1 DPDK

This section will go into much more detail about the Data Plane Development Kit (DPDK), focussing on the basic concepts used by the fast packet processing framework for use within custom applications. The architecture is shown in figure 1 with each element described in the subsequent section. DPDK does have a number of more advanced features which can be exploited but aren't discussed in this report.

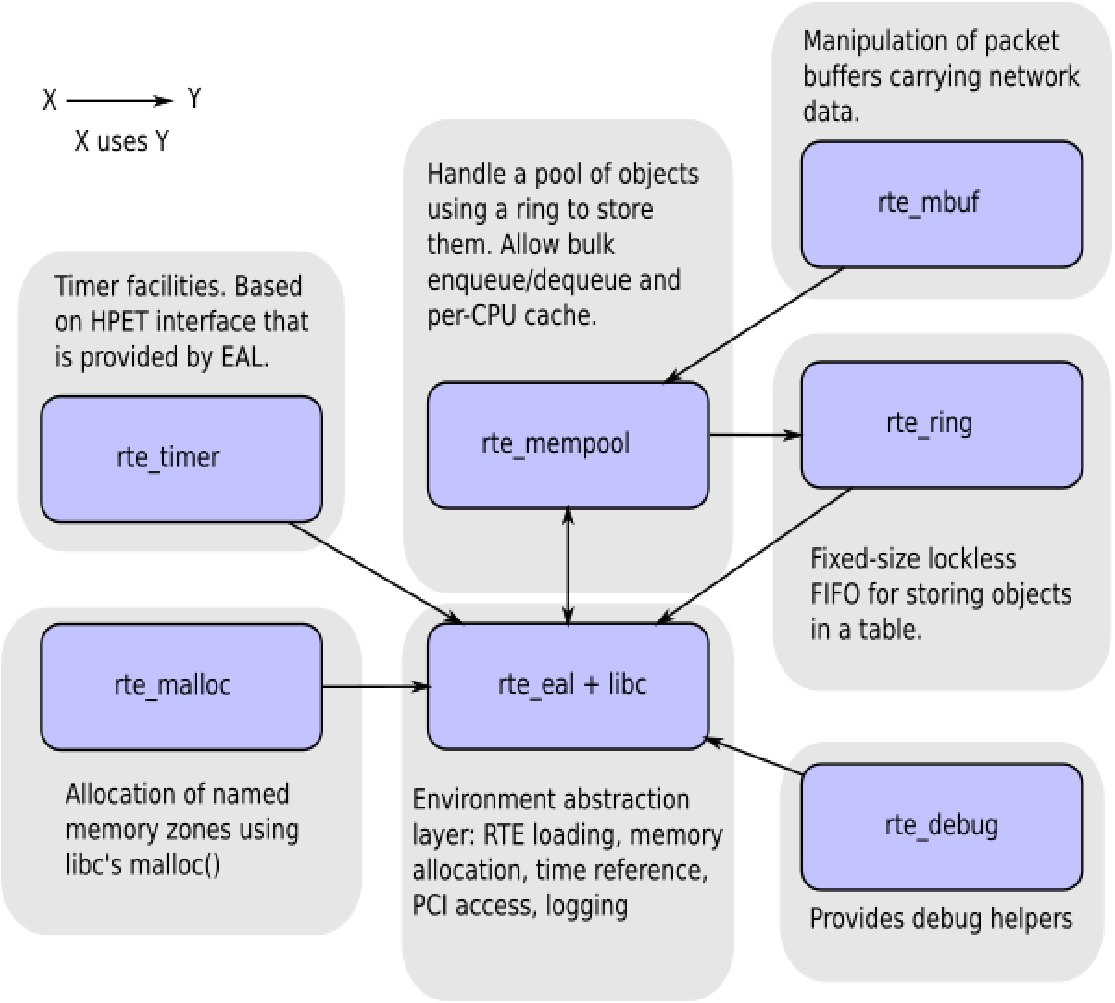
X → Y
X uses Y
Manipulation of packet buffers carrying network data.
rte_mbuf
Handle a pool of objects using a ring to store them. Allow bulk enqueue/dequeue and per-CPU cache.
Timer facilities. Based on HPET interface that is provided by EAL.
rte_timer
rte_mempool
rte_ring
Fixed-size lockless FIFO for storing objects in a table.
rte_malloc
rte_eal + libc
Allocation of named memory zones using libc's malloc()
Environment abstraction layer: RTE loading, memory allocation, time reference, PCI access, logging
rte_debug
Provides debug helpers

