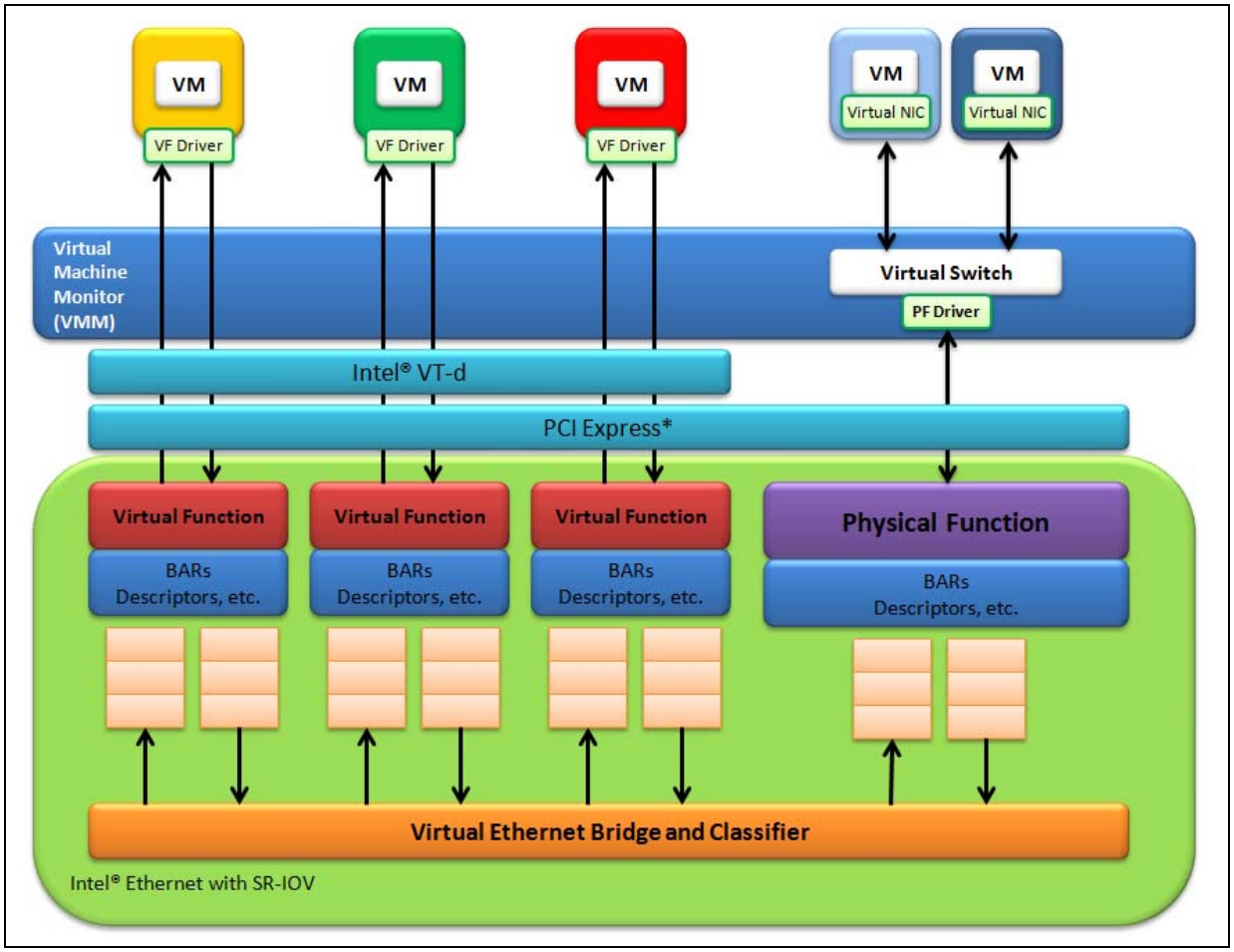
Therefore, a NIC is logically distributed among multiple virtual machines (as shown in Figure 9), while still having global data in common to share with the Physical Function and other Virtual Functions. The Intel® DPDK igbvf or ixgbevf as a Poll Mode Driver (PMD) serves for the Intel® 82576 Gigabit Ethernet Controller, Intel® Ethernet Controller I350 family, or Intel® 82599 10 Gigabit Ethernet Controller NIC's virtual PCI function. Meanwhile the Intel® DPDK Poll Mode Driver (PMD) also supports "Physical Function" of such NIC's on the host.

The Intel® DPDK PF/VF Poll Mode Driver (PMD) supports the Layer 2 switch on Intel® 82576 Gigabit Ethernet Controller, Intel® Ethernet Controller I350 family, and Intel® 82599 10 Gigabit Ethernet Controller NICs so that guest can choose it for inter virtual machine traffic in SR-IOV mode.

For more detail on SR-IOV, please refer to the following documents:

- SR-IOV provides hardware based I/O sharing
- PCI-SIG-Single Root I/O Virtualization Support on IA
- Scalable I/O Virtualized Servers

Figure 9. Virtualization for a Single Port NIC in SR-IOV Mode



9.1.1 Physical and Virtual Function Infrastructure

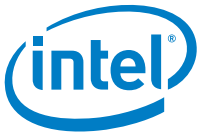
The following describes the Physical Function and Virtual Functions infrastructure for the supported Ethernet Controller NICs.

Virtual Functions operate under the respective Physical Function on the same NIC Port and therefore have no access to the global NIC resources that are shared between other functions for the same NIC port.

A Virtual Function has basic access to the queue resources and control structures of the queues assigned to it. For global resource access, a Virtual Function has to send a request to the Physical Function for that port, and the Physical Function operates on the global resources on behalf of the Virtual Function. For this out-of-band communication, an SR-IOV enabled NIC provides a memory buffer for each Virtual Function, which is called a "Mailbox".

9.1.1.1 Intel® 82599 10 Gigabit Ethernet Controller VF Infrastructure

The programmer can enable a maximum of *63 Virtual Functions* and there must be *one Physical Function* per Intel® 82599 10 Gigabit Ethernet Controller NIC port. The reason for this is that the device allows for a maximum of 128 queues per port and a



virtual/physical function has to have at least one queue pair (RX/TX). The current implementation of the Intel® DPDK ixgbevf driver supports a single queue pair (RX/TX) per Virtual Function. The Physical Function in host could be either configured by the Linux* ixgbe driver (in the case of the Linux Kernel-based Virtual Machine [KVM]) or by DPDK PMD PF driver. When using both DPDK PMD PF/VF drivers, the whole NIC will be taken over by DPDK based application.

For example,

- Using Linux* ixgbe driver:

  `rmmod ixgbe` (To remove the ixgbe module)
  `insmod ixgbe max_vfs=2,2` (To enable two Virtual Functions per port)

- Using the Intel® DPDK PMD PF ixgbe driver:

  Kernel Params: `iommu=pt, intel_iommu=on`
  `modprobe uio`
  `insmod igb_uio`
  `./pci_unbind.py –b igb_uio bb:ss.f`
  `echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs` (To enable two VFs on a specific PCI device)

  Launch the Intel® DPDK testpmd/example or your own host daemon application using the Intel® DPDK PMD library.

Virtual Function enumeration is performed in the following sequence by the Linux* `pci` driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

9.1.1.2 Intel® 82576 Gigabit Ethernet Controller and Intel® Ethernet Controller I350 Family VF Infrastructure

In a virtualized environment, an Intel® 82576 Gigabit Ethernet Controller serves up to eight virtual machines (VMs). The controller has 16 TX and 16 RX queues. They are generally referred to (or thought of) as queue pairs (one TX and one RX queue). This gives the controller 16 queue pairs.

A pool is a group of queue pairs for assignment to the same VF, used for transmit and receive operations. The controller has eight pools, with each pool containing two queue pairs, that is, two TX and two RX queues assigned to each VF.

In a virtualized environment, an Intel® Ethernet Controller I350 family device serves up to eight virtual machines (VMs) per port. The eight queues can be accessed by eight different VMs if configured correctly (the i350 has 4x1GbE ports each with 8T X and 8 RX queues), that means, one Transmit and one Receive queue assigned to each VF.

For example,

- Using Linux* igb driver:

  `rmmod igb` (To remove the igb module)
  `insmod igb max_vfs=2,2` (To enable two Virtual Functions per port)

- Using Intel® DPDK PMD PF igb driver:

  Kernel Params: `iommu=pt, intel_iommu=on`
  `modprobe uio`
  `insmod igb_uio`
  `./pci_unbind.py -b igb_uio bb:ss.f`
  `echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs` (To enable two VFs on a specific pci device)

  Launch Intel® DPDK testpmd/example or your own host daemon application using the Intel® DPDK PMD library.

Virtual Function enumeration is performed in the following sequence by the Linux* `pci` driver for a four-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence, starting from 0 to 7. However:

- Virtual Functions 0 and 4 belong to Physical Function 0
- Virtual Functions 1 and 5 belong to Physical Function 1
- Virtual Functions 2 and 6 belong to Physical Function 2
- Virtual Functions 3 and 7 belong to Physical Function 3

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

9.1.2 Validated Hypervisors

The validated hypervisor is:

- KVM (Kernel Virtual Machine) with `Qemu`, version 0.14.0

However, the hypervisor is bypassed to configure the Virtual Function devices using the Mailbox interface, the solution is hypervisor-agnostic. Xen* and VMware* (when SR-IOV is supported) will also be able to support the Intel® DPDK with Virtual Function driver support.

9.1.3 Expected Guest Operating System in Virtual Machine

The expected guest operating systems in a virtualized environment are:

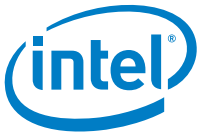
- Fedora* 14 (64-bit)
- Ubuntu* 10.04 (64-bit)

For supported kernel versions, refer to the *Intel® DPDK IPL Release Notes*.

9.2 Setting Up a KVM Virtual Machine Monitor

The following describes a target environment:

- Host Operating System: Fedora 14
- Hypervisor: KVM (Kernel Virtual Machine) with `Qemu` version 0.14.0
- Guest Operating System: Fedora 14
- Linux Kernel Version: Refer to the *Intel® DPDK IPL Getting Started Guide*
- Target Applications: `l2fwd-vf`, `l3fwd-vf`



The setup procedure is as follows:

1. Before booting the Host OS, open **BIOS setup** and **enable Intel® VT features**.
2. While booting the Host OS kernel, pass the `intel_iommu=on` kernel command line argument using GRUB. When using Intel® DPDK PF driver on host, pass the iommu=pt kernel command line argument in GRUB.
3. Download `qemu-kvm-0.14.0` from http://sourceforge.net/projects/kvm/files/qemu-kvm/ and install it in the Host OS using the following steps:

   When using a recent kernel (2.6.25+) with `kvm` modules included:

   ```
   tar xzf qemu-kvm-release.tar.gz
   cd qemu-kvm-release
   ./configure --prefix=/usr/local/kvm
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   When using an older kernel, or a kernel from a distribution without the `kvm` modules, you must download (from the same link), compile and install the modules yourself:

   ```
   tar xjf kvm-kmod-release.tar.bz2
   cd kvm-kmod-release
   ./configure
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   `qemu-kvm` installs in the /usr/local/bin directory.

   For more details about KVM configuration and usage, please refer to: http://www.linux-kvm.org/page/HOWTO1.
4. Create a Virtual Machine and install Fedora 14 on the Virtual Machine. This is referred to as the Guest Operating System (Guest OS).
5. Download and install the latest `ixgbe` driver from: http://downloadcenter.intel.com/Detail_Desc.aspx?agr=Y&DwnldID=14687
6. In the Host OS
   When using Linux kernel ixgbe driver, unload the Linux ixgbe driver and reload it with the max_vfs=2,2 argument:

   ```
   rmmod ixgbe
   "modprobe ixgbe max_vfs=2,2"
   ```

   When using DPDK PMD PF driver, insert Intel® DPDK kernel module `igb_uio` and set the number of VF by sysfs max_vfs:

   ```
   modprobe uio
   insmod igb_uio
   ./pci_unbind.py -b igb_uio 02:00.0 02:00.1 0e:00.0 0e:00.1
   echo 2 > /sys/bus/pci/devices/0000\:02\:00.0/max_vfs
   echo 2 > /sys/bus/pci/devices/0000\:02\:00.1/max_vfs
   echo 2 > /sys/bus/pci/devices/0000\:0e\:00.0/max_vfs
   echo 2 > /sys/bus/pci/devices/0000\:0e\:00.1/max_vfs
   ```

Note: You need to explicitly specify number of vfs for each port, for example, in the command above, it creates two vfs for the first two `ixgbe` ports.

Let say we have a machine with four physical ixgbe ports:

```
0000:02:00.0
0000:02:00.1
0000:0e:00.0
0000:0e:00.1
```

The command above creates two vfs for device 0000:02:00.0:

```
ls -alrt /sys/bus/pci/devices/0000\:02\:00.0/virt*

lrwxrwxrwx. 1 root root 0 Apr 13 05:40 /sys/bus/pci/devices/0000:02:00.0/
virtfn1 -> ../0000:02:10.2
lrwxrwxrwx. 1 root root 0 Apr 13 05:40 /sys/bus/pci/devices/0000:02:00.0/
virtfn0 -> ../0000:02:10.0
```

It also creates two vfs for device 0000:02:00.1:

```
ls -alrt /sys/bus/pci/devices/0000\:02\:00.1/virt*
lrwxrwxrwx. 1 root root 0 Apr 13 05:51 /sys/bus/pci/devices/0000:02:00.1/
virtfn1 -> ../0000:02:10.3
lrwxrwxrwx. 1 root root 0 Apr 13 05:51 /sys/bus/pci/devices/0000:02:00.1/
virtfn0 -> ../0000:02:10.1
```

7. List the PCI devices connected and notice that the Host OS shows two Physical Functions (traditional ports) and four Virtual Functions (two for each port). This is the result of the previous step.
8. Insert the pci_stub module to hold the PCI devices that are freed from the default driver using the following command (see http://www.linux-kvm.org/page/How_to_assign_devices_with_VT-d_in_KVM Section 4 for more information):

   ```
   sudo /sbin/modprobe pci-stub
   ```

   Unbind the default driver from the PCI devices representing the Virtual Functions. A script to perform this action is as follows:

   ```
   echo "8086 10ed" > /sys/bus/pci/drivers/pci-stub/new_id
   echo 0000:08:10.0 > /sys/bus/pci/devices/0000:08:10.0/driver/unbind
   echo 0000:08:10.0 > /sys/bus/pci/drivers/pci-stub/bind
   ```

   where, 0000:08:10.0 belongs to the Virtual Function visible in the Host OS.

9. Now, start the Virtual Machine by running the following command:

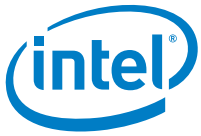
   ```
   /usr/local/kvm/bin/qemu-system-x86_64 -m 4096 -smp 4 -boot c -hda
   lucid.qcow2 -device pci-assign,host=08:10.0
   ```

   where:

   — -m = memory to assign
   — -smp = number of smp cores
   — -boot = boot option
   — -hda = virtual disk image
   — -device = device to attach

   Notes:

   — The pci-assign,host=08:10.0 value indicates that you want to attach a PCI device to a Virtual Machine and the respective (Bus:Device.Function) numbers should be passed for the Virtual Function to be attached.



— qemu-kvm-0.14.0 allows a maximum of four PCI devices assigned to a VM, but this is qemu-kvm version dependent since qemu-kvm-0.14.1 allows a maximum of five PCI devices.

— qemu-system-x86_64 also has a -cpu command line option that is used to select the cpu_model to emulate in a Virtual Machine. Therefore, it can be used as:

```
/usr/local/kvm/bin/qemu-system-x86_64 -cpu ?
```

(to list all available cpu_models)

```
/usr/local/kvm/bin/qemu-system-x86_64 -m 4096 -cpu host -smp 4 -boot c -hda
lucid.qcow2 -device pci-assign,host=08:10.0
```

(to use the same cpu_model equivalent to the host cpu)

For more information, please refer to: http://wiki.qemu.org/Features/CPUModels

10.

Install and run DPDK host app to take over the Physical Function. Eg.

```
make install T=x86_64-default-linuxapp-gcc
./x86_64-default-linuxapp-gcc/app/testpmd –c f –n 4 -- -i
```

11. Finally, access the Guest OS using vncviewer with the localhost:5900 port and check the lspci command output in the Guest OS. The virtual functions will be listed as available for use.

