<https://cloud.tencent.com/developer/article/1394706>

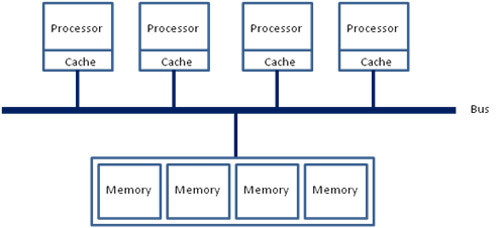
产生性能瓶颈有多方面的原因，包括硬件（自身能力限制或BIOS设置不当）、操作系统（某些feature没打开）和软件。软件方面的性能瓶颈主要是由于编码不当导致，常见原因有以下几种：

* 数据结构cache line未对齐，访问的数据跨cache line
* 不恰当的内存操作
* cache miss严重，经常跨socket访问数据
* 锁竞争
* 中断太多
* context切换频繁

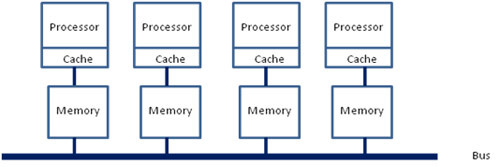
本文讲述了在编码时如何利用x86平台的特点（主要是内存方面）来避免性能瓶颈的技巧，并对性能优化给出一种思路。

**UMA与NUMA**

UMA(Uniform Memory Access)，即统一内存访问。在UMA内存架构中，所有处理器通过一条总线共享内存，如下图所示：



因为UMA结构中，所有处理器均通过同一条总线访问内存，故访问内存所花时间是一样的。且访问时间与数据在内存中的位置无关。而在NUMA（Non Uniform Memory Access，非一致性内存访问）结构中，访问内存所需时间与数据在内存中的位置有关，每个处理器都有自己的本地内存，访问本地内存速度很快，通过共享总线可以访问其他处理器的本地内存。结构如图所示：



正如前面所说，在NUMA结构中，访问内存所需时间与数据在内存中的位置有很大关系。处理器访问自己的本地内存要比访问其他处理器的本地内存要快得多。DPDK支持NUMA架构，接下来主要介绍一些进行内存操作方面需要注意的地方。

**减少内存拷贝**

出于性能考虑，要最小化数据的内存拷贝。在写代码的时候，当遇见需要拷贝数据时，考虑有没有一种更好的解决方式替代，如传递指针而非整个数据结构；在需要使用strcpy和memcpy时，用rte\_strcpy和rte\_memcpy作替。

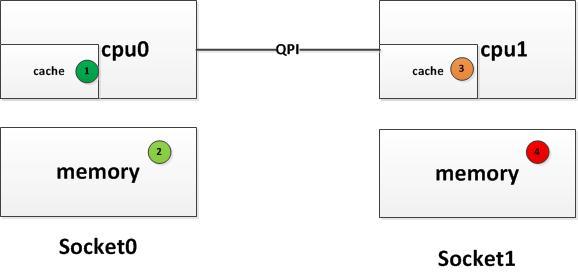
**合理分配内存**

在实时处理数据包转发的系统中，一般不建议在数据面进行动态内存分配，因为不停的申请和释放动态内存会使堆产生碎片，管理这样的堆开销很大。如果真的需要在程序中动态申请内存，要避免使用libc的malloc接口，使用DPDK提供的类malloc函数作为替代。DPDK主要提供三种内存模型：rte\_malloc、rte\_mempool、rte\_memzone，它们的使用场景如下：

* 需要使用malloc时，用rte\_malloc
* 需要高性能分配内存时，用rte\_mempool
* 当程序需要分配一大块内存，在程序的生命周期不释放，用rte\_memzone

关于内存申请，通常的做法是在程序初始化阶段分配好固定大小内存，通过指针链表串连起来管理它的分配与回收。例如，NAT中的分片处理，当后续分片先于首片到达设备时需要先缓存起来，每次从链表头取出一个结点来缓存报文，当缓存报文处理完后，又把该结点“回收”到链表头部。

**尽量访问本地内存**



根据前面的介绍，在NUMA系统中，访问本地内存比访问远端内存更佳高效。如上图所示，一个运行在core0上的应用访问数字标号处数据的速度快慢从高到低为1 > 2 > 3 > 4。在程序运行时，要避免进行过多的远端内存访问，DPDK提供在指定socket上分配memory的API。

如果内存充裕的话，可以考虑复制一份数据到另一个socket上来提升数据读取的速度。

**数据结构设计**

* 成员变量从大到小排列，避免过多的padding。考虑如下两个结构体：

struct s1

{

int a;

char b;

char c;

};

struct s2

{

char b;

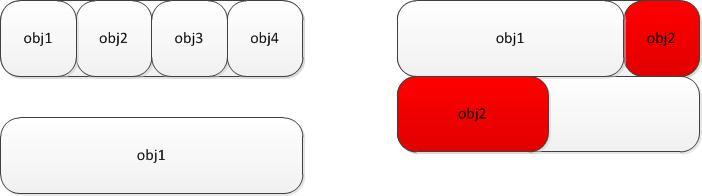
int a;

char c;

