# 37. 编写高效代码

本章提供了一些使用DPDK开发高效代码的技巧。 有关其他更详细的信息，请参阅  
Intel® 64 and IA-32 Architectures Optimization Reference Manual，这是编写高效代码的宝贵参考。

## 37.1. 内存

本节介绍在DPDK环境中开发应用程序时使用内存的一些关键注意事项。

### 37.1.1. 内存拷贝：不要在数据面程序中使用libc

通过Linux应用程序环境，DPDK中可以使用许多libc函数。 这可以简化应用程序的移植和控制平面的开发。 但是，这些功能中有许多不是为了性能而设计的。 诸如memcpy() 或 strcpy() 之类的函数不应该在数据平面中使用。 要复制小型结构体，首选方法是编译器可以优化一个更简单的技术。 请参阅英特尔新出版的 VTune™ Performance Analyzer Essentials 以获取建议。

对于经常调用的特定函数，提供一个自制的优化函数也是一个好主意，该函数应声明为静态内联。

DPDK API提供了一个优化的rte\_memcpy() 函数。

### 37.1.2. 内存申请

libc的其他功能，如malloc()，提供了一种灵活的方式来分配和释放内存。 在某些情况下，使用动态分配是必要的，但是建议不要在数据层面使用类似malloc的函数，因为管理碎片堆可能代价高昂，并且分配器可能无法针对并行分配进行优化。

如果您确实需要在数据平面中进行动态分配，最好使用固定大小对象的内存池。 这个API由librte\_mempool提供。 这个数据结构提供了一些提高性能的服务，比如对象的内存对齐，对对象的无锁访问，NUMA感知，批量get/put和percore缓存。 rte\_malloc() 函数对mempools使用类似的概念。

### 37.1.3. 内存区域的并发访问

几个lcore对同一个内存区域进行的读写（RW）访问操作可能会产生大量的数据高速缓存未命中，这代价非常昂贵。 通常可以使用per-lcore变量来解决这类问题。例如，在统计的情况下。 至少有两个解决方案：

* 使用 RTE\_PER\_LCORE 变量。注意，在这种情况下，处于lcore x的数据在lcore y上是无效的。
* 使用一个表结构（每个lcore一个）。在这种情况下，每个结构都必须缓存对齐。

如果在同一缓存行中没有RW变量，那么读取主要变量可以在不损失性能的情况下在内核之间共享。

### 37.1.4. NUMA

在NUMA系统上，由于远程内存访问速度较慢，所以最好访问本地内存。 在DPDK中，memzone，ring，rte\_malloc和mempool API提供了在特定内存槽上创建内存池的方法。

有时候，复制数据以优化速度可能是一个好主意。 对于经常访问的大多数读取变量，将它们保存在一个socket中应该不成问题，因为数据将存在于缓存中。

### 37.1.5. 跨存储器通道分配

现代内存控制器具有许多内存通道，可以支持并行数据读写操作。 根据内存控制器及其配置，通道数量和内存在通道中的分布方式会有所不同。 每个通道都有带宽限制，这意味着如果所有的存储器访问都在同一通道上完成，则存在潜在的性能瓶颈。

默认情况下， [Mempool Library](https://link.jianshu.com?t=mempool_lib.html" \l "mempool-library" \t "_blank) 分配对象在内存通道中的地址。

## 37.2. lcore之间的通信

为了在内核之间提供基于消息的通信，建议使用提供无锁环实现的DPDK ring API。  
该环支持批量访问和突发访问，这意味着只需要一次昂贵的原子操作即可从环中读取多个元素（请参阅 [Ring lib](https://link.jianshu.com?t=ring_lib.html" \t "_blank)）。  
使用批量访问操作时，性能会大大提高。  
出队消息的代码算法可能类似于以下内容：

#define MAX\_BULK 32

while (1) {

/\* Process as many elements as can be dequeued. \*/

count = rte\_ring\_dequeue\_burst(ring, obj\_table, MAX\_BULK, NULL);

if (unlikely(count == 0))

continue;

my\_process\_bulk(obj\_table, count);

