【鹅厂网事】高性能网关设备及服务实践

唐华新

2018-04-28 唐华新 阅 1353 转 11

一、引言

随着互联网的高速发展，内容量的提升以及对内容智能的需求、云产业的快速突起，作为互联网的计算基石服务器的形态以及使用成为了炙手可热的话题，全球各家大型互联网公司都持续的在服务器平台上有非常大的动作，譬如facebook的OCP等，而整个服务器的生态链也得到了促进和发展。随着服务器硬件性能的提升和网络硬件的开放，传统PC机的处理性能甚者可以和网络设备相媲美。另一方面SDN技术的发展，基础架构网络逐渐偏向基于通用计算平台或模块化计算平台的架构融合，来支持多样化的网络功能，传统的PC机器在分布式计算平台上的优势更为明显。在这些针对海量数据处理或海量用户的服务场景，高性能编程显得尤为重要。

本文讲述了从C10K到C10M过程中编程模式的改变；接着介绍了Intel DPDK开发套件如何突破操作系统限制，给开发高性能网络服务的程序员带来的福音；之后总结高性能程序设计的一些其它的优化方法；最后分享我们利用DPDK技术来实现的高性能网关设备和服务程序案例。

二、C10K到C10M

当前，解决C10K问题的服务器已经非常多。Nginx和Lighttpd两款非常优秀的基于事件驱动的web服务框架，Tornado和Django则是基于python开发的非阻塞的web框架；Yaws和Cowboy则是用Erlang开发轻量级web框架。这些软件使得C10K已经不再是问题了。

今天，C10M成为新的研究主题了。也许你会感到奇怪，千万级并发不是网络设备的性能吗？那是设备厂商该做的事情吧，答案在以前是，但如今不是。在互联网设备厂商相对封闭软件体系架构中，我们很少关注设备内部的配置。但是当你去拆开一款交换机之后就会发现，里面很可能就是我们PC机使用的x86芯片，即使不是x86，那也是标准的RISC处理器。也就是说现在的互联网硬件设备实际上很少是硬件组成——大部分都是软件来实现的。所以千万并发应该是我们软件开发人员应该去研究的问题！腾讯自研的Bobcat就是一个基于x86平台的自研设备，性能可达千万级别。

2.1异步模式的弊端

我们知道，在解决C10K问题的时候，需要将软件设计为异步模式，使用epoll来高效的处理网络读写事件。但在面对C10M问题中，这样设计是反而比较糟糕，为什么这么说呢？

一方面在基于多线程的服务器设计框架中，在没有请求到来的时候，线程将会休眠，当数据到来时，将由操作系统唤醒对应的线程，也就是说内核需要负责线程间频繁的上下文切换，我们是在依靠操作系统调度系统来服务网络包的调度。

另一方面在以Ngnix为代表的服务器场景，看上去仅使用一个线程监听Epoll事件来避免上下文切换，但我们仍将繁重的事件通知工作交由操作系统来处理。

最后要说的是，网卡驱动收包本身也是一个异步的过程，一般是当十几个或者更多的数据包到达之后通过软中断例程一次性将数据包递交到内核，而中断性能本身就不高。相比老的suse

在千万级并发场景下，我们的目标是要回到最原始的方式，使用轮询方式来完成一切操作，这样才能提升性能。

2.2数据包的可扩展性

Unix诞生之初就是为电话电报控制而设计的，它的控制平面和数据转发平面没有分离，不适合处理大规模网络数据包。如果能让应用程序直接接管网络数据包处理、内存管理以及CPU调度，那么性能可以得到一个质的提升。

为了达到这个目标，第一个要解决的问题就是绕过Linux内核协议栈，因为Linux内核协议栈性能并不是很优秀，如果让每一个数据包都经过Linux协议栈来处理，那将会非常的慢。像Wind River和6 Wind Gate等公司自研的内核协议栈宣称比Linux UDP/TCP协议栈性能至少提高500%以上，因此能不用Linux协议栈就不用。

不用协议栈的话当然就需要自己写驱动了，应用程序直接使用驱动的接口来收发报文。PF_RING，Netmap和intelDPDK等可以帮助你完成这些工作，并不需要我们自己去花费太多时间。

Intel官方测试文档给出了一个性能测试数据，在1S Sandbridge-EP 8*2.0GHz cores服务器上进行性能测试，不用内核协议栈在用户态下吞吐量可高达80Mpps（每个包处理消耗大约200 cpu clocks），相比之下，使用Linux内核协议栈性能连1Mpps都无法达到。