12. Configure and install the Intel® DPDK with an x86_64-default-linuxapp-gcc configuration on the Guest OS as normal, that is, there is no change to the normal installation procedure.

```
make config T=x86_64-default-linuxapp-gcc O=x86_64-default-linuxapp-gcc
cd x86_64-default-linuxapp-gcc
make
```

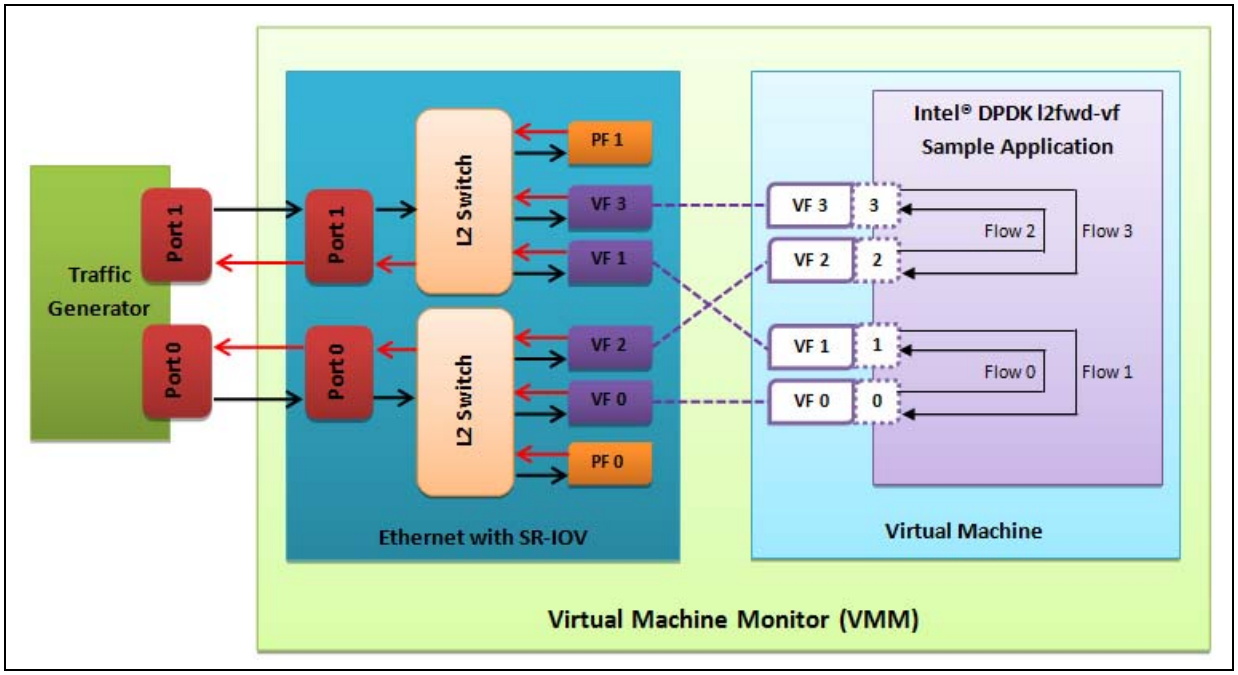
Note: If you are unable to compile the Intel® DPDK and you are getting "error: CPU you selected does not support x86-64 instruction set", power off the Guest OS and start the virtual machine with the correct -cpu option in the qemu-system-x86_64 command as shown in step 9. You must select the best x86_64 cpu_model to emulate or you can select host option if available.

13. Run the Intel® DPDK l2fwd-vf sample application in the Guest OS with Hugepages enabled. For the expected benchmark performance, you must pin the cores from the Guest OS to the Host OS (taskset can be used to do this) and you must also look at the PCI Bus layout on the board to ensure you are not running the traffic over the QPI Inteface.

Notes:

- The Virtual Machine Manager (the Fedora package name is virt-manager) is a utility for virtual machine management that can also be used to create, start, stop and delete virtual machines. If this option is used, step 2 and 6 in the instructions provided will be different.
- virsh, a command line utility for virtual machine management, can also be used to bind and unbind devices to a virtual machine in Ubuntu. If this option is used, step 6 in the instructions provided will be different.
- The Virtual Machine Monitor (see Figure 10) is equivalent to a Host OS with KVM installed as described in the instructions.

Figure 10. Performance Benchmark Setup



9.3 Intel® DPDK SR-IOV PMD PF/VF Driver Usage Model

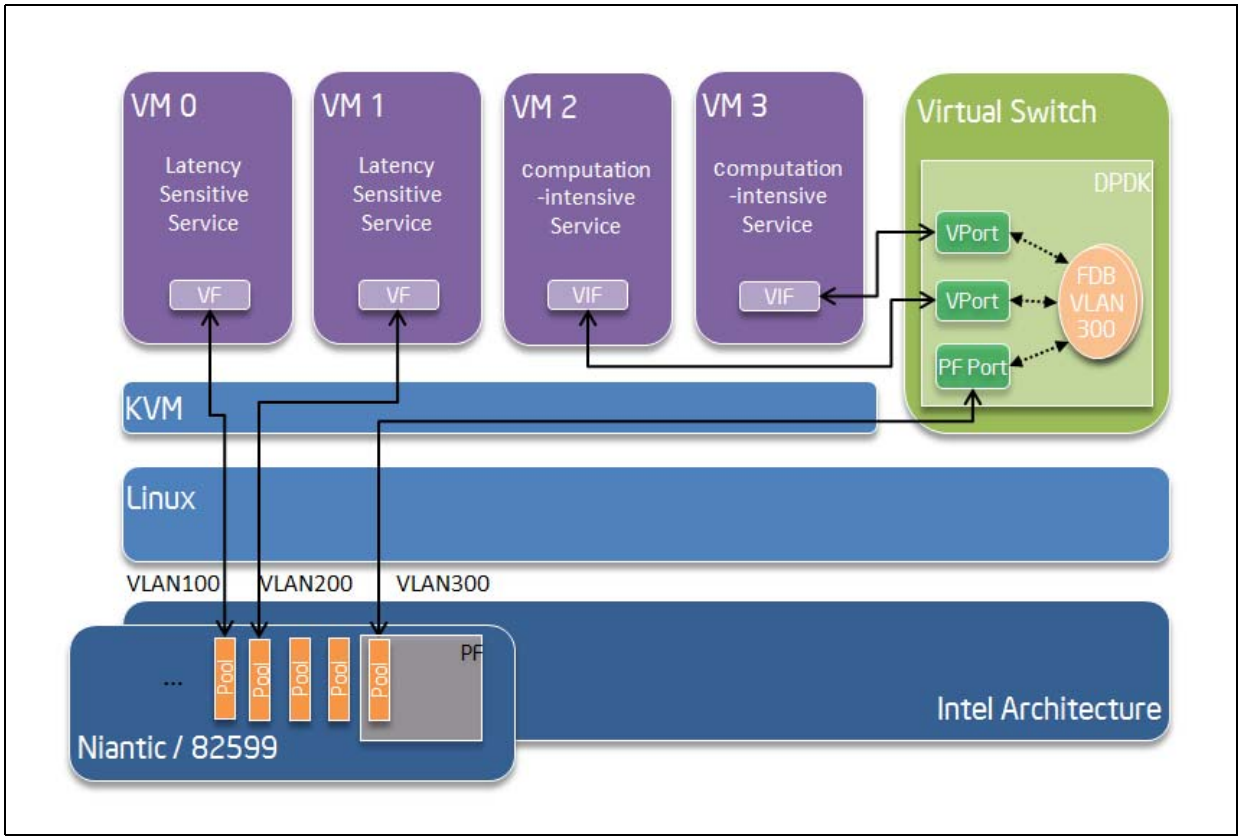
9.3.1 Fast Host-based Packet Processing

Software Defined Network (SDN) trends are demanding fast host-based packet handling. In a virtualization environment, the Intel® DPDK VF PMD driver performs the same throughput result as a non-VT native environment.

With such host instance fast packet processing, lots of services such as filtering, QoS, DPI can be offloaded on the host fast path.

Figure 11 shows the scenario where some VMs directly communicate externally via a VFs, while others connect to a virtual switch and share the same uplink bandwidth.

Figure 11. Fast Host-based Packet Processing



9.4 SR-IOV (PF/VF) Approach for Inter-VM Communication

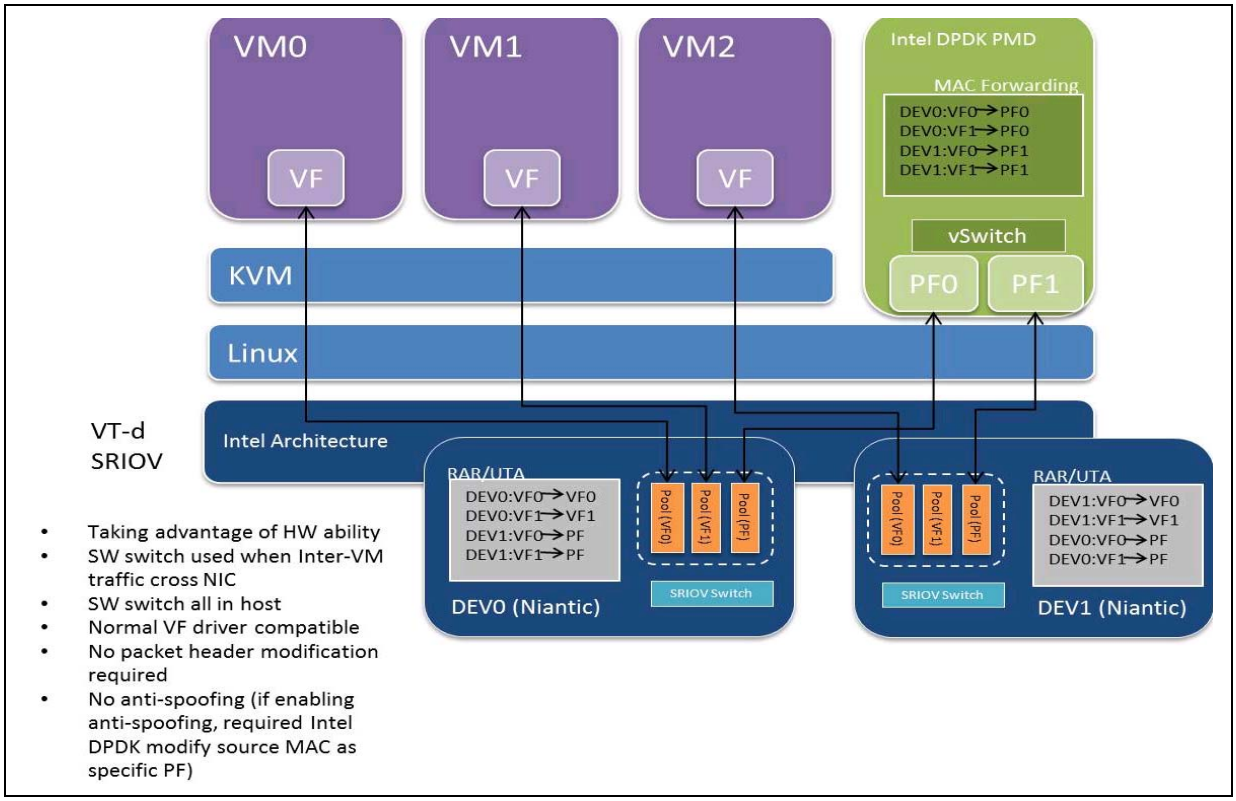
Inter-VM data communication is one of the traffic bottle necks in virtualization platforms. SR-IOV device assignment helps a VM to attach the real device, taking advantage of the bridge in the NIC. So VF-to-VF traffic within the same physical port (VM0<->VM1) have hardware acceleration. However, when VF crosses physical ports (VM0<->VM2), there is no such hardware bridge. In this case, the Intel® DPDK PMD PF driver provides host forwarding between such VMs.

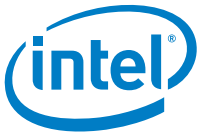
Figure 12 shows an example. In this case an update of the MAC address lookup tables in both the NIC and host Intel® DPDK application is required.

In the NIC, writing the destination of a MAC address belongs to another cross device VM to the PF specific pool. So when a packet comes in, its destination MAC address will match and forward to the host Intel® DPDK PMD application.

In the host Intel® DPDK application, the behavior is similar to L2 forwarding, that is, the packet is forwarded to the correct PF pool. The SR-IOV NIC switch forwards the packet to a specific VM according to the MAC destination address which belongs to the destination VF on the VM.

Figure 12. Inter-VM Communication





10.0 Driver for VM Emulated Devices

The Intel® DPDK EM poll mode driver supports the following emulated devices:

- qemu-kvm emulated Intel® 82540EM Gigabit Ethernet Controller (qemu e1000 device)
- VMware* emulated Intel® 82545EM Gigabit Ethernet Controller
- VMware emulated Intel® 8274L Gigabit Ethernet Controller.

10.1 Validated Hypervisors

The validated hypervisors are:

- KVM (Kernel Virtual Machine) with Qemu, version 0.14.0
- KVM (Kernel Virtual Machine) with Qemu, version 0.15.1
- VMware ESXi 5.0, Update 1

10.2 Recommended Guest Operating System in Virtual Machine

The recommended guest operating system in a virtualized environment is:

- Fedora* 18 (64-bit)

For supported kernel versions, refer to the *Intel® DPDK IPL Release Notes*.

10.3 Setting Up a KVM Virtual Machine

The following describes a target environment:

- Host Operating System: Fedora 14
- Hypervisor: KVM (Kernel Virtual Machine) with `Qemu` version, 0.14.0
- Guest Operating System: Fedora 14
- Linux Kernel Version: Refer to the *Intel® DPDK IPL Getting Started Guide*
- Target Applications: `testpmd`

The setup procedure is as follows:

1. 1. Download `qemu-kvm-0.14.0` from http://sourceforge.net/projects/kvm/files/qemu-kvm/ and install it in the Host OS using the following steps:

   When using a recent kernel (2.6.25+) with `kvm` modules included:

```
tar xzf qemu-kvm-release.tar.gz
cd qemu-kvm-release
./configure --prefix=/usr/local/kvm
make
```

```
sudo make install
sudo /sbin/modprobe kvm-intel
```

When using an older kernel or a kernel from a distribution without the `kvm` modules, you must download (from the same link), compile and install the modules yourself:

```
tar xjf kvm-kmod-release.tar.bz2
cd kvm-kmod-release
./configure
make
sudo make install
sudo /sbin/modprobe kvm-intel
```

Note that `qemu-kvm` installs in the `/usr/local/bin` directory.

For more details about KVM configuration and usage, please refer to: http://www.linux-kvm.org/page/HOWTO1.

2. Create a Virtual Machine and install Fedora 14 on the Virtual Machine. This is referred to as the Guest Operating System (Guest OS).
3. Start the Virtual Machine with at least one emulated e1000 device.

   Note that Qemu provides several choices for the emulated network device backend. Most commonly used is a TAP networking backend that uses a TAP networking device in the host. For more information about Qemu supported networking backends and different options for configuring networking at Qemu, please refer to:

   — http://www.linux-kvm.org/page/Networking
   — http://wiki.qemu.org/Documentation/Networking
   — http://qemu.weilnetz.de/qemu-doc.html

   For example, to start a VM with two emulated e1000 devices, issue the following command:

   ```
   /usr/local/kvm/bin/qemu-system-x86_64 -cpu host -smp 4 -hda qemu1.raw -m 1024
   -net nic,model=e1000,vlan=1,macaddr=DE:AD:1E:00:00:01
   -net tap,vlan=1,ifname=tapvm01,script=no,downscript=no
   -net nic,model=e1000,vlan=2,macaddr=DE:AD:1E:00:00:02
   -net tap,vlan=2,ifname=tapvm02,script=no,downscript=no
   ```

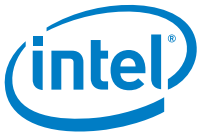
   where:

   — `-m` = memory to assign
   — `-smp` = number of smp cores
   — `-hda` = virtual disk image

   This command starts a new virtual machine with two emulated 82540EM devices, backed up with two TAP networking host interfaces, `tapvm01` and `tapvm02`.

   ```
   # ip tuntap show
   tapvm01: tap
   tapvm02: tap
   ```

4. Configure your TAP networking interfaces using `ip/ifconfig` tools.
5. Log in to the guest OS and check that the expected emulated devices exist:



```
# lspci -d 8086:100e

00:04.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet
Controller (rev 03)

00:05.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet
Controller (rev 03)
```

6. Install the Intel® DPDK and run `testpmd`.

10.4 Known Limitations of Emulated Devices

The following are know limitations:

1. The Qemu e1000 RX path does not support multiple descriptors/buffers per packet. Therefore, `rte_mbuf` should be big enough to hold the whole packet. For example, to allow `testpmd` to receive jumbo frames, use the following:

```
testpmd [options]   --    --mbuf-size=<your-max-packet-size>
```

2. Qemu e1000 does not validate the checksum of incoming packets.

§ §

11.0 Poll Mode Driver for Emulated Virtio NIC

virtio is a para-virtualization framework initiated by IBM and supported by the KVM hypervisor. In the Intel® Data Plane Development Kit (Intel® DPDK), we provide a virtio Poll Mode Driver (PMD) as a software solution, compared to an SRIOV hardware solution, for fast "guest VM to guest VM" communication and "guest VM to host" communication.

vhost is a kernel acceleration module for the virtio qemu backend. The Intel® DPDK extends kni to support the vhost raw socket interface, which enables vhost to directly read/write packets from/to a physical port. With this enhancement, virtio can achieve quite promising performance.