};

结构体s1的大小为8字节，结构体s2为12字节，在定义时不考虑padding的话，每个结构体变量会浪费4字节。

* 避免数据结构跨cache line，一条cache line应当正好放下整数个（N≥1）对象。



红色标识的obj2被load在两条cache line，如果访问这个对象时cpu需要更多的时钟，在数据结构时应该避免。可以在定义数据结构时用宏\_\_rte\_cache\_aligned或加入padding成员。

struct s3

{

uint32\_t x;

uint16\_t y;

}\_\_rte\_cache\_aligned;

struct s4

{

uint32\_t x;

uint16\_t y;

uint16\_t padding;

}

**数据预取**

一般访问CPU的cache效率最高，提前将需要处理的数据load到cache可以提高性能，但预取必须在合适的时间点发起，过早发起预取会导致数据还没有被使用就被替换出cache，最终适得其反，所以需要根据实际应用场景和多次尝试找到最合适的预取时间点。

通常在进行数据包处理时会先对数据包进行预取操作。DPDK提供的接口rte\_prefetch0会触发cpu进行预取操作，如下是预取数据包的示例代码：

/\* Prefetch first packets \*/

for (j = 0; j < PREFETCH\_OFFSET && j < nb\_rx; j++) {

rte\_prefetch0(rte\_pktmbuf\_mtod(pkts\_burst[j], void \*));

}

/\* Prefetch and forward already prefetched packets \*/

for (j = 0; j < (nb\_rx - PREFETCH\_OFFSET); j++) {

rte\_prefetch0(rte\_pktmbuf\_mtod(pkts\_burst[j + PREFETCH\_OFFSET], void \*));

l3fwd\_simple\_forward(pkts\_burst[j], portid);

}

/\* Forward remaining prefetched packets \*/

for (; j < nb\_rx; j++) {

l3fwd\_simple\_forward(pkts\_burst[j], portid);

}

**其他技巧**

* 对于一些代码行数不多且经常被调用的函数，定义为静态内联函数（static inline function），可以省去函数调用开销
* 分支预测，对于经常发生或不常发生的分支使用likely()/unlikely()来帮助编译器生成更加高效的可执行文件，减少cache miss

**性能瓶颈分析的一般方法**

上面提的一些技巧可以帮助在开发过程中规避部分性能陷阱，但仅仅做到这些是不够的，就像任何程序都有bug一样，性能瓶颈始终是存在的。通过阅读代码很难发现产生瓶颈的原因，这时候就需要借助一些测量工具来帮助定位原因了。

通常作性能瓶颈分析时需要找一个软件基准版本，给出一个metric值（通常由测试仪器给出，比如时延、吞吐量），然后再通过分析工具定位出产生性能缺陷的代码，反复修改这部分代码再给出一个metric值与之间的值作比较，直到metric值达到自己的预期为止。

Linux提供了很多开源工具来分析程序性能，比如iostat、perf、vmstat等。Intel也提供了一款专业的性能分析工具VTune帮助开发人员分析和定位程序性能瓶颈。Intel处理器内部有许多事件计数器，当有事件发生时对应的计数器加一，与程序性能相关的计数器有如下几种：

* cache misses
* 分支预测错误
* DTLB misses
* 长延时指令和异常

通过查看这些计数器值大小便可断定瓶颈原因，这是一种较底层的分析方法，需要对Intel CPU架构有所了解，且不能定位到产生瓶颈的具体代码行。VTune提供的另外一种分析方法Hotspots，能够帮助开发人员找出程序中消耗CPU最多的（热点）函数，通过这些列出的热点函数可以快速定位到代码行。通常使用Hotspots分析能够找出一般常见的性能瓶颈。

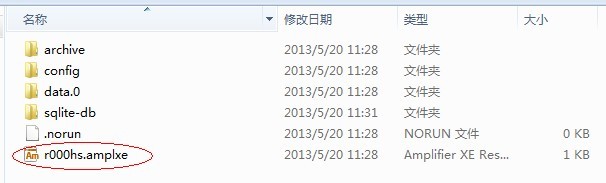
VTune提供Windows下的GUI和Linux下的CLI两种版本。我在项目中一般先用CLI版本的VTune采集运行程序机器的数据，然后将产生的结果移至windows下用GUI版本的VTune来分析，图形化的界面能够更利于定位分析。下面是利用VTune分析程序hotspots的demo：