2.3多核的可扩展性

超级计算中心的天河二号使用了超过3w颗Xeon E5来提高性能。在程序设计过程中，即使在多核环境下也很快会碰到瓶颈，单纯的增加了处理器个数并不能线性提升程序性能，反而会使整体性能越来越低。一是因为编写代码的质量问题，没有充分利用多核的并行性，二是服务器软件和硬件本身的一些特性成为新的瓶颈，像总线竞争、存储体公用等诸多影响性能平行扩展的因素。

那么，我们怎样才能让程序能在多个CPU核心上平行扩展：尽量让每个核维护独立数据结构；使用原子操作来避免冲突；使用无锁数据结构避免线程间相互等待；设置CPU亲缘性，将操作系统和应用进程绑定到特定的内核上，避免CPU资源竞争；在NUMA架构下尽量避免远端内存访问。当然自己来实现无锁结构时要非常小心，避免出现ABA问题和不同CPU架构下的内存模型的差异。

2.4内存的可扩展性

内存的访问速度永远也赶不上cache和cpu的频率，为了能让性能平行扩展，最好是少访问。

从内存消耗来看，如果每个用户连接占用2K的内存，10M个用户将消耗20G内存，而操作系统的三级cache连20M都达不到，这么多并发连接的情况下必然导致cache失效，从而频繁的访问内存来获取数据。而一次内存访问大约需要300 cpuclocks，这期间CPU几乎被空闲。因此减少访存次数来避免cachemisses是我们设计的目标。

指针不要随意指向任意内存地址，因为这样每一次指针的间接访问可能会导致多次cache misses，最好将需要访问的数据放到一起，方便一次性加载到cache中使用。

按照4K页来计算，32G的数据需要占用64M的页表，使得页表甚至无法放到cache中，这样

2.5总结

C10M的思想就是将控制层留给Linux做，其它数据层全部由应用程序来处理。没有线程调度、没有系统调用、没有中断等，当你的程序仍运行在Linux用户空间，并仅仅对数据进行高效的分析和处理。

网卡：摒弃Linux内核协议栈，可以使用PF_RING，Netmap，intelDPDK来自己实现驱动；

CPU：使用多核编程技术替代多线程，将OS绑在指定核上运行；

内存：使用大页面，减少访问；

三、Intel® DPDK技术引入

Intel® DPDK全称Intel Data Plane Development Kit，是intel提供的数据平面开发工具集，为Intel architecture（IA）处理器架构下用户空间高效的数据包处理提供库函数和驱动的支持，它不同于Linux系统以通用性设计为目的，而是专注于网络应用中数据包的高性能处理。目前已经验证可以运行在大多数Linux操作系统上，包括FreeBSD 9.2、Fedora release18、Ubuntu 12.04 LTS、RedHat Enterprise Linux 6.3和Suse EnterpriseLinux 11 SP2等。DPDK使用了BSDLicense，极大的方便了企业在其基础上来实现自己的协议栈或者应用。

需要强调的是，DPDK应用程序是运行在用户空间上利用自身提供的数据平面库来收发数据包，绕过了Linux内核协议栈对数据包处理过程。Linux内核将DPDK应用程序看作是一个普通的用户态进程，包括它的编译、连接和加载方式和普通程序没有什么两样。DPDK程序启动后只能有一个主线程，然后创建一些子线程并绑定到指定CPU核心上运行。

DPDK核心组件由一系列库函数和驱动组成，为高性能数据包处理提供基础操作。内核态模块主要实现轮询模式的网卡驱动和接口，并提供PCI设备的初始化工作；用户态模块则提供大量给用户直接调用的函数。