In future releases, enhancements to vhost backend are planned to realize peak performance from the virtio PMD driver.

For a basic qemu-KVM installation and other Intel EM poll mode drivers in the guest VM, please refer to Chapter 10.0, "Driver for VM Emulated Devices".

In this chapter, the use of the virtio PMD driver with two backends is demonstrated, 1) standard qemu vhost backend and 2) vhost kni backend.

11.1 Virtio Implementation in the Intel® DPDK

For details on virtio, refer to the *Virtio PCI Card Specification* written by Rusty Russell.

As a PMD, virtio provides the packet reception and transmission callbacks, virtio_recv_pkts and virtio_xmit_pkts.

In the virtio_recv_pkts callback, an index in the range [vq->vq_used_cons_idx , vq->vq_ring.used->idx] in vring is available for virtio to burst out.

In the virtio_xmit_pkts callback, the same index range in vring is available for virtio to clean. virtio will enqueue packets to be transmitted into vring, advance vq->vq_ring.avail->idx, then notify the host backend if necessary.

11.2 Features and Limitations of the Virtio PMD

In this release, the virtio PMD driver provides the basic functionality of packet reception and transmission.

- This release does not support mergeable buffers per packet for performance consideration. The packet size supported is from 64 to 1518. rte_mbuf should be big enough to hold the whole packet.
- The descriptor number for the RX/TX queue is hardcoded to be 256 by qemu. If given a different descriptor number by the upper application, the virtio PMD generates a warning and defaults to the hard-coded value.
- Features such as mac/vlan filter are not supported.

- RTE_PKTMBUF_HEADROOM should be defined larger than sizeof(struct virtio_net_hdr), which is 10 bytes.
- virtio does not support runtime configuration.

11.3 Prerequisites

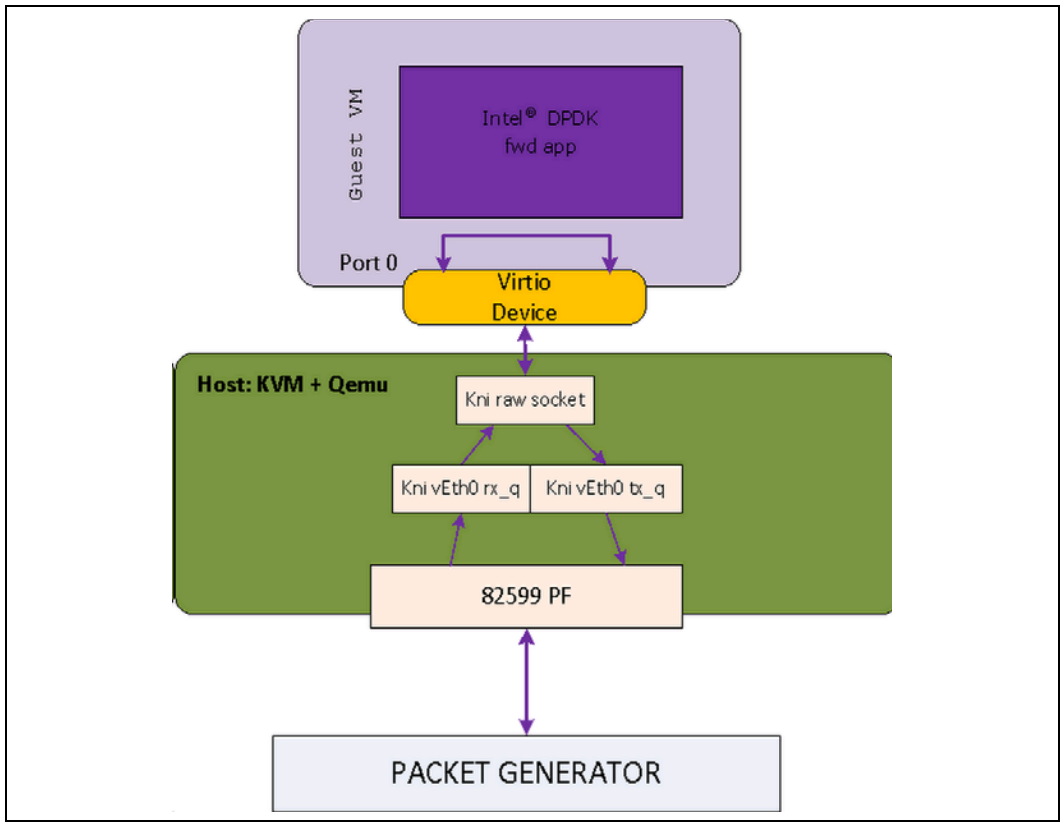
The following prerequisites apply:

- In the BIOS, turn VT-x on.
- Set intel_iommu=on and iommu=pt on the kernel command line.
- Linux kernel with KVM module; the vhost module is loaded and ioeventfd is supported. The qemu standard backend without vhost support is not tested and is not supported.

11.4 Virtio with kni vhost Backend

This section demonstrates a kni vhost backend example setup for Phy-VM communication.

Figure 13.



1. Load the kni kernel module:

```
insmod rte_kni.ko
```

Note: Other basic Intel® DPDK preparation such as, hugepage enabling, igb_uio port binding are not listed here. Please refer to the *Intel® DPDK IPL Getting Started Guide* for detailed instructions.

2. Launch the kni user application:

```
examples/kni/build/app/kni -c 0xf -n 4 -- -p 0x1 -i 0x1 -o 0x2
```

   This command generates one network device vEth0 for a physical port. If more physical ports are specified, the generated network devices are vEth1, vEth2 and so on.

   For each physical port, kni creates two user threads. One thread loops to fetch packets from the physical NIC port into the kni receive queue. The other user thread loops to send packets in the kni transmit queue.

   For each physical port, kni also creates a kernel thread that retrieves packets from the kni receive queue, places them onto kni's raw socket's queue and wakes up the vhost kernel thread to exchange packets with the virtio virt queue.

   For more details about kni, please refer to Chapter 17.0, "Kernel NIC Interface"

3. Enable the kni raw socket functionality for the specified physical NIC port, get the generated file descriptor and set it in the qemu command line parameter.

   Always remember to set both ioeventfd_on and vhost_on.

   For example:

```
echo 1 > /sys/class/net/vEth0/sock_en
fd=`cat /sys/class/net/vEth0/sock_fd`
exec qemu-system-x86_64 -enable-kvm -cpu host \
-m 2048 -smp 4  -name dpdk-test1-vm1 \
-drive file=/data/DPDKVMS/dpdk-vm.img  \
-netdev tap, fd=$fd,id=mynet_kni, script=no,vhost=on \
-device virtio-net-pci,netdev=mynet_kni,bus=pci.0,addr=0x3,ioeventfd=on \
-vnc:1  -daemonize
```

Note: In the above example, virtio port 0 in the guest VM will be associated with vEth0, which in turn corresponds to a physical port, which means that received packets come from vEth0, and transmitted packets are sent to vEth0.

4. In the guest, bind the virtio device to the igb_uio kernel module and start the forwarding application.

   When the virtio port in guest bursts rx, it gets packets from the raw socket's receive queue. When the virtio port bursts tx, it sends packets to the tx_q.

```
modprobe uio
echo 512 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
insmod x86_64-default-linuxapp-gcc/kmod/igb_uio.ko
python tools/pci_unbind.py -b igb_uio 00:03.0
```

   Start testpmd as the forwarding application.

5. Use the packet generator to inject a packet stream into the KNI physical port.

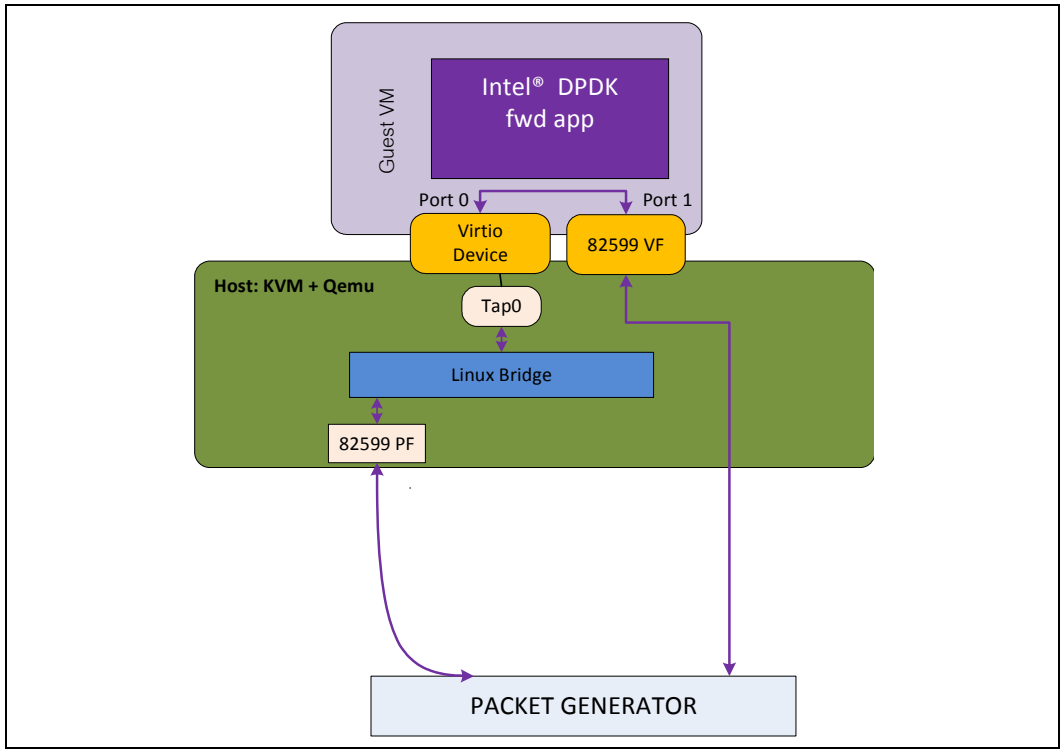
The packet reception and transmission flow path is:

Packet generator -> 82599 PF -> KNI rx queue -> KNI raw socket queue -> Guest VM virtio port 0 rx burst -> Guest VM virtio port 0 tx burst -> KNI tx queue -> 82599 PF -> packet generator

11.5 Virtio with a qemu virtio Backend

The following figure shows a host to VM communication example using a `qemu vhost` backend.

Figure 14. Host2VM Communication Example Using a qemu vhost Backend



The following is an example:

```
qemu-system-x86_64 -enable-kvm -cpu host -m 2048 -smp 2 -mem-path /dev/
      hugepages -mem-prealloc
-drive file=/data/DPDKVMS/dpdk-vm1
-netdev tap,id=vm1_p1,ifname=tap0,script=no,vhost=on
-device virtio-net-pci,netdev=vm1_p1,bus=pci.0,addr=0x3,ioeventfd=on
-device pci-assign,host=04:10.1 \
```

In this example, the packet reception flow path is:

Packet generator -> 82599 PF -> Linux Bridge -> TAP0's socket queue -> Guest VM virtio port 0 rx burst -> Guest VM 82599 VF port1 tx burst -> packet generator

The packet transmission flow path is:

Packet generator -> Guest VM 82599 VF port1 rx burst -> Guest VM virtio port 0 tx burst -> tap -> Linux Bridge -> 82599 PF -> packet generator

§ §

12.0 Libpcap and Ring Based Poll Mode Drivers

In addition to Poll Mode Drivers (PMDs) for physical and virtual hardware, the Intel® DPDK also includes two pure-software PMDs. These two drivers are:

- A `libpcap`-based PMD (`librte_pmd_pcap`) that reads and writes packets using `libpcap`, - both from files on disk, as well as from physical NIC devices using standard Linux kernel drivers.
- A ring-based PMD (`librte_pmd_ring`) that allows a set of software FIFOs (that is, rte_ring) to be accessed using the PMD APIs, as though they were physical NICs.

Note: The `libpcap`-based PMD is disabled by default in the build configuration files, owing to an external dependency on the `libpcap` development files which must be installed on the board. Once the `libpcap` development files are installed, the library can be enabled by setting `CONFIG_TRTE_LIBRTE_PNMD_PCAP=y` and recompiling the Intel® DPDK.

12.1 Using the Drivers from the EAL Command Line

For ease of use, the Intel® DPDK EAL also has been extended to allow pseudo-ethernet devices, using one or more of these drivers, to be created at application startup time during EAL initialization.

To do so, the `--use-device=` parameter must be passed to the EAL. This takes the list of physical PCI device IDs to be used by the application as the parameter value, that is, a whitelist of NIC ports, but also can take options to allow ring and pcap-based Ethernet to be allocated and used transparently by the application. This can be used, for example, for testing on a virtual machine where there are no Ethernet ports.

12.1.1 Libpcap-based PMD

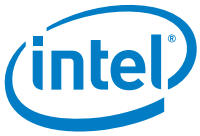
Pcap-based devices can be created using the whitelist `--use-device` option. The device name must start with the `eth_pcap` prefix followed by numbers or letters. The name is unique for each device. Each device can have multiple stream options and multiple devices can be used. Multiple device definitions can be arranged in a comma separated list. Device name and stream options must be separated by semicolons as shown below:

```
$RTE_TARGET/app/testpmd -c f -n 4 --use-device
'eth_pcap0;stream_opt0=..;stream_opt1=..,eth_pcap1;stream_opt0=..'
```

12.1.1.1 Device Streams

Multiple ways of stream definitions can be assessed and combined as long as the following two rules are respected:

- A device is provided with two different streams – reception and transmission.
- A device is provided with one network interface name used for reading and writing packets.



The different stream types are:

- `rx_pcap`**:** Defines a reception stream based on a `pcap` file. The driver reads each packet within the given `pcap` file as if it was receiving it from the wire. The value is a path to a valid `pcap` file.

  ```
  rx_pcap=/path/to/file.pcap
  ```

- `tx_pcap`: Defines a transmission stream based on a pcap file. The driver writes each received packet to the given pcap file. The value is a path to a pcap file. The file is overwritten if it already exists and it is created if it does not.

  ```
  tx_pcap=/path/to/file.pcap
  ```

- `rx_iface`: Defines a reception stream based on a network interface name. The driver reads packets coming from the given interface using the Linux kernel driver for that interface. The value is an interface name.

  ```
  rx_iface=eth0
  ```

- `tx_iface`: Defines a transmission stream based on a network interface name. The driver sends packets to the given interface using the Linux kernel driver for that interface. The value is an interface name.

  ```
  tx_iface=eth0
  ```

- `iface`: Defines a device mapping a network interface. The driver both reads and writes packets from and to the given interface. The value is an interface name.

  ```
  iface=eth0
  ```

12.1.1.2 Examples of Usage

Read packets from one `pcap` file and write them to another:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --use-device 'eth_pcap0;rx_pcap=/path/to/
file_rx.pcap;tx_pcap=/path/to/file_tx.pcap' -- --port-topology=chained
```

Read packets from a network interface and write them to a `pcap` file:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --use-device
'eth_pcap0;rx_iface=eth0;tx_pcap=/path/to/file_tx.pcap' -- --port-topology=chained
```

Read packets from a `pcap` file and write them to a network interface:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --use-device 'eth_pcap0;rx_pcap=/path/to/
file_rx.pcap;tx_iface=eth1' -- --port-topology=chained
```

Forward packets through two network interfaces:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --use-device
'eth_pcap0;iface=eth0,eth_pcap1;iface=eth1'
```

12.1.1.3 Using libpcap-based PMD with the testpmd Application

One of the first things that `testpmd` does before starting to forward packets is to flush the RX streams by reading the first 512 packets on every RX stream and discarding them. When using a `libpcap`-based PMD this behavior can be turned off using the following command line option:

```
--no-flush-rx
```

It is also available in the runtime command line:

```
set flush_rx on/off
```

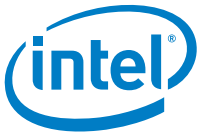
It is useful for the case where the `rx_pcap` is being used and no packets are meant to be discarded. Otherwise, the first 512 packets from the input pcap file will be discarded by the RX flushing operation.