Figure 1: DPDK basic architecture overview

1.1 Environment Abstraction Layer (EAL)

The EAL abstracts the specific environment from the user and provides a constant interface in order for applications to be ported to other environments without problems. The EAL compilation may change depending on the architecture (32-bit or 64-bit), operating system, compilers in use or network interface controllers. This is done via the creation of a set of libraries that are environment specific and are responsible for the low-level operations gaining access to hardware and memory resources.

On the start-up of an application, the EAL is responsible for finding PCI information about devices and addresses via the igb_uio user space module, performing physical memory allocation using huge pages (section 1.3) and starting user-level threads for the logical cores (section 1.2) (lcores) of the application.

1.2 Logical Cores

Also know as lcores within DPDK, logical cores shouldn't be confused with processor cores. Lcores are threads which allow different applications functions to be run within different threads. This can allow different lcores to have access to different ports and queues while processing packets as well.

Within DPDK, lcores are implemented with POSIX [?] threads (on Linux) and make use of processor affinity (CPU pinning) [?]. This allows lcores to be only run on certain processing cores which reduces context switching and cache memory swapping and therefore increasing overall performance. However, this only works of the number of lcores is equal or less than the number of available processing cores so the number of lcores which are allowed to be initiated are limited. Generally a processor can only run 1 thread per core, but hyper-threaded CPUs can work on 2. Hyper-threading gives the core extra registers and execution units two allow the core to store the state of 2 threads and work on them simultaneously.

Machines with multiple sockets with multiple processing units add extra complication to lcores. As a core only exists on one socket, any memory associated with this socket (section 1.5.4) is only fastly accessible from that socket. This means that any lcore accessing a port must be on the same socket or else performance is greatly reduced. This needs to be manually assigned on the start-up of the application.

1.3 Huge Pages

Huge pages [?] [?] are a way of increasing performance when dealing with large amounts of memory. Normally, memory is managed in 4096 byte (4KB) blocks known as pages, which are listed within the CPU memory management unit (MMU). However, if a system uses a large amount of memory, increasing the number of standard pages is expensive on the CPU as the MMU can only handle thousands of pages and not millions.

The solution is to increase the page size from 4KB to 2MB or 1GB (if supported) which keeps the number of pages references small in the MMU but increases the overall memory for the system. Huge pages should be assigned at boot time for better overall management, but can be manual assigned on initial boot up if required.

DPDK makes uses of huge pages simply for increased performance due to the large amount of packet throughput in memory. This is even the case if the system memory size is relatively small and the application isn't processing an extreme number of packets.

1.4 Ring Buffer

A ring buffer (figure 2) is used by DPDK to manage transmit and receive queues for each network port on the system. This fixed sized first-in-first-out queue allows multiple objects to be enqueued and dequeued from the ring at the same time from any number of consumers or producers. A major disadvantage of this is that once the ring is full (more of a concern in receive queues), it allows no more objects can be added to the ring, resulting in dropped packets or packet caches. It is therefore imperative that applications can processes packets at the required rate. Head and tail pointers are used for each consumer and producer which indicates the next slot available in the buffer.