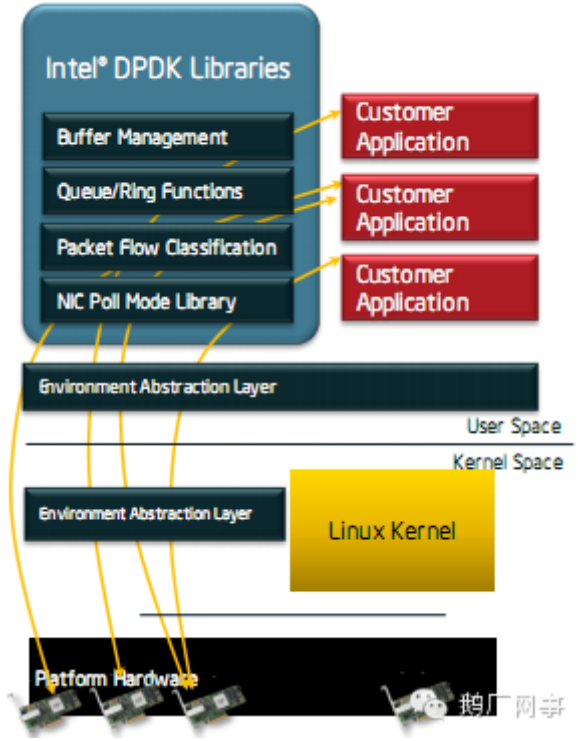


图3 DPDK架构图

EAL（Environment Abstraction Layer）即环境抽象层，为应用提供了一个通用接口，隐藏了与底层库与设备打交道的相关细节。EAL实现了DPDK运行的初始化工作，基于大页表的内存分配，多核亲缘性设置，原子和锁操作，并将PCI设备地址映射到用户空间，方便应用程序访问。

Buffer Manager API通过预先从EAL上分配固定大小的多个内存对象，避免了在运行过程中动态进行内存分配和回收来提高效率，常常用作数据包buffer来使用。

Queue Manager API以高效的方式实现了无锁的FIFO环形队列，适合与一个生产者多个消费者、一个消费者多个生产者模型来避免等待，并且支持批量无锁的操作。

Flow Classification API通过Intel SSE基于多元组实现了高效的hash算法，以便快速的将数据包进行分类处理。该API一般用于路由查找过程中的最长前缀匹配中，安全产品中根据Flow五元组来标记不同用户的场景也可以使用。