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --use-device 'eth_pcap0;rx_pcap=/path/to/
file_rx.pcap;tx_pcap=/path/to/file_tx.pcap' -- --port-topology=chained --no-flush-rx
```

12.1.2 Rings-based PMD

To run an Intel® DPDK application on a machine without any Ethernet devices, a pair of ring-based `rte_ethdevs` can be used as below. The device names passed to the `--use-device` option must start with `eth_ring` and take no additional parameters. Multiple devices may be specified, separated by commas.

```
./testpmd -c E -n 4 --use-device=eth_ring0,eth_ring1 -- -i
EAL: Detected lcore 1 as core 1 on socket 0
...
Interactive-mode selected
Configuring Port 0 (socket 0)
Configuring Port 1 (socket 0)
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done

testpmd> start tx_first
   io packet forwarding - CRC stripping disabled - packets/burst=16
   nb forwarding cores=1 - nb forwarding ports=2
   RX queues=1 - RX desc=128 - RX free threshold=0
   RX threshold registers: pthresh=8 hthresh=8 wthresh=4
   TX queues=1 - TX desc=512 - TX free threshold=0
   TX threshold registers: pthresh=36 hthresh=0 wthresh=0
   TX RS bit threshold=0 - TXQ flags=0x0
testpmd> stop
Telling cores to stop...
Waiting for lcores to finish...

  ---------------------- Forward statistics for port 0  ----------------------
  RX-packets: 231192368      RX-dropped: 0             RX-total: 231192368
  TX-packets: 231192384      TX-dropped: 0             TX-total: 231192384
  ----------------------------------------------------------------------------

  ---------------------- Forward statistics for port 1  ----------------------
  RX-packets: 231192368      RX-dropped: 0             RX-total: 231192368
  TX-packets: 231192384      TX-dropped: 0             TX-total: 231192384
  ----------------------------------------------------------------------------
```



```
++++++++++++++ Accumulated forward statistics for allports++++++++++
RX-packets: 462384736      RX-dropped: 0             RX-total: 462384736
TX-packets: 462384768      TX-dropped: 0             TX-total: 462384768
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Done.
```

12.1.3 Using the Poll Mode Driver from an Application

Both drivers can provide similar APIs to allow the user to create a PMD, that is, `rte_ethdev` structure, instances at run-time in the end-application, for example, using `rte_eth_from_rings()` or `rte_eth_from_pcaps()` APIs. For the rings-based PMD, this functionality could be used, for example, to allow data exchange between cores using rings to be done in exactly the same way as sending or receiving packets from an Ethernet device. For the `libpcap`-based PMD, it allows an application to open one or more pcap files and use these as a source of packet input to the application.

12.1.3.1 Usage Examples

To create two pseudo-ethernet ports where all traffic sent to a port is looped back for reception on the same port (error handling omitted for clarity):

```
struct rte_ring *r1, *r2;
int port1, port2;
r1 = rte_ring_create("R1", 256, SOCKET0, RING_F_SP_ENQ|RING_F_SC_DEQ);
r2 = rte_ring_create("R2", 256, SOCKET0, RING_F_SP_ENQ|RING_F_SC_DEQ);

/* create an ethdev where RX and TX are done to/from r1, and
* another from r2*/
port1 = rte_eth_from_rings(r1, 1, r1, 1, SOCKET0);
port2 = rte_eth_from_rings(r2, 1, r2, 1, SOCKET0);
```

To create two pseudo-Ethernet ports where the traffic is switched between them, that is, traffic sent to port 1 is read back from port 2 and vice-versa, the final two lines could be changed as below:

```
port1 = rte_eth_from_rings(r1, 1, r2, 1, SOCKET0);
port2 = rte_eth_from_rings(r2, 1, r1, 1, SOCKET0);
```

This type of configuration could be useful in a pipeline model, for example, where one may want to have inter-core communication using pseudo Ethernet devices rather than raw rings, for reasons of API consistency.

Enqueuing and dequeuing items from an `rte_ring` using the rings-based PMD may be slower than using the native rings API. This is because Intel® DPDK Ethernet drivers make use of function pointers to call the appropriate enqueue or dequeue functions, while the `rte_ring` specific functions are direct function calls in the code and are often inlined by the compiler.

Once an `ethdev` has been created, for either a ring or a pcap-based PMD, it should be configured and started in the same way as a regular Ethernet device, that is, by calling `rte_eth_dev_configure()` to set the number of receive and transmit queues, then calling `rte_eth_rx_queue_setup()/tx_queue_setup()` for each of those queues and finally calling `rte_eth_dev_start()` to allow transmission and reception of packets to begin.

§ §

13.0 Timer Library

The Timer library provides a timer service to Intel® 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 CPU's 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.

13.1 Implementation Details

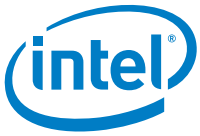
Timers are tracked on a per-lcore basis, with all pending timers for a core being maintained in order of timer expiry in a `skiplist` data structure. The `skiplist` used has ten levels and each entry in the table appears in each level with probability ¼^level. This means that all entries are present 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 favour 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.

13.2 Use Cases

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

13.3 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).

§ §

14.0 Hash Library

The Intel® 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 Intel® DPDK Hash requires that all the keys have the same number of bytes which is set at the hash creation time.

14.1 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:

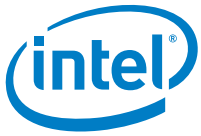
- Hash function to translate the key into a bucket index
- Number of entries per bucket

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.

The current hash implementation handles the key management only. The actual data associated with each key has to be 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.

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.



14.2 Implementation Details

The hash table is implemented as an array of entries which is further divided into buckets, with the same number of consecutive array entries in each bucket. For any input key, there is always a single bucket where that key can be stored in the hash, therefore only the entries within that 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 a hash bucket, 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 bucket is identified, the scope of the hash add, delete and lookup operations is reduced to the entries in that 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.

14.3 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 Intel® 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.

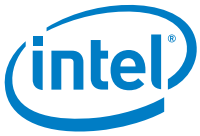
The flow table operations on the application side are described below:

- 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.

14.4 References

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

§ §



15.0 LPM Library

The Intel® 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.

15.1 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.

15.2 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 has 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.

15.3 Use Case: IPv4 Forwarding

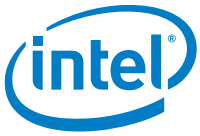
The LPM algorithm is used to implement Classless Inter-Domain Routing (CIDR) strategy used by routers implementing IPv4 forwarding.

15.4 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`)

§ §

§ §



16.0 Multi-process Support

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

The EAL has been modified to allow different types of Intel® 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 Intel® 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 Intel® 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 Intel® DPDK processes can be used together. These are more fully documented in the "Multi-process Sample Application" chapter in the *Intel® DPDK IPL Sample Application's User Guide*.

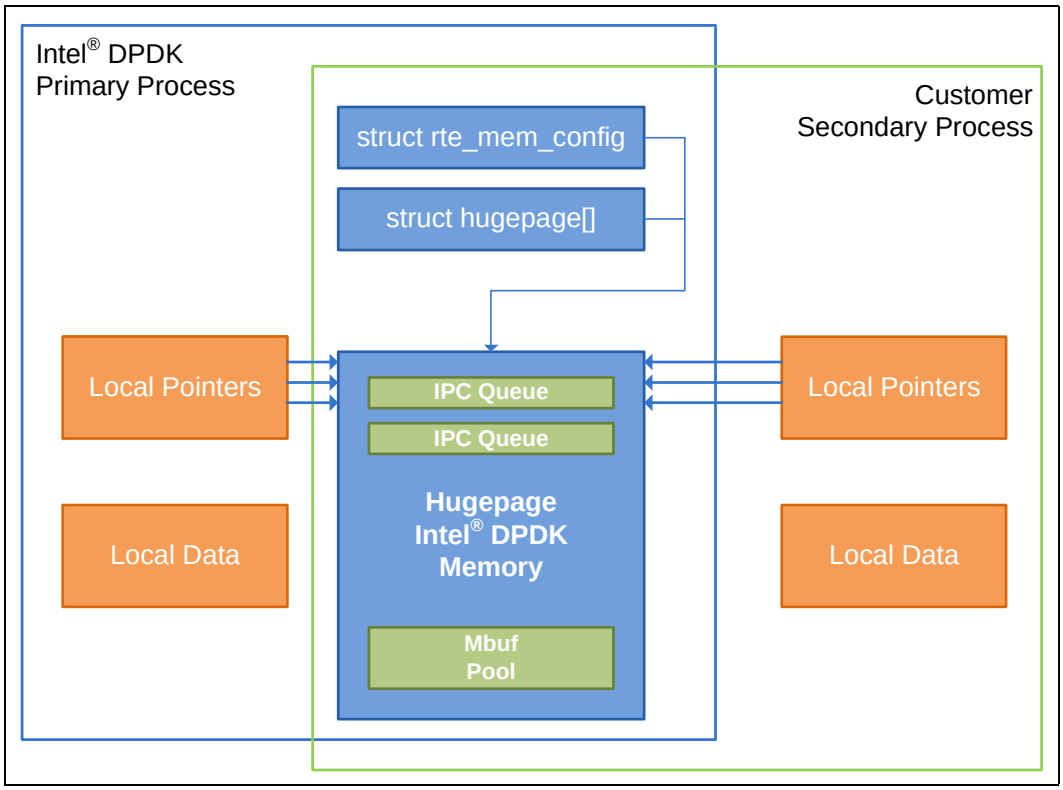
16.1 Memory Sharing

The key element in getting a multi-process application working using the Intel® 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 Intel 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 the Section 16.3, "Multi-process Limitations" on page 74 for details of how Linux kernel Address-Space Layout Randomization (ASLR) can affect memory sharing.

Figure 15. Memory Sharing in the Intel® DPDK Multi-process Sample Application



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

16.2 Deployment Models

16.2.1 Symmetric/Peer Processes

Intel® 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 Intel® 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 *Intel® DPDK IPL Sample Application's User Guide*.

16.2.2 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 *Intel® DPDK IPL Sample Application's User Guide*.

16.2.3 Running Multiple Independent Intel® DPDK Applications

In addition to the above scenarios involving multiple Intel® DPDK processes working together, it is possible to run multiple Intel® 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 Intel® 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 Intel® 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.

16.2.4 Running Multiple Independent Groups of Intel® DPDK Applications

In the same way that it is possible to run independent Intel® DPDK applications side-by-side on a single system, this can be trivially extended to multi-process groups of Intel® 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 Intel® DPDK processes running side-by-side apply in this usage scenario also.

16.3 Multi-process Limitations

There are a number of limitations to what can be done when running Intel® 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 Intel® 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 Intel® DPDK processes used, it may not be possible to have HPET timers available in each Intel® 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 Intel® DPDK process instance can open and mmap `/dev/hpet`. If the number of required Intel® 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.

§ §

17.0 Kernel NIC Interface

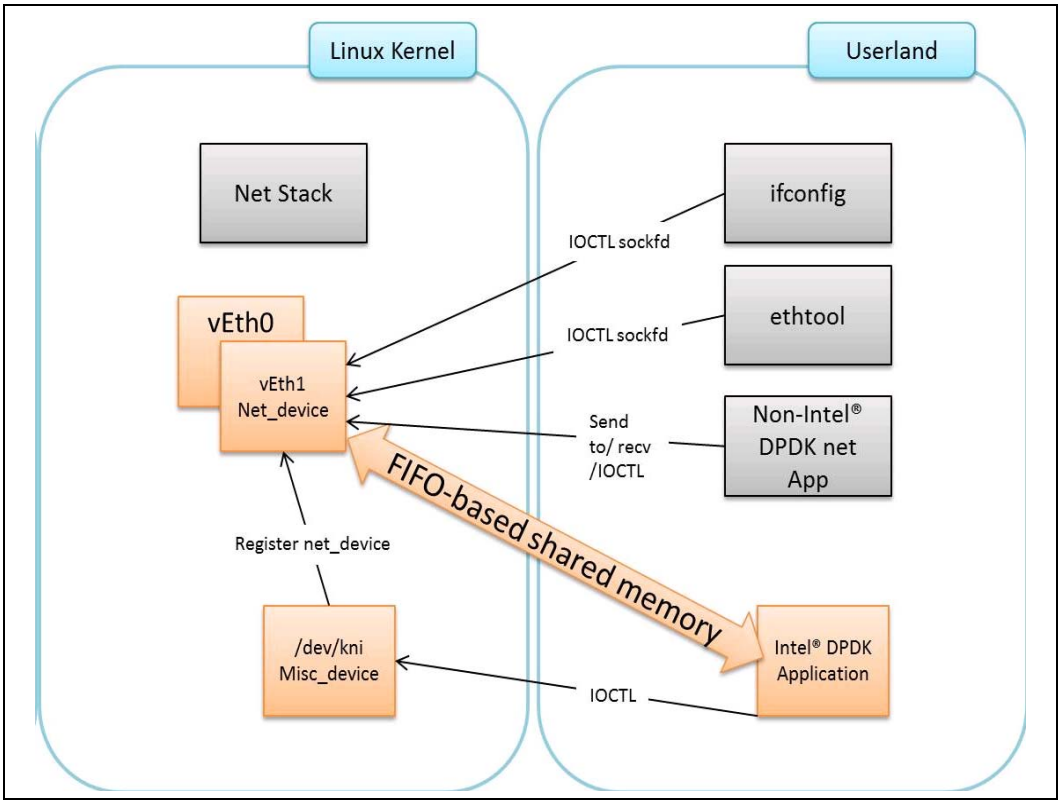
The Intel® DPDK Kernel NIC Interface (KNI) allows userspace applications access to the Linux* control plane.

The benefits of using the Intel® 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 Intel® 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 Intel® DPDK Kernel NIC Interface are shown in Figure 16.

Figure 16. Components of an Intel® DPDK KNI Application



17.1 The Intel® 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 Intel® DPDK mbufs and FIFOs.
  - The interface name is provided from userspace.
  - The MAC address can be the real NIC MAC address or random.

17.2 KNI Creation and Deletion

The KNI interfaces are created by an Intel® 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 Intel® DPDK source code for more details.

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

Once KNI interfaces are created, the KNI context information can be queried by calling the `rte_kni_info_get()` function.

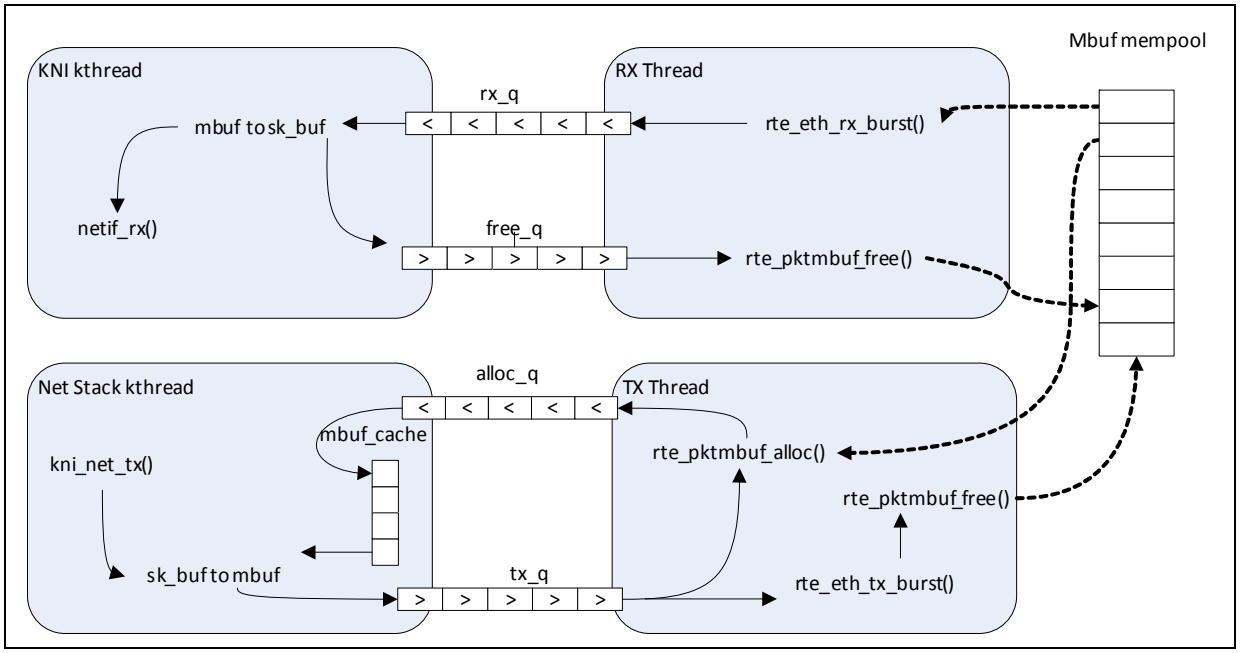
The KNI interfaces can be deleted by an Intel® DPDK application dynamically after being created. Furthermore, all those KNI interfaces not deleted will be deleted on the release operation of the miscellaneous device (when the Intel® DPDK application is closed).

17.3 Intel® DPDK mbuf Flow

To minimize the amount of Intel® 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 Intel® DPDK application only.

Figure 17 shows a typical scenario with packets sent in both directions.

Figure 17. Packet Flow via mbufs in the Intel® DPDK KNI



17.4 Use Case: Ingress

On the Intel® 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.

17.5 Use Case: Egress

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

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 Intel® 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.

17.6 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 VMs (VF or EM devices).

17.7 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 Intel® 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.

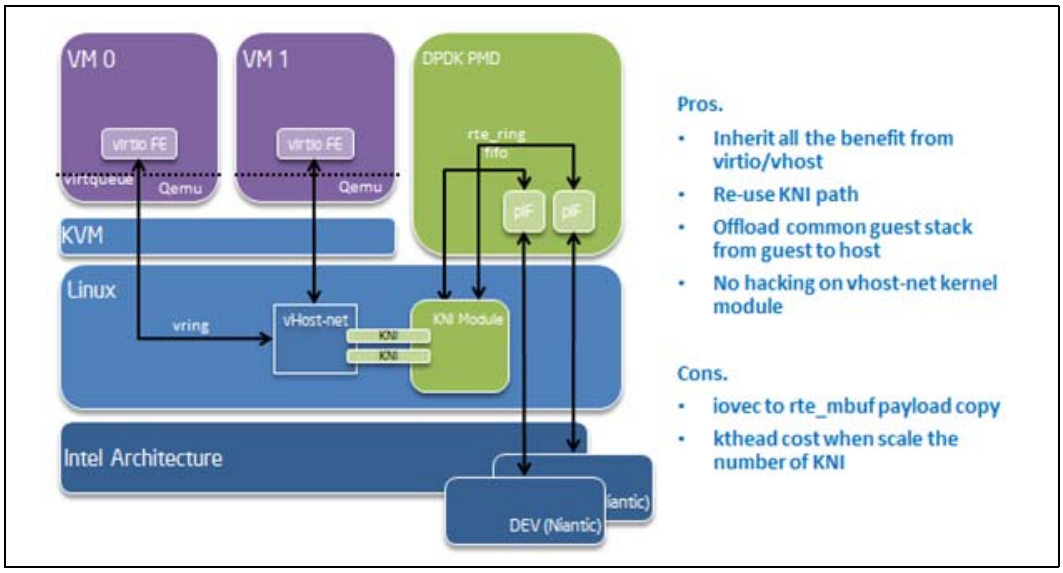
17.8 KNI Working as a Kernel vHost Backend

`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 Intel® DPDK Kernel NIC interface provides the ability to hookup `vHost` traffic into userspace Intel® DPDK application. Together with the Intel® DPDK PMD `virtio`, it significantly improves the throughput between guest and host. In the scenario where Intel® DPDK is running as fast path in the host, `kni-vhost` is an efficient path for the traffic.

17.8.1 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 Intel® 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.

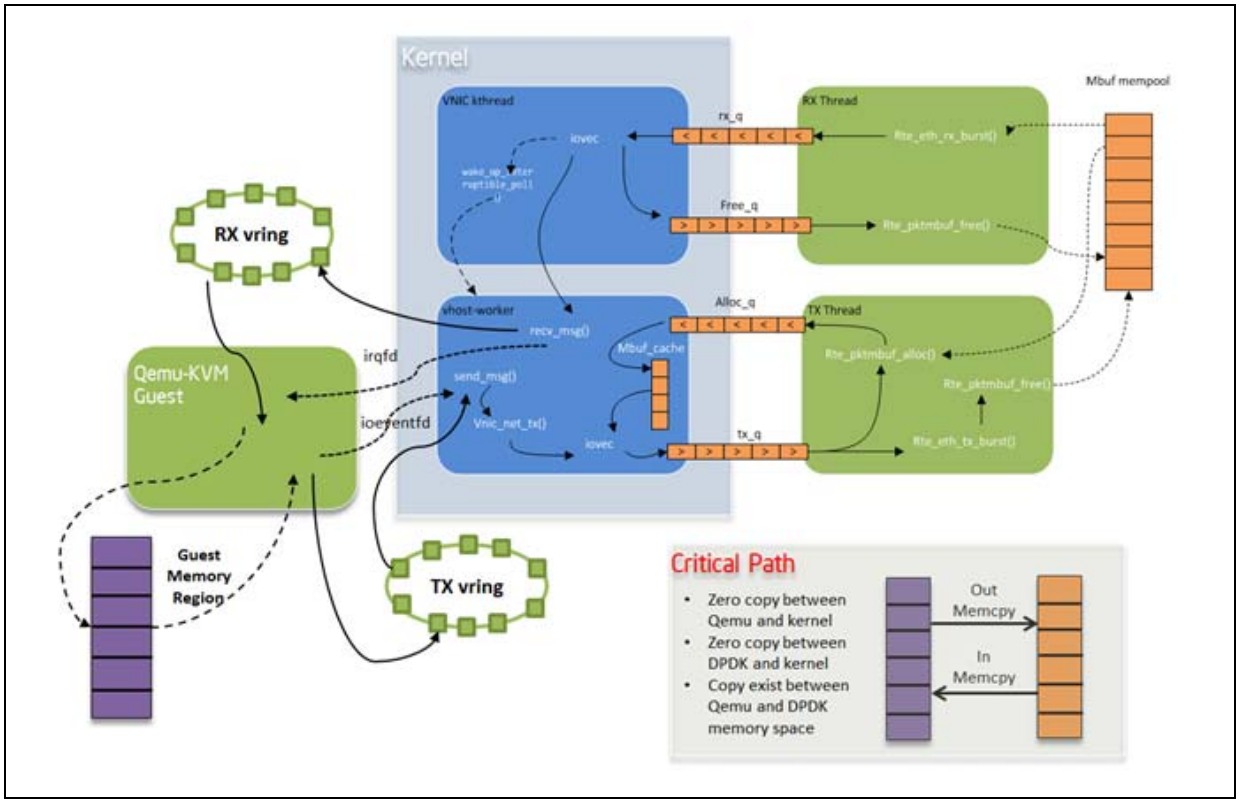
Figure 18. vHost-net Architecture Overview



17.8.2 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.

Figure 19. KNI Traffic Flow



17.8.3 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 Intel® DPDK and insert `igb_uio` 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:

```
./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 Intel® 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
```

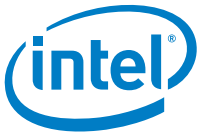
   It is simple to enable raw socket using sysfs sock_en and get raw socket fd using sock_fd under the KNI device node.
   Then, using the qemu-kvm command with the -netdev option to assign such raw socket fd as vhost's backend.
   Note: The key word tap must exist as qemu-kvm now only supports vhost with a tap beckend, so here we cheat qemu-kvm by an existing fd.

17.8.4 Compatibility Configure Option

There is a CONFIG_RTE_KNI_VHOST_VNET_HDR_EN configuration option in Intel® 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.

§ §



18.0 Thread Safety of Intel® DPDK Functions

The Intel® 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 Intel® 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.

18.1 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 an Intel® 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 Intel® DPDK lockless `ring` library and therefore is also multi-thread safe.

18.2 Performance Insensitive API

Outside of the performance sensitive areas described in Section 18.1, the Intel® DPDK provides a thread-safe API for most other libraries. For example, `malloc(librte_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.

18.3 Library Initialization

It is recommended that Intel® DPDK libraries are initialized in the main thread at application startup rather than subsequently in the forwarding threads. However, the Intel® 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`.

18.4 Interrupt Thread

The Intel® 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 Intel® DPDK processing threads. These function callbacks should avoid manipulating Intel® DPDK objects that are also managed by the normal Intel® 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.

§ §

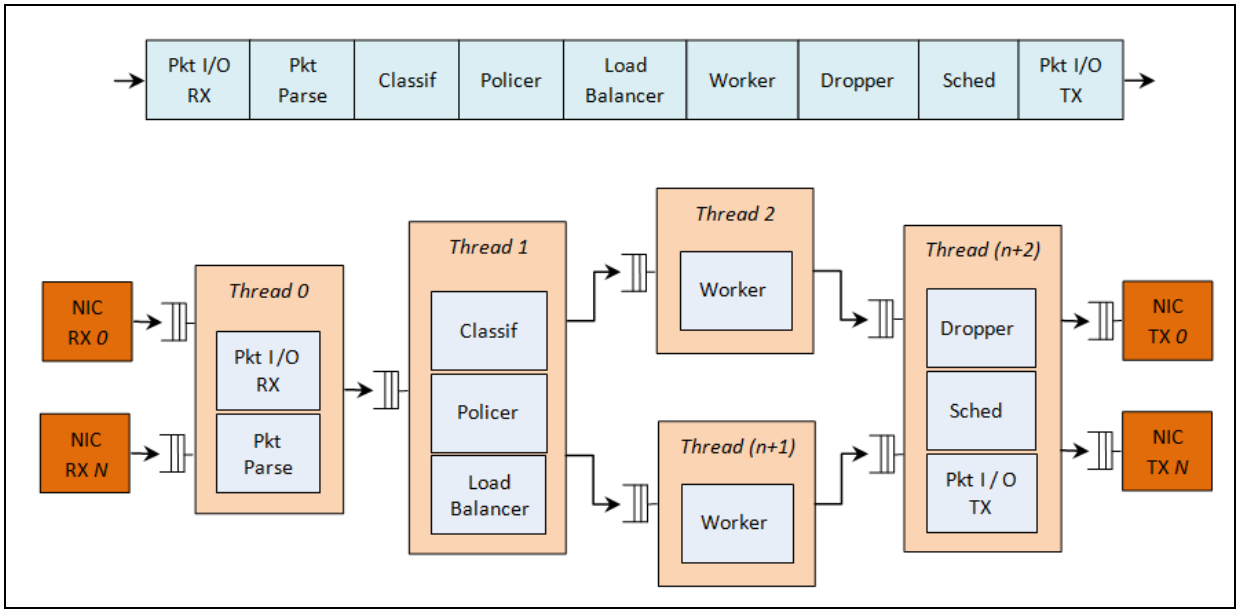
19.0 Quality of Service (QoS) Framework

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

19.1 Packet Pipeline with QoS Support

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

Figure 20. Complex Packet Processing Pipeline with QoS Support



This pipeline can be built using reusable Intel® 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 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.

Table 1. Packet Processing Pipeline Implementing QoS (Continued)

#	Block	Functional Description
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 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.

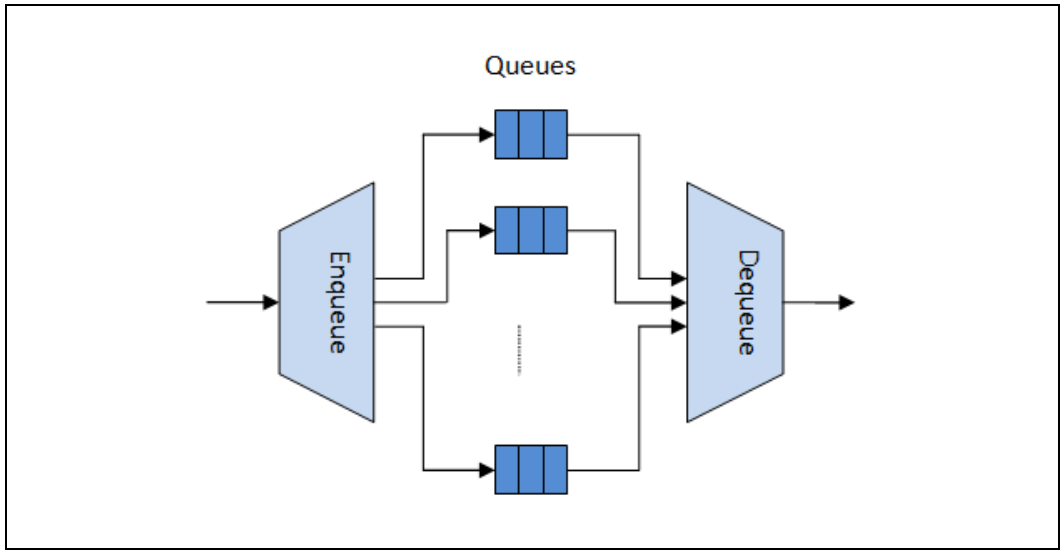
19.2 Hierarchical Scheduler

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.

19.2.1 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).

Figure 21. Hierarchical Scheduler Block Internal Diagram



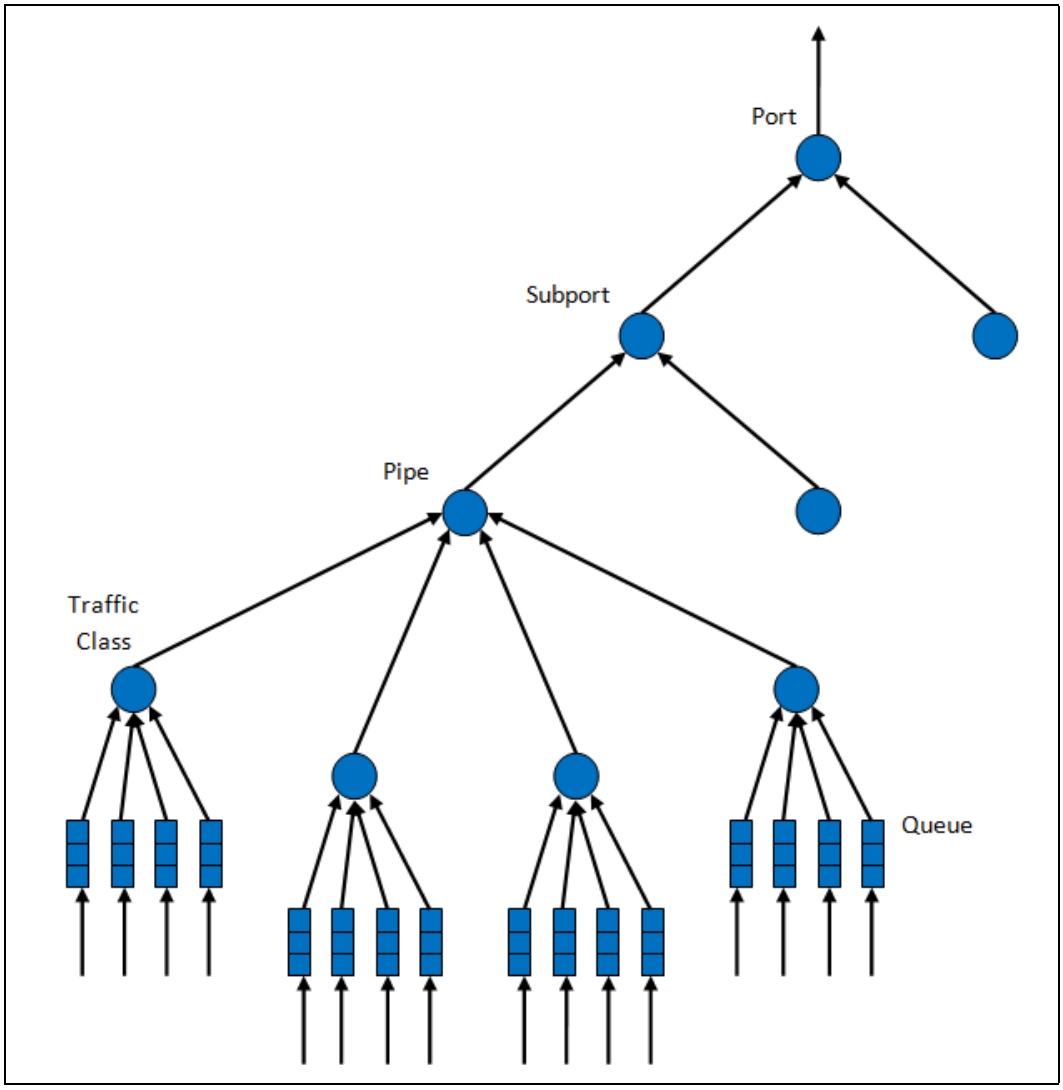
The hierarchical scheduler is optimized for a large number of packet queues. When only a small number of queues are needed, message passing queues should be used instead of this block. See Section 19.2.5, “Worst Case Scenarios for Performance” on page 101 for a more detailed discussion.

19.2.2 Scheduling Hierarchy

The scheduling hierarchy is shown in Figure 22. 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.

Figure 22. Scheduling Hierarchy per Port



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

Table 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.

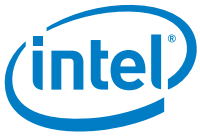


Table 3. Port Scheduling Hierarchy (Continued)

#	Level	Siblings per Parent	Functional Description
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 TC is oversubscribed (configuration time event), pipe TC upper limit is capped to a dynamically adjusted value that is shared by all the subport pipes.
5	Queue	4	1. Queues of the same TC are serviced using Weighted Round Robin (WRR) according to predefined weights.

19.2.3 Application Programming Interface (API)

19.2.3.1 Port Scheduler Configuration API

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

19.2.3.2 Port Scheduler Enqueue API

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

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

19.2.3.3 Port Scheduler Dequeue API

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

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

19.2.3.4 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);
```