}

## 37.3. PMD 驱动

DPDK轮询模式驱动程序（PMD）也能够在批量/突发模式下工作，允许在发送或接收功能中对每个呼叫的一些代码进行分解。

避免部分写入。 当PCI设备通过DMA写入系统存储器时，如果写入操作位于完全缓存行而不是部分写入操作，则其花费较少。 在PMD代码中，已采取了尽可能避免部分写入的措施。

### 37.3.1. 低报文延迟

传统上，吞吐量和延迟之间有一个折衷。 可以调整应用程序以实现高吞吐量，但平均数据包的端到端延迟通常会因此而增加。 类似地，可以将应用程序调整为平均具有低端到端延迟，但代价是较低的吞吐量。

为了实现更高的吞吐量，DPDK尝试通过突发处理数据包来合并单独处理每个数据包的成本。

以testpmd应用程序为例，突发大小可以在命令行上设置为16（也是默认值）。 这允许应用程序一次从PMD请求16个数据包。 然后，testpmd应用程序立即尝试传输所有接收到的数据包，在这种情况下是全部16个数据包。

在网络端口的相应的TX队列上更新尾指针之前，不发送分组。 当调整高吞吐量时，这种行为是可取的，因为对RX和TX队列的尾指针更新的成本可以分布在16个分组上， 有效地隐藏了写入PCIe 设备的相对较慢的MMIO成本。 但是，当调优为低延迟时，这不是很理想，因为接收到的第一个数据包也必须等待另外15个数据包才能被接收。 直到其他15个数据包也被处理完毕才能被发送，因为直到TX尾指针被更新，NIC才知道要发送数据包，直到所有的16个数据包都被处理完毕才被发送。

为了始终如一地实现低延迟，即使在系统负载较重的情况下，应用程序开发人员也应避免处理数据包。 testpmd应用程序可以从命令行配置使用突发值1。 这将允许一次处理单个数据包，提供较低的延迟，但是增加了较低吞吐量的成本。

## 37.4. 锁和原子操作

原子操作意味着在指令之前有一个锁定前缀，导致处理器的LOCK＃信号在执行下一条指令时被断言。 这对多核环境中的性能有很大的影响。

可以通过避免数据平面中的锁定机制来提高性能。 它通常可以被其他解决方案所取代，比如percore变量。 而且，一些锁定技术比其他锁定技术更有效率。 例如，Read-Copy-Update（RCU）算法可以经常替换简单的rwlock

## 37.5. 编码考虑

### 37.5.1. 内联函数

小函数可以在头文件中声明为静态内联。 这避免了调用指令的成本（和关联的上下文保存）。 但是，这种技术并不总是有效的。 它取决于许多因素，包括编译器。

### 37.5.2. 分支预测

英特尔的C/C ++编译器icc/gcc内置的帮助函数likely()和unlikely()允许开发人员指出是否可能采取代码分支。 例如：

if (likely(x > 1))

do\_stuff();

## 37.6. 设置目标CPU类型

DPDK通过DPDK配置文件中的CONFIG\_RTE\_MACHINE选项支持CPU微体系结构特定的优化。 优化程度取决于编译器针对特定微架构进行优化的能力，因此，只要有可能，最好使用最新的编译器版本。

如果编译器版本不支持特定的功能集（例如，英特尔®AVX指令集），则编译过程将优雅地降级到编译器支持的任何最新功能集。

由于构建和运行时目标可能不相同，因此生成的二进制文件还包含在main()函数之前运行的平台检查，并检查当前机器是否适合运行二进制文件。

除编译器优化之外，一组预处理器定义会自动添加到构建过程中（不管编译器版本如何）。 这些定义对应于目标CPU应该能够支持的指令集。 例如，为任何支持SSE4.2的处理器编译的二进制文件将定义RTE\_MACHINE\_CPUFLAG\_SSE4\_2，从而为不同的平台启用编译时代码路径选择。