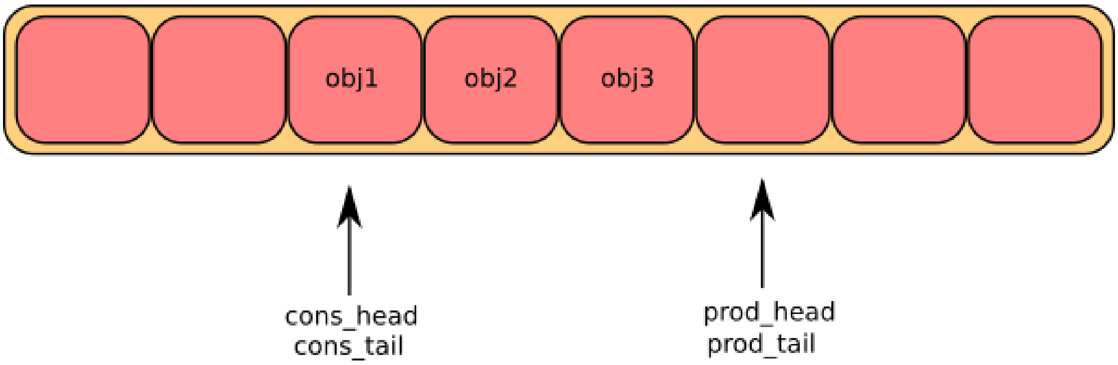

Figure 2: Ring Buffer

1.5 Memory Usage

DPDK tries to regulate memory usage in order to be more efficient and therefore handles its own memory management. However, it does allow application to allocate memory blocks for specific port queues and for data sharing between lcores. The major memory techniques used are described below.

1.5.1 Allocation

Since DPDK make use of huge pages, it provides its own memory allocation (malloc) library to allow memory blocks of any size, which also improves the portability of applications. However, even the DPDK malloc library is slow compared to pool memory access due to synchronisation constraints. Generally this library should only be used at initialisation time but does support NUMA for specific socket memory access and memory alignment.

1.5.2 Pools

Memory pools allow for fixed size allocation of memory which uses a ring to store fixed sized objects. These a almost guaranteed to be used for message buffer storage for receive and transmit queues for ports. They are initialised with a number of parameters to increase performance such as cache sizes and NUMA socket identification as well as a name identifier.

Pools offer increased performance over the standard memory allocation since the object padding is optimised so each object starts on a different memory channel and rank. Furthermore, a per core cache can be enabled at initialisation. This offers performance advantages, as without caching per core locks are required for every pool access. Caches offer cores lock free access to data, while bulk requested can be carried out on the pool to reduce locking.

1.5.3 Message Buffers

Message Buffers (mbufs) are stored within a specified memory pool and are used to carry data between different processes within the application and are primarily used for carrying network packets. Mbufs also contain meta-data about the information it is carrying which includes the data length, message type and offsets for the start of the data. The headroom shown in figure 3 shows empty bytes between the meta-data and start of the data which allows the data to be memory aligned for quicker access. Mbufs can also be chained together to allow for longer data, more commonly, jumbo packets.