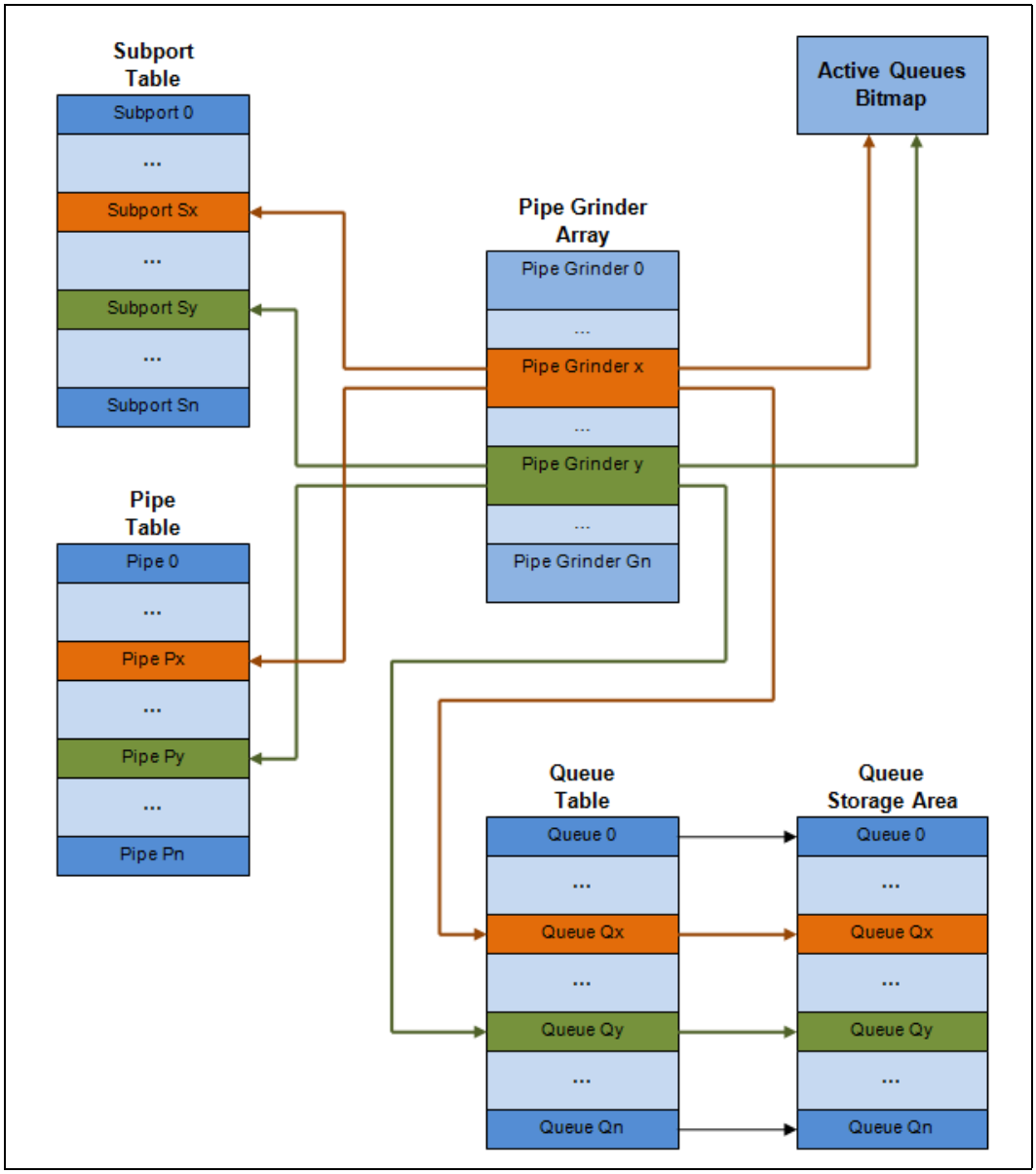
```
    /* Hierarchical scheduler dequeue */
    n_pkts_tx = rte_sched_port_dequeue(port, pkts_tx, N_PKTS_TX);

    /* Write packets to NIC TX queue */
    rte_eth_tx_burst(NIC_TX_PORT, NIC_TX_QUEUE, pkts_tx, n_pkts_tx);
}
```

19.2.4 Implementation

19.2.4.1 Internal Data Structures per Port

A schematic of the internal data structures in shown in Figure 23 with details in Table 4.

Figure 23. Internal Data Structures per Port



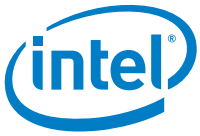


Table 4. Scheduler Internal Data Structures per Port

#	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 for any given pipe are stored in the same cache line.
4	Queue storage area	Config (default: 64 x8)	# queues per port	Wr	Rd	Array of elements per queue; each element is 8 byte in size (mbuf pointer).
5	Active queues bitmap	1 bit per queue	1	Wr (Set)	Rd, Wr (Clear)	The bitmap maintains one status bit per queue: queue not active (queue is empty) or queue active (queue is not empty). Queue bit is set by the scheduler enqueue and cleared by the scheduler dequeue when queue becomes empty. Bitmap scan operation returns the next non-empty pipe and its status (16-bit mask of active queue in the pipe).
6	Grinder	~128	Config (default: 8)	-	Rd, Wr	Short list of active pipes currently under processing. The grinder contains temporary data during pipe processing. Once the current pipe exhausts packets or credits, it is replaced with another active pipe from the bitmap.

19.2.4.2 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.

19.2.4.2.1 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.

19.2.4.2.2 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.

19.2.4.3 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).

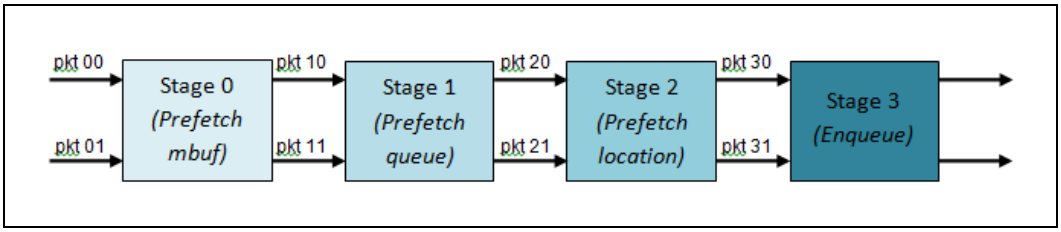
It should be noted the strong data dependency between these steps, as steps 2 and 3 cannot start before the result from steps 1 and 2 becomes available, which prevents the processor out of order execution engine to provide any significant performance optimizations.

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.

Figure 24 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.

Figure 24. 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).

19.2.4.4 Dequeue State Machine

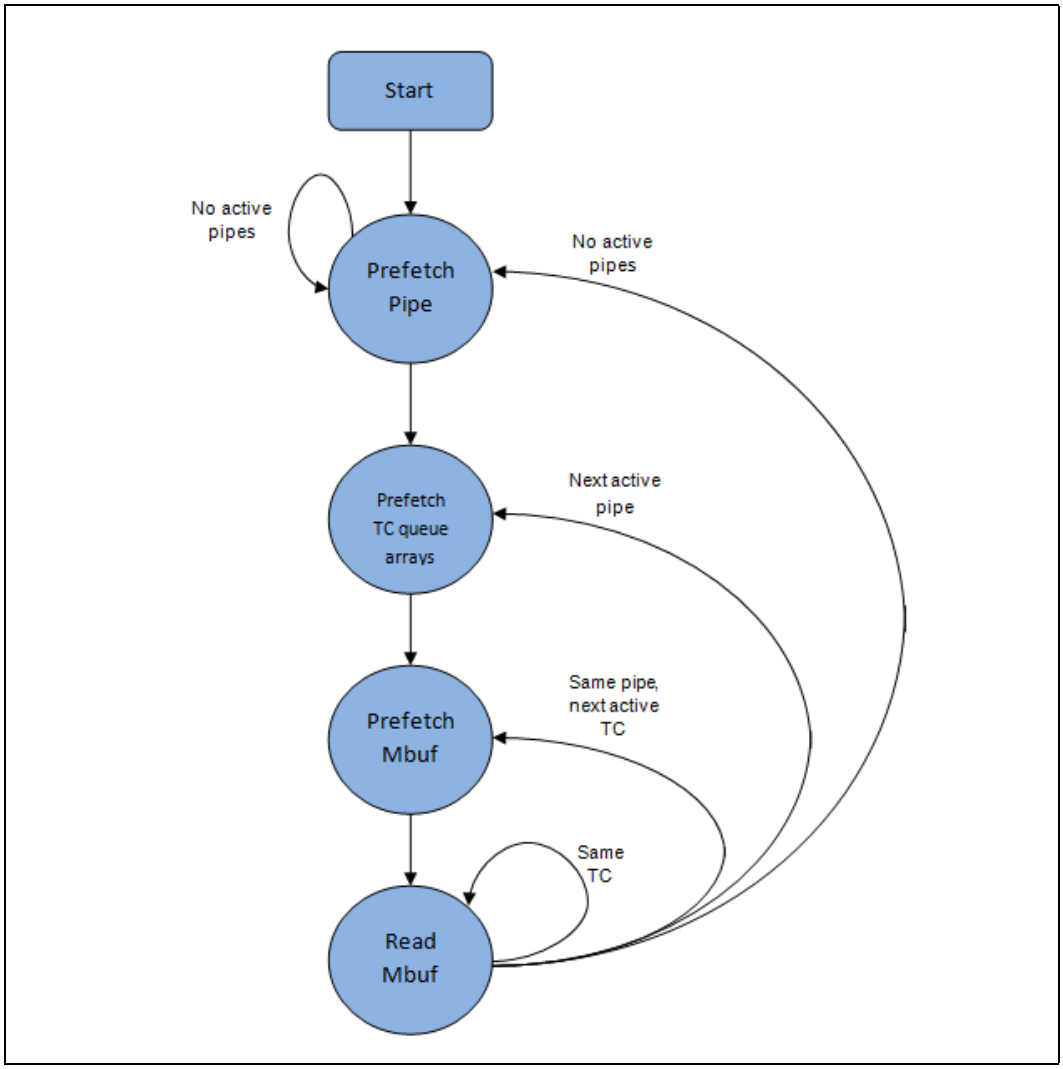
The sequence of steps to schedule the next packet from the current pipe is:

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.

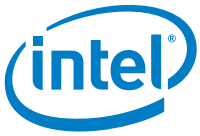
Figure 25. Pipe Prefetch State Machine for the Hierarchical Scheduler Dequeue Operation



19.2.4.5 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.



19.2.4.5.1 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.

19.2.4.5.2 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.

19.2.4.5.3 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.

19.2.4.6 Credit Logic

19.2.4.6.1 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.

19.2.4.6.2 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 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	

19.2.4.6.3 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 6 and Table 7.

Table 6. Token Bucket Generic Parameters

#	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.

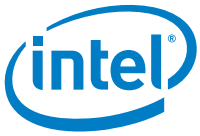


Table 7. Token Bucket Generic Operations

#	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 8, while the implementation of the token bucket operations is described in Table 9.

Table 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 19.2.4.5.1, "Internal Time Reference" on page 93 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 9. Token Bucket Operations

#	Token bucket operation	Description
1	Initialization	*tb_credits = 0;* or *tb_credits = tb_size / 2;*

Table 9. Token Bucket Operations

2	Credit update	Credit update options: 1. Every time a packet is sent for a port, update the credits of all the subports and pipes of that port. Not feasible. 2. Every time a packet is sent, update the credits for the pipe and subport. Very accurate, but not needed (a lot of calculations). 3. 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 19.2.4.4, "Dequeue State Machine" on page 91, 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. 1. Full accuracy can be achieved by selecting the value for *tb_period* for which *tb_credits_per_period = 1*. 2. 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;

3	Credit consumption (on packet scheduling)	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).
		Scheduling operations:
		pkt_credits = pkt_len + frame_overhead;
		if (tb_credits >= pkt_credits){tb_credits -= pkt_credits;}

19.2.4.6.4 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 10 and Table 11.

Table 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 19.2.4.5.1, "Internal Time Reference" on page 93 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*.

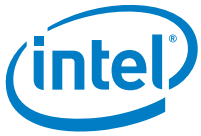


Table 10. Subport / Pipe Traffic Class Upper Limit Enforcement Persistent Data Structure

#	Subport or pipe field	Unit	Description
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 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;}

19.2.4.6.5 Weighted Round Robin (WRR)

The evolution of the WRR design solution from simple to complex is shown in Table 12.

Table 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.

Table 12. Weighted Round Robin (WRR)

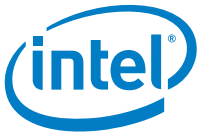
#	All Queues Active?	Equal Weights for All Queues?	All Packets Equal?	Strategy
3	Yes	No	No	**Packet level weighted round robin** This case can be reduced to the previous case by introducing a cost per byte that is different for each queue. Queues with lower weights have a higher cost per byte. This way, 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. Next queue: queue with smallest T. Before packet consumption from queue #i: *T(i) |= mask(i)* After packet consumption from queue #i: T(j) -= T(i), j ≠ i T(i) = pkt_len * t(i) Note: T(j) uses the T(i) value before T(i) is updated.

19.2.4.6.6 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

Table 13 summarizes some of the possible approaches for handling this problem, with the third approach selected for implementation.

Table 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. Table 14 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 15 illustrates the watermark operation.

Table 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 ++;
		}
		
		Pipe level:
		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 15. Watermark Calculation

No.	Subport Traffic Class Operation	Description
1	Initialization	**Subport level**: wm = WM_MAX

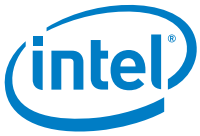


Table 15. Watermark Calculation (Continued)

No.	Subport Traffic Class Operation	Description

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;
		}

19.2.5 Worst Case Scenarios for Performance

19.2.5.1 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.

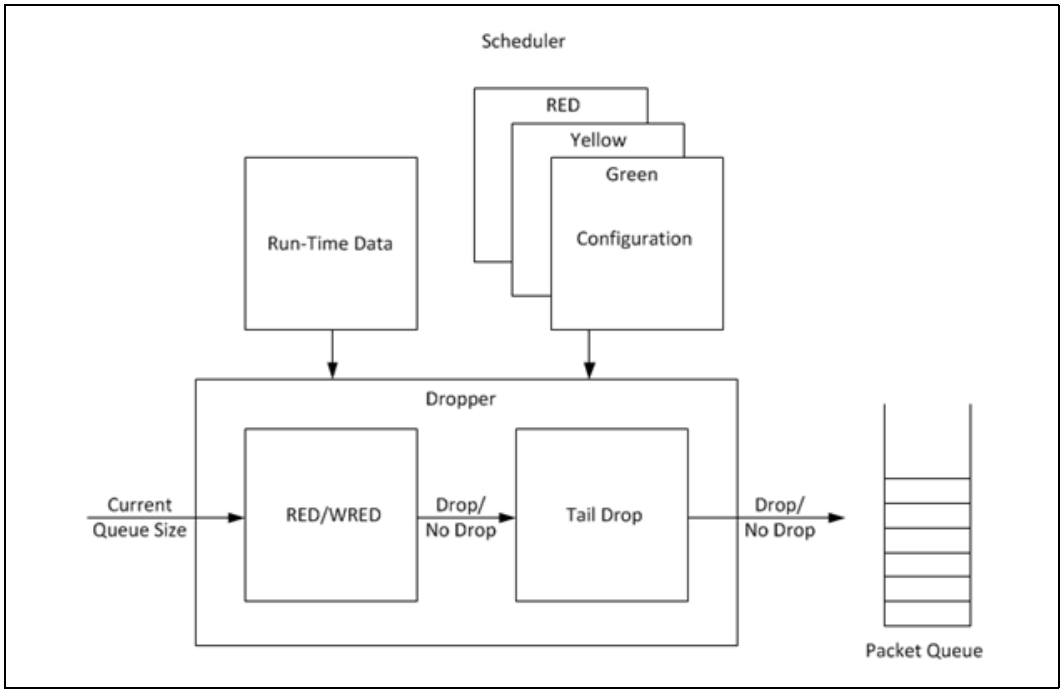
19.2.5.2 Single Queue with 100% Line Rate

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.

19.3 Dropper

The purpose of the Intel® 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. Figure 1 illustrates how the dropper integrates with the scheduler. The Intel® DPDK currently does not support congestion management so the dropper provides the only method for congestion avoidance.