1. 模拟测试环境，运行需要调优的程序；

2. 执行amplxe-cl，指定采集类型和目标程序，开始采集数据，运行结束后会在当前目录下生成类似r000hs名称的目录，里面存放的是收集的结果

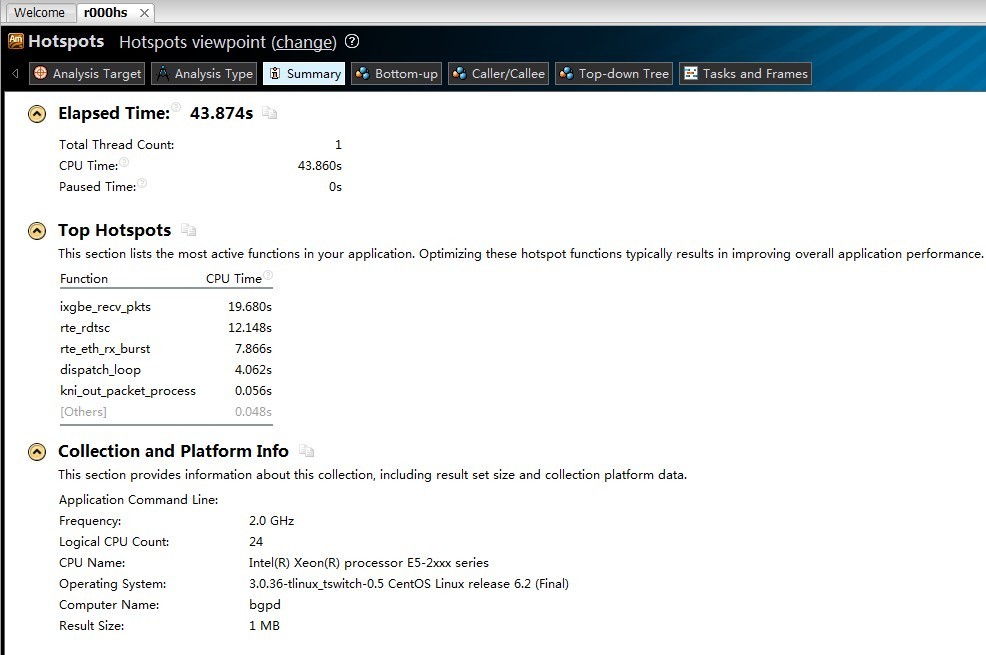
./amplxe-cl -collect hotspots -target-pid=29892

3. 将目录拷贝到windows下，用VTune打开文件r000hs.amplxe

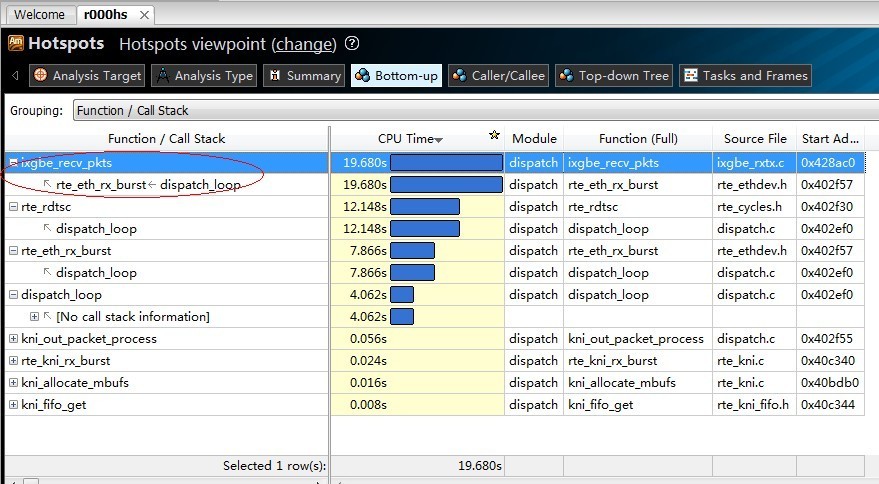


VTune打开后，出现的是一个关于hotspots的视图（因为之前指定收集的类型为hotspots，如果指定其他收集类型比如cpu events，则会出现对应视图）。里面有多个标签页记录了在采集过程中最耗CPU时间的函数。

Summary标签页记录了程序性能的大概数据，包括CPU耗时，top hotspots和系统信息。



Bottom-up标签页按函数消耗CPU时间从大到小排序，并可以查看函数的调用栈，如果目标程序没有采用编译优化，VTune甚至能定位到具体代码行，通过这些信息就可以很容易找到哪些代码最消耗CPU时间了。



在性能调优时，最好使用未经编译器优化的版本测试，这样VTune能够帮助定位到具体的代码行。采集时间不宜过长，否则会导致VTune分析缓慢，一般设置为60s即可，可通过amplxe-cl的参数指定，具体使用方法参考<http://software.intel.com/zh-cn/blogs/2010/11/10/amplxe-cl>。

**总结**

性能优化是个体力活又是一个细心活，需要反复的代码修改和测试数据才能找到性能瓶颈所在。熟悉底层开发环境和x86系统结构对性能优化有很大帮助，只有在了解开发环境之后才能写出跑得更快的代码，只有熟知CPU内部结构才能在优化时提供更多的线索，一款好的分析工具也是必不可少的。本文只是讲了性能优化技巧的冰山一角，更多的是要在实际项目中不断摸索积累经验，具体问题具体对待。

s