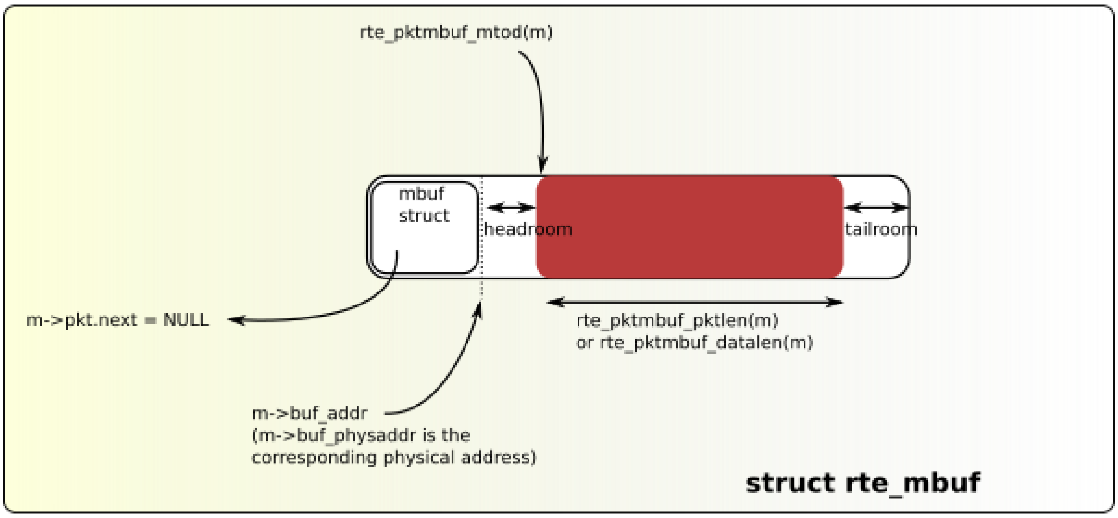

Figure 3: Message Buffer

1.5.4 NUMA

DPDK can make use of non-uniform memory access (NUMA) if the system supports it. NUMA (figure 4) is a method for speeding up memory access when multiple processors are trying to access the same memory and therefore reduces the processors waiting time. Each processor will receive its own bank of memory which is faster to access as it doesn't have to wait. As applications become more extensive, processors may need to share memory, which is possible via moving the data between memory banks. This somewhat negates the need for NUMA, but NUMA can be very effective depending on the application. DPDK can make extensive use of NUMA as each logical core is generally responsible for its own queues, and since queues can't be shared between logical cores (although dedicated ring buffers can), data sharing is rare.

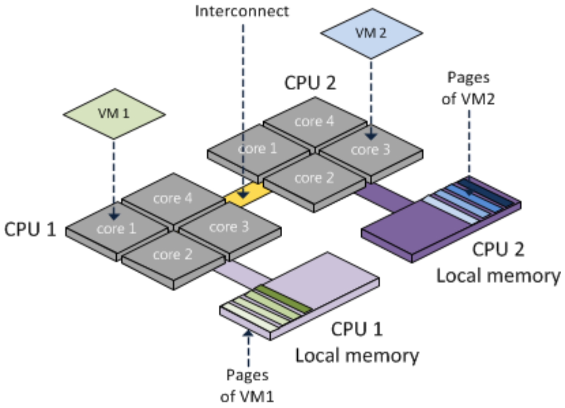

Figure 4: NUMA

1.6 Poll Mode Driver (PMD)

A Poll Mode Driver (PMD) is a user space device which allows configuration of network interface controllers and their associated queues. Consequently this means that the network stack typically associated with the port isn't used. To solve this either a custom network stack needs to be implemented or an open source stack like IwIP[1] can be layered onto of DPDK. PMD's work without interrupts which allow for quicker receiving, processing and transmitting of packets and are therefore lock free. Generally anything which can be achieved by interrupts can also be achieved via using rings and continuous polling of the rings. This means that 2 lcores running in parallel can't receive from the same queue on the same port. However, 2 parallel cores can receive from the same port on different queues.

There a number of consideration for application design depending on the hardware in use. Optimal performance can be achieved by carefully considering the hardware properties such as caches, bus speed and bandwidth along side the software design choices. For example, NICs are more efficient at transmitting multiple packets in a burst rather than individually but consequently overall throughput may be reduced.

1.7 Models

DPDK supports 2 methods for packet processing applications:

Pipe-line This is an asynchronous model where lcores a designated to perform certain tasks. Generally certain lcores will receive packets via the PMD API and simply pass those packets to other lcores via the use of a ring. These other lcores will then process the packets depending the requirements and then either forward the packets to the PMD for transmitting or pass them onto other lcore via a ring.

Run-to-completion This is a synchronous model where each lcore will retrieve the packets, process them individually and then output them for transmission. Each lcore should be assigned its own receive and transmit queue on a given port in order to negate the need for locks. This report will focus on the run-to-completion model.

[1] https://en.wikipedia.org/wiki/LwIP