Figure 26. High-level Block Diagram of the Intel® DPDK Dropper



The dropper uses the Random Early Detection (RED) congestion avoidance algorithm as documented in the reference publication[1]. 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.

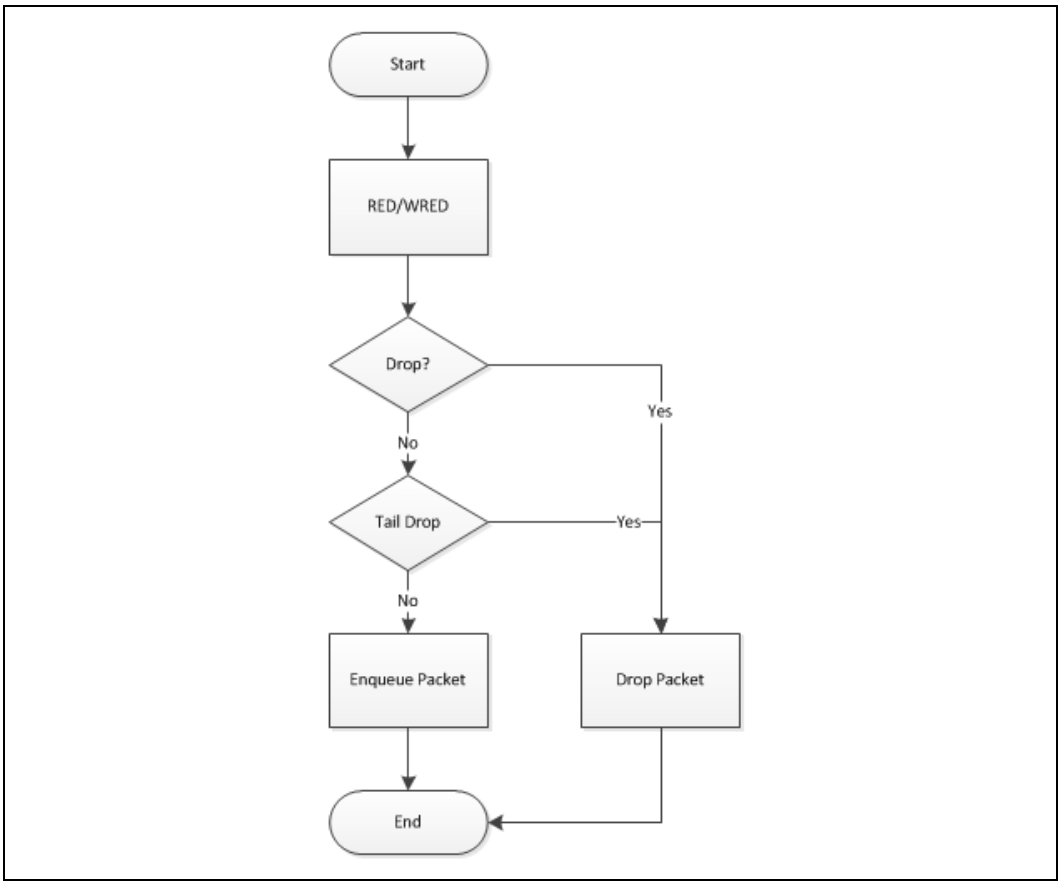
The dropper also supports Weighted Random Early Detection (WRED) by allowing the scheduler to select different RED configurations for the same packet queue at run-time.

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.

1. S. F. a. V. Jacobson, “Random Early Detection Gateways for Congestion Avoidance,” IEEE/ACM Transactionson Networking, pp. 1--22, 1993.

The flow through the dropper is illustrated in Figure 27. The RED/WRED algorithm is exercised first and tail drop second.

Figure 27. Flow Through the Dropper



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 Section 19.3.1, the enqueue operation is explained in Section 19.3.2 and the mark empty operation is explained in Section 19.3.3.

19.3.1 Configuration

A RED configuration contains the parameters given in Table 16.

Table 16. RED Configuration Parameters

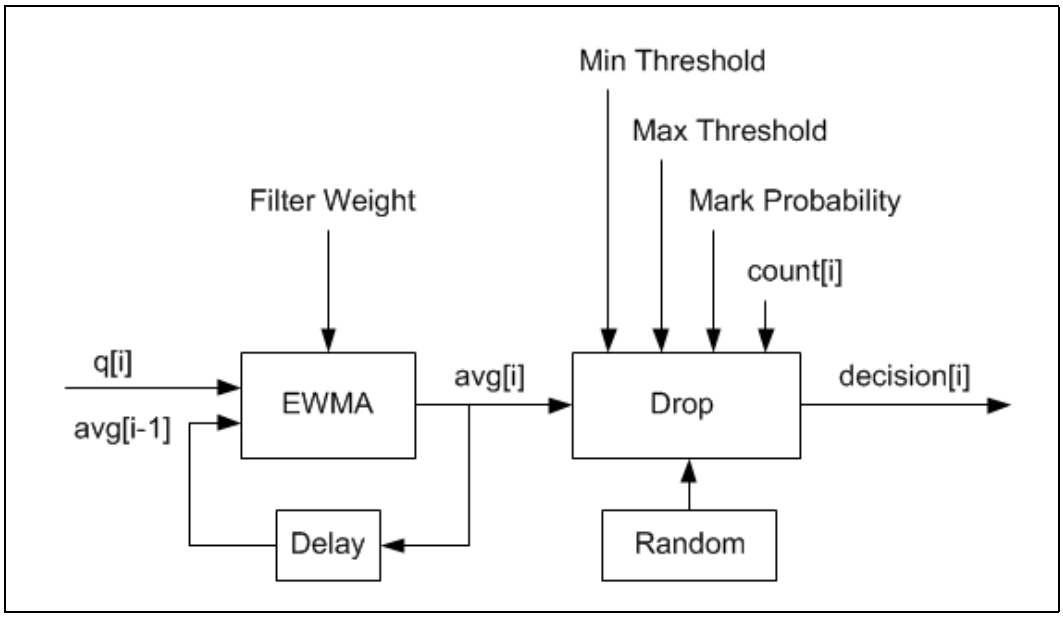
Parameter	Minimum	Maximum	Typical
Minimum Threshold	0	1022	1/4 x queue size
Maximum Threshold	1	1023	1/2 x queue size
Inverse Mark Probability	1	255	10
EWMA Filter Weight	1	12	9

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 $1/2^9$.

19.3.2 Enqueue Operation

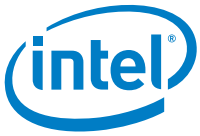
In the example shown in Figure 28, `q` (actual queue size) is the input value, `avg` (average queue size) and `count` (number of packets since the last drop) are run-time values, `decision` is the output value and the remaining values are configuration parameters.

Figure 28. Example Data Flow Through Dropper



19.3.2.1 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.

19.3.2.1.1 Average Queue Size Calculation when the Queue is not Empty

The definition of the EWMA filter is given in the following equation.

Equation 1.

$$avg[i] = (1 - w_q) \times avg[i-1] + w_q \times q[i]$$

Where:

- *avg* = average queue size
- w_q = filter weight
- *q* = actual queue size

Note: The filter weight, $w_q = 1/2^n$, where *n* is the filter weight parameter value passed to the dropper module on configuration (see Section 19.3.1).

19.3.2.1.2 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[1]:

Equation 2.

$$avg[i] = \text{avg}[i-1] \times (1 - \text{w}_\text{q})^m$$

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

1. S. F. a. V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," IEEE/ACM Transactions on Networking, pp. 1--22, 1993.

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 19.2.4.5.1, "Internal Time Reference" on page 93). 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.

19.3.2.1.3 Implementation

A numerical method is used to compute the factor $(1-w_q)^m$ that appears in Equation 2 on page 105.

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 $\log_2(1-w_q)$ for each value of w_q supported by the dropper module. The factor $(1-w_q)^m$ 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 $(1-w_q)^m$ is obtained using this method, the average queue size can be calculated from Equation 2 on page 105.

19.3.2.1.4 Alternative Approaches

Other methods for calculating the factor $(1-w_q)^m$ in the expression for computing average queue size when the queue is empty (Equation 2 on page 105) 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 19.3.2.1.3) 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 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.

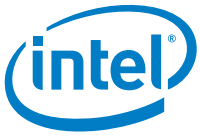


Table 17. Relative Performance of Alternative Approaches

Method	Relative Performance
Current dropper method (see Section 19.3.2.1.3)	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 condition, any relative performance value above 100% runs slower than the reference method.	

19.3.2.2 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.

19.3.2.2.1 Initial Packet Drop Probability

The initial drop probability is calculated using the following equation.

Equation 3.

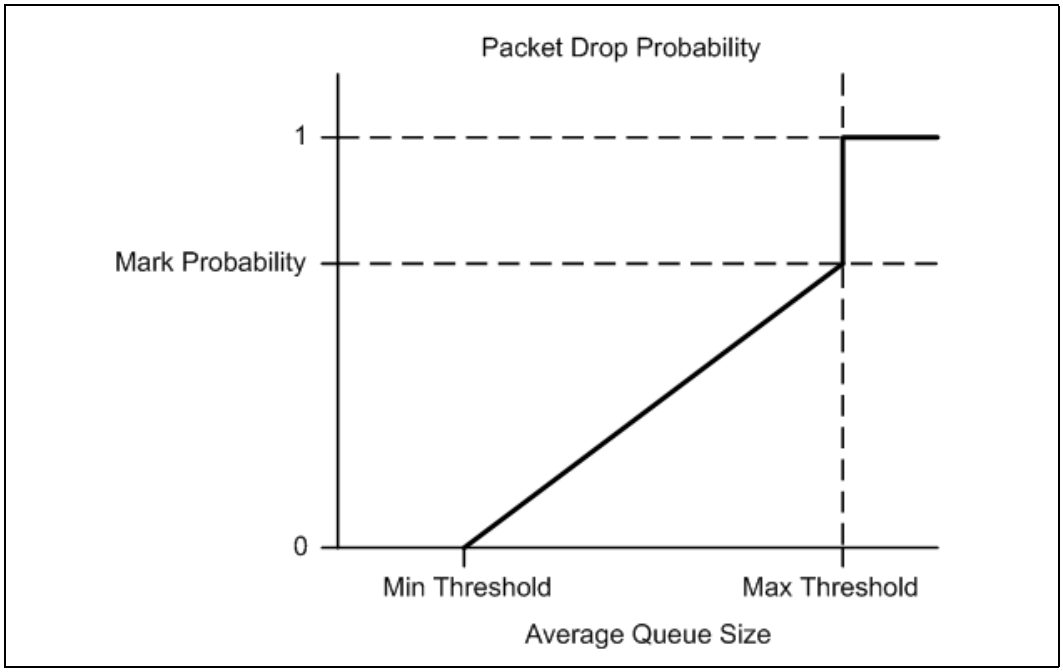
$$p_b = \begin{cases} 0, & avg < min_{th} \\ max_p\left(\frac{avg - min_{th}}{max_{th} - min_{th}}\right), & min_{th} \le avg < max_{th} \\ 1, & avg \ge max_{th} \end{cases}$$

Where:

- max_p = mark probability
- avg = average queue size
- min_{th} = minimum threshold
- max_{th} = maximum threshold

The calculation of the packet drop probability using Equation 3 is illustrated in Figure 29. 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.

Figure 29. Packet Drop Probability for a Given RED Configuration



19.3.2.2.2 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.

Equation 4.

$$p_a = \frac{p_b}{(2 - count \times p_b)}$$

Where:

- P_b = initial drop probability (from Equation 3 on page 107)
- *count* = number of packets that have arrived since the last drop

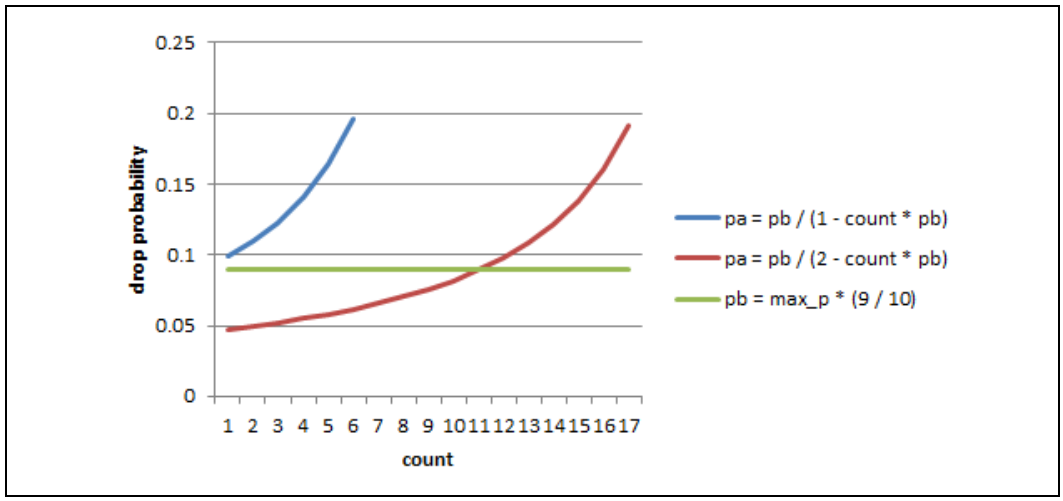
The constant 2, in Equation 4 is the only deviation from the drop probability formulae given in the reference document [1] where a value of 1 is used instead. It should be noted that the value p_a computed from Equation 4 can be negative or greater than 1. If this is the case, then a value of 1 should be used instead.

The initial and actual drop probabilities are shown in Figure 30. The actual drop probability is shown for the case where the formula given in the reference document[1] is used (blue curve) and also for the case where the formula implemented in the dropper module, Equation 4 is used (red curve). The formula in the reference document results in a significantly higher drop rate compared to the mark probability configuration

1. S. F. a. V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," IEEE/ACM Transactions on Networking, pp. 1--22, 1993.

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.

Figure 30. Initial Drop Probability (pb), Actual Drop probability (pa) Computed Using a Factor 1 (Blue Curve) and a Factor 2 (Red Curve)



19.3.3 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.

19.3.4 Source Files Location

The source files for the Intel® DPDK dropper are located at:

- `DPDK/lib/librte_sched/rte_red.h`
- `DPDK/lib/librte_sched/rte_red.c`

19.3.5 Integration with the Intel® DPDK QoS Scheduler

RED functionality in the Intel® DPDK QoS scheduler is disabled by default. To enable it, use the Intel® 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/defconfig_x86_64-default-linuxapp-gcc`. 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).

19.3.6 Integration with the Intel® DPDK QoS Scheduler Sample Application

The Intel® 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 Section 19.3.1. A sample RED configuration is shown below. In this example, the queue size is 64 packets.

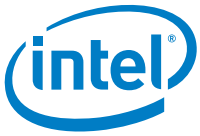
Note: For correct operation, the same EWMA filter weight parameter (wred weight) should be used for each packet color (green, yellow, red) in the same traffic class (tc).

```
; 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 18.

Table 18. RED Configuration Corresponding to RED Configuration File

RED Paramter	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



19.3.7 Application Programming Interface (API)

19.3.7.1 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,
                    const uint64_t time)
```

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 19.2.4.5.1, “Internal Time Reference” on page 93). The dropper reuses the scheduler time stamps for performance reasons.

19.3.7.2 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.

19.4 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.

19.4.1 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.

19.4.1.1 Color Blind and Color Aware Modes

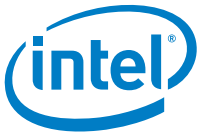
For both algorithms, the color blind mode is functionally equivalent to the color aware mode with input color set as green. For color aware mode, a packet with red input color can only get the red output color, while a packet with yellow input color can only get the yellow or red output colors.

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.

19.4.2 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.

§ §

20.0 Power Management

The Intel® 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.

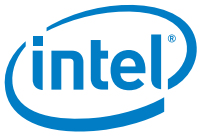
The interfaces for adjusting the operating CPU frequency are in the power management library. C-State control is implemented in applications according to the different use cases.

20.1 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 Intel® 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.



20.2 Core-load Throttling through C-States

Core state can be altered by speculative sleeps whenever the specified lcore has nothing to do. In the Intel® DPDK, if no packet is received after polling, speculative sleeps can be triggered according the strategies defined by the user space application.

20.3 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.

20.4 User Cases

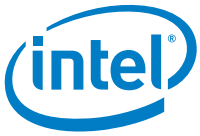
The power management mechanism is used to save power when performing L3 forwarding.

20.5 References

- `l3fwd-power`: The sample application in Intel® DPDK that performs L3 forwarding with power management.
- The "L3 Forwarding with Power Management Sample Application" chapter in the *Intel® DPDK IPL Sample Application's User Guide*.

§ §

Part 2: Development Environment



21.0 Source Organization

This section describes the organization of sources in the Intel® DPDK framework.

21.1 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 Intel® 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 Intel® DPDK libraries, including debug options. The user should look at the config file and become familiar with the options. The config file is also used to create a header file, which will be located in the new build directory.

21.2 Libraries

Libraries are located in subdirectories of `$(RTE_SDK)/lib`. By convention, we call a *library* any code that provides an API to an application. Typically, it generates an archive file (`.a`), but a kernel module should also go in the same directory.

The `lib` directory contains:

```
lib
+-- librte_cmdline       # command line interface helper
+-- librte_eal           # environment abstraction layer
+-- librte_ether         # generic interface to poll mode driver
+-- librte_hash          # hash library
+-- librte_kni           # kernel NIC interface
+-- librte_lpm           # longest prefix match library
+-- librte_malloc        # malloc-like functions
+-- librte_mbuf          # packet and control mbuf manipulation library
+-- librte_mempool       # memory pool manager (fixedsized objects)
+-- librte_meter         # QoS metering library
+-- librte_net           # various IP-related headers
+-- librte_pmd_e1000     # 1GbE poll mode drivers (igb and em)
+-- librte_pmd_ixgbe     # 10GbE poll mode driver
+-- librte_power         # power management library
+-- librte_ring          # software rings (act as lockless FIFOs)
+-- librte_sched         # QoS scheduler and dropper library
`-- librte_timer         # timer library
```

21.3 Applications

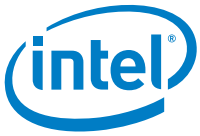
Applications are sources that contain a `main()` function. They are located in the `$(RTE_SDK)/app` and `$(RTE_SDK)/examples` directories.

The `app` directory contains sample applications that are used to test the Intel® DPDK (autotests). The `examples` directory contains sample applications that show how libraries can be used.

```
app
+-- chkincs               # test prog to check include depends
+-- test                  # autotests, to validate DPDK features
`-- test-pmd              # test and bench poll mode driver examples

examples
+-- cmdline               # Example of using cmdline library
+-- dpdk_qat              # Example showing integration with Intel QuickAssist
+-- exception_path        # Sending packets to and from Linux ethernet device (TAP)
+-- helloworld            # Helloworld basic example
+-- ipv4_reassembly       # Example showing IPv4 Reassembly
+-- ipv4_frag             # Example showing IPv4 Fragmentation
+-- ipv4_multicast        # Example showing IPv4 Multicast
+-- kni                   # Kernel NIC Interface example
+-- l2fwd                 # L2 Forwarding example 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 applications with multiple DPDK processes
+-- qos_meter             # QoS metering example
+-- qos_sched             # QoS scheduler and dropper example
+-- timer                 # Example of using librte_timer library
`-- vmdq_dcb              # Intel 82599 Ethernet Controller VMDQ and DCB receiving
`-- vmdq                  # Example of VMDQ receiving for both Intel 10G (82599)
                            and 1G (82576, 82580 and I350) Ethernet Controllers
```

Note: The actual `examples` directory may contain additional sample applications to those shown above. Check the latest Intel® DPDK source files for details.

§ §



22.0 Development Kit Build System

The Intel® DPDK requires a build system for compilation activities and so on. This section describes the constraints and the mechanisms used in the Intel® DPDK framework.

There are two use-cases for the framework:

- Compilation of the Intel® 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 Intel® DPDK

22.1 Building the Development Kit Binary

The following provides details on how to build the Intel® DPDK binary.

22.1.1 Build Directory Concept

After installation, a build directory structure is created. Each build directory contains include files, libraries, and applications:

```
~/DPDK$ ls
app                                 MAINTAINERS
config                              Makefile
COPYRIGHT                           mk
doc                                 scripts
examples                            lib
tools                               x86_64-default-linuxapp-gcc
x86_64-default-linuxapp-icc         i686-default-linuxapp-gcc
i686-default-linuxapp-icc
...
~/DEV/DPDK$ ls i686-default-linuxapp-gcc
app  build  hostapp  include  kmod  lib  Makefile

~/DEV/DPDK$ ls i686-default-linuxapp-gcc/app/
chkincs      dump_cfg      test      testpmd
chkincs.map  dump_cfg.map  test.map  testpmd.map

~/DEV/DPDK$ ls i686-default-linuxapp-gcc/lib/
libethdev.a       librte_hash.a    librte_mbuf.a     librte_pmd_ixgbe.a
librte_cmdline.a  librte_lpm.a     librte_mempool.a  librte_ring.a
librte_eal.a      librte_malloc.a  librte_pmd_e1000.a  librte_timer.a

~/DEV/DPDK$ ls i686-default-linuxapp-gcc/include/
arch                        rte_cpuflags.h    rte_memcpy.h
cmdline_cirbuf.h            rte_cycles.h      rte_memory.h
cmdline.h                   rte_debug.h       rte_mempool.h
cmdline_parse_etheraddr.h   rte_eal.h         rte_memzone.h
cmdline_parse.h             rte_errno.h       rte_pci_dev_ids.h
cmdline_parse_ipaddr.h      rte_ethdev.h      rte_pci.h
cmdline_parse_num.h         rte_ether.h       rte_per_lcore.h
cmdline_parse_portlist.h    rte_fbk_hash.h    rte_prefetch.h
cmdline_parse_string.h      rte_hash_crc.h    rte_random.h
```

```
cmdline_rdline.h           rte_hash.h        rte_ring.h
cmdline_socket.h           rte_interrupts.h  rte_rwlock.h
cmdline_vt100.h            rte_ip.h          rte_sctp.h
exec-env                   rte_jhash.h       rte_spinlock.h
rte_alarm.h                rte_launch.h      rte_string_fns.h
rte_atomic.h               rte_lcore.h       rte_tailq.h
rte_branch_prediction.h    rte_log.h         rte_tcp.h
rte_byteorder.h            rte_lpm.h         rte_timer.h
rte_common.h               rte_malloc.h      rte_udp.h
rte_config.h               rte_mbuf.h
```

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-default-linuxapp-gcc O=my_sdk_build_dir
```

This creates a new `my_sdk_build_dir` directory. After that, we can compile by doing:

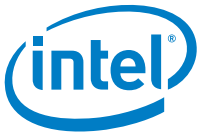
```
cd my_sdk_build_dir
make
```

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 Intel® DPDK.

22.2 Building External Applications

Since Intel® 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-default-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/DPDK.vars.mk, and ${RTE_SDK}/mk/DPDK.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.

22.3 Makefile Description

22.3.1 General Rules For Intel® DPDK Makefiles

In the Intel® DPDK, Makefiles always follow the same scheme:

1. Include $(RTE_SDK)/mk/DPDK.vars.mk at the beginning.
2. Define specific variables for RTE build system.
3. Include a specific $(RTE_SDK)/mk/DPDK.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/DPDK.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/DPDK.extapp.mk
```

22.3.2 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.

22.3.2.1 Application

These Makefiles generate a binary application.

- `rte.app.mk`: Application in the development kit framework
- `rte.extapp.mk`: External application
- `rte.hostapp.mk`: Host application in the development kit framework

22.3.2.2 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

22.3.2.3 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.

22.3.2.4 Kernel Module

- `rte.module.mk`: Build a kernel module in the development kit framework.

22.3.2.5 Objects

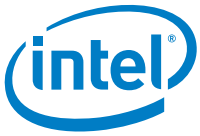
- `rte.obj.mk`: Object aggregation (merge several `.o` in one) in the development kit framework.
- `rte.extobj.mk`: Object aggregation (merge several `.o` in one) outside the development kit framework.

22.3.2.6 Misc

- `rte.doc.mk`: Documentation in the development kit framework
- `rte.gnuconfigure.mk`: Build an application that is configure-based (used to compile *newlib*).
- `rte.subdir.mk`: Build several directories in the development kit framework.

22.3.3 Useful Variables Provided by the Build System

- `RTE_SDK`: The absolute path to the Intel® 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 overriden 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.

22.3.4 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 Intel® 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.

22.3.5 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:

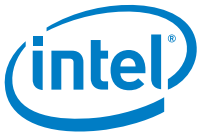
  ```
  CFLAGS += $(WERROR_CFLAGS)
  ```

  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.

22.3.6 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.
- NO_AUTOLIBS: If set, the libraries provided by the framework will not be included in the LDLIBS variable automatically.
- 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_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.

§ §



23.0 Development Kit Root Makefile Help

The Intel® DPDK provides a root level Makefile with targets for configuration, building, cleaning, testing, installation and others. These targets are explained in the following sections.

23.1 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-default-linuxapp-gcc`

23.2 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 Intel® 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`

23.3 Install Targets

- `install`
  Build the Intel® DPDK binary. Actually, this builds each supported target in a separate directory. The name of each directory is the name of the target.

The name of the targets to install can be optionally specified using `T=mytarget`. The target name can contain wildcard `*` characters. The list of available targets are in `$(RTE_SDK)/config` (remove the `defconfig_` prefix).
Example: `make install T=x86_64-*`

- `uninstall`
  Remove installed target directories.

23.4 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`
- `testall`
  Launch automatic tests for all installed target directories (after a `make install`).
  The name of the targets to test can be optionally specified using `T=mytarget`. The target name can contain wildcard (`*`) characters. The list of available targets are in `$(RTE_SDK)/config` (remove the `defconfig_` prefix).
  Examples: `make testall`, `make testall T=x86_64-*`

23.5 Documentation Targets

- `doxydoc`
  Generate the Doxygen documentation (pdf only).

23.6 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`

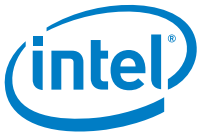
23.7 Misc Targets

- `help`
  Show this help.

23.8 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_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.

23.9 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-default-linuxapp-gcc
make O=mybuild
```

is equivalent to:

```
cd $(RTE_SDK)
make config O=mybuild T=x86_64-default-linuxapp-gcc
cd mybuild
# no need to specify O= now
make
```

23.10 Compiling for Debug

To compile the Intel® 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'
```

The Intel® DPDK and any user or sample applications can then be compiled in the usual way. For example:

```
make install T=x86_64-default-linuxapp-gcc
make -C examples/<theapp>
```

§ §

24.0 Extending the Intel® DPDK

This chapter describes how a developer can extend the Intel® DPDK to provide a new library, a new target, or support a new target.

24.1 Example: Adding a New Library libfoo

To add a new library to the Intel® DPDK, proceed as follows:

1. Add a new configuration option:

```
for f in config/*; do \
   echo CONFIG_RTE_LIBFOO=y >> $f; done
```

2. 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
```

3. Add a `foo()` function in `libfoo`.
   Definition is in `foo.c`:

```
void foo(void)
{
}
```

   Declaration is in `foo.h`:

```
extern void foo(void);
```

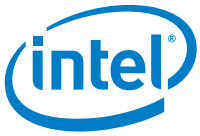
4. Update `lib/Makefile`:

```
vi ${RTE_SDK}/lib/Makefile
# add:
#  DIRS-$(CONFIG_RTE_LIBFOO) += libfoo
```

5. 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
```

6. Update `mk/DPDK.app.mk`, and add `-lfoo` in `LDLIBS` variable when the option is enabled. This will automatically add this flag when linking an Intel® DPDK application.



7. Build the Intel® DPDK with the new library (we only show a specific target here):

```
cd ${RTE_SDK}
make config T=x86_64-default-linuxapp-gcc
make
```

8. Check that the library is installed:

```
ls build/lib
ls build/include
```

24.1.1 Example: Using libfoo in the Test Application

The test application is used to validate all functionality of the Intel® 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 Intel® DPDK with the updated test application (we only show a specific target here):

```
cd ${RTE_SDK}
make config T=x86_64-default-linuxapp-gcc
make
```

§ §

25.0 Building Your Own Application

25.1 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-default-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
```

25.2 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-default-linuxapp-gcc
~/my_rte_app$ make
  CC main.o
  LD helloworld
  INSTALL-APP helloworld
  INSTALL-MAP helloworld.map
```

25.3 Customizing Makefiles

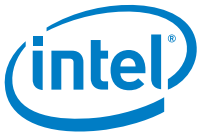
25.3.1 Application Makefile

The default makefile provided with the Hello World sample application is a good starting point. It includes:

- `$(RTE_SDK)/mk/DPDK.vars.mk` at the beginning
- `$(RTE_SDK)/mk/DPDK.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`).

25.3.2 Library Makefile

It is also possible to build a library in the same way:

- Include `$(RTE_SDK)/mk/DPDK.vars.mk` at the beginning.
- Include `$(RTE_SDK)/mk/DPDK.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`.

25.3.3 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`)
- `NO_AUTOLIBS`: If set, the libraries provided by the framework will not be included in the `LDLIBS` variable automatically.

§ §

26.0 External Application/Library Makefile help

External applications or libraries should include specific Makefiles from `RTE_SDK`, located in `mk` directory. These Makefiles are:

- `${RTE_SDK}/mk/DPDK.extapp.mk`: Build an application
- `${RTE_SDK}/mk/DPDK.extlib.mk`: Build a static library
- `${RTE_SDK}/mk/DPDK.extobj.mk`: Build objects (`.o`)

26.1 Prerequisites

The following variables must be defined:

- `${RTE_SDK}`: Points to the root directory of the Intel® DPDK.
- `${RTE_TARGET}`: Reference the target to be used for compilation (for example, `x86_64-default-linuxapp-gcc`).

26.2 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`

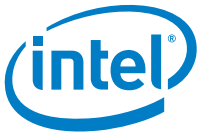
26.3 Help Targets

- `help`
  Show this help.

26.4 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.

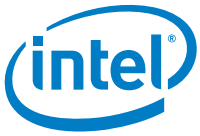
26.5 Make from Another Directory

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-default-linuxapp-icc
make -f /path/to/my_app/Makefile S=/path/to/my_app O=/path/to/build_dir
```

§ §

Part 3: Performance Optimization



27.0 Performance Optimization Guidelines

27.1 Introduction

The following sections describe optimizations used in the Intel® 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 Intel® DPDK.

And finally, they give an introduction to application profiling using a Performance Analyzer from Intel to optimize the software.

§ §

28.0 Writing Efficient Code

This chapter provides some tips for developing efficient code using the Intel® 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.

28.1 Memory

This section describes some key memory considerations when developing applications in the Intel® DPDK environment.

28.1.1 Memory Copy: Do not Use libc in the Data Plane

Many `libc` functions are available in the Intel® 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 Intel® DPDK API provides an optimized `rte_memcpy()` function.

28.1.2 Memory Allocation

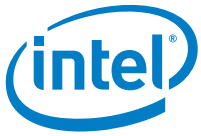
Other functions of `libc`, such as `malloc()`, provide a flexible way to allocate and free memory. In some cases, using dynamic allocation is necessary, but it is really not advised to use malloc-like functions in the data plane because managing a fragmented heap can be costly and the allocator may not 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.

28.1.3 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.

28.1.4 NUMA

On a NUMA system, it is preferable to access local memory since remote memory access is slower. In the Intel® 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.

28.1.5 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.

28.2 Communication Between lcores

To provide a message-based communication between lcores, it is advised to use the Intel® 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 Chapter 5.0, "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);
}
```

28.3 PMD Driver

The Intel® 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.

28.3.1 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 Intel® 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.

28.4 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`.

28.5 Coding Considerations

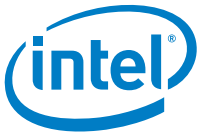
28.5.1 Inline Functions

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.

28.5.2 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();
```



28.6 Setting the Target CPU Type

The Intel® DPDK supports CPU microarchitecture-specific optimizations by means of CONFIG_RTE_MACHINE option in the Intel® DPDK configuration file. The degree of optimization depends on the compiler's ability to optimize for a specitic 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.

§ §

29.0 Profile Your Application

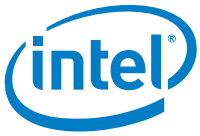
Intel processors provide performance counters to monitor events. Some tools provided by Intel can be used to profile and benchmark an application. See the *VTune™ Performance Analyzer Essentials* publication from Intel Press for more information.

For an Intel® 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

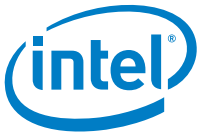
Refer to the *Intel Performance Analysis Guide* for details about application profiling.



30.0 Glossary

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 Intel® 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 Intel® 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
GbE	Gigabit Ethernet
HPET	High Precision Event Timer; a hardware timer that provides a precise time reference on x86 platforms.
I/O	Input/Output
lcore	A logical execution unit of the processor, sometimes called a *hardware thread*.

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)
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
Target	In the Intel® DPDK, the target is a combination of architecture, machine, executive environment and toolchain. For example: i686-default-linuxapp-gcc.
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