PMD则实现了Intel 1GbE、10GbE和40GbE网卡下基于轮询收发包的工作模式，大大加速网卡收发包性能。

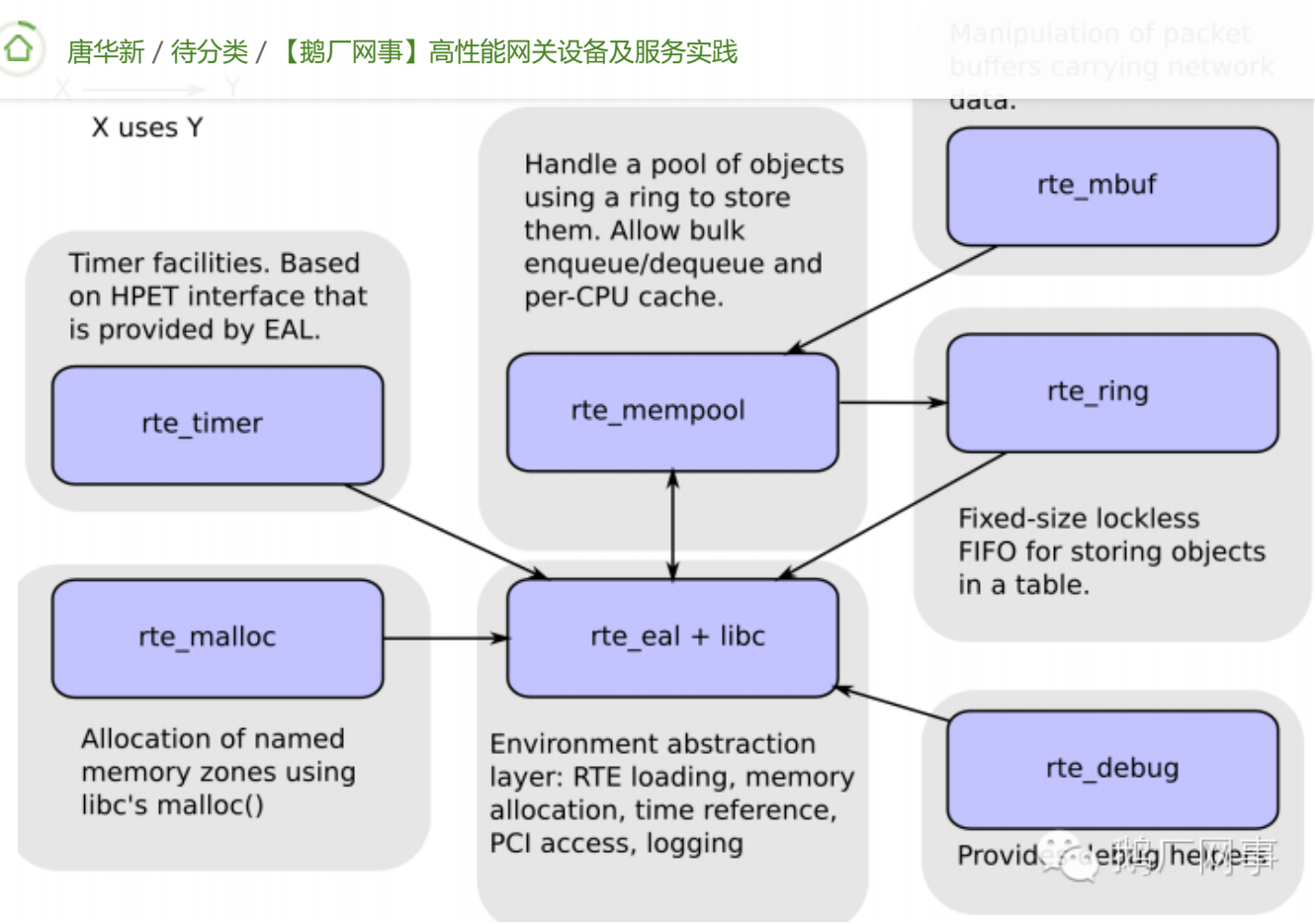


图4 DPDK的核心组件

图4展示了DPDK核心组件的依赖关系，详细介绍可以参考《Intel Data Plane Development kit：Software Architecture Specification》。

3.2DPDK的核心思想

3.2.1PMD

当前Linux操作系统都是通过中断方式通知CPU来收发数据包，我们假定网卡每收到10个据包触发一次软中断，一个CPU核心每秒最多处理2w次中断，那么当一个核每秒收到20w个包时

DPDK针对Intel网卡实现了基于轮询方式的PMD（Poll Mode Drivers）驱动，该驱动由API、用户空间运行的驱动程序构成，该驱动使用无中断方式直接操作网卡的接收和发送队列（除了链路状态通知仍必须采用中断方式以外）。目前PMD驱动支持Intel的大部分1G、10G和40G的网卡。

PMD驱动从网卡上接收到数据包后，会直接通过DMA方式传输到预分配的内存中，同时更新无锁环形队列中的数据包指针，不断轮询的应用程序很快就能感知收到数据包，并在预分配的内存地址上直接处理数据包，这个过程非常简洁。如果要是让Linux来处理收包过程，首先网卡通过中断方式通知协议栈对数据包进行处理，协议栈先会对数据包进行合法性进行必要的校验，然后判断数据包目标是否本机的socket，满足条件则会将数据包拷贝一份向上递交给用户socket来处理，不仅处理路径冗长，还需要从内核到应用层的一次拷贝过程。

3.2.2大页表

为实现物理地址到虚拟地址的转换，Linux一般通过查找TLB来进行快速映射，如果在查找TLB没有命中，就会触发一次缺页中断，将访问内存来重新刷新TLB页表。Linux下默认页大小为4K，当用户程序占用4M的内存时，就需要1K的页表项，如果使用2M的页面，那么只需要2条页表项，这样有两个好处：第一是使用hugepage的内存所需的页表项比较少，对于需要大量内存的进程来说节省了很多开销，像oracle之类的大型数据库优化都使用了大页面配置；第二是TLB冲突概率降低，TLB是cpu中单独的一块高速cache，一般只能容纳100条页表项，采用hugepage可以大大降低TLB miss的开销。

DPDK目前支持了2M和1G两种方式的hugepage。通过修改默认/etc/grub.conf中hugepage配置为“default_hugepagesz=1Ghugepagesz=1G hugepages=32 isolcpus=0-22”，然后通过mount –thugetlbfs nodev /mnt/huge就将hugepage文件系

3.2.3CPU亲缘性

多线程编程早已不是什么新鲜的事物了，多线程的初衷是提高整体应用程序的性能，但是如果不加注意，就会将多线程的创建和销毁开销，锁竞争，访存冲突，cache失效，上下文切换等诸多消耗性能的因素引入进来。这也是Ngnix使用单线程模型能获得比Apache多线程下性能更高的秘籍。

为了进一步提高性能，就必须仔细斟酌考虑线程在CPU不同核上的分布情况，这也就是常说的多核编程。多核编程和多线程有很大的不同：多线程是指每个CPU上可以运行多个线程，涉及到线程调度、锁机制以及上下文的切换；而多核则是每个CPU核一个线程，核心之间访问数据无需上锁。为了最大限度减少线程调度的资源消耗，需要将Linux绑定在特定的核上，释放其余核心来专供应用程序使用。

同时还需要考虑CPU特性和系统是否支持NUMA架构，如果支持的话，不同插槽上CPU的进程要避免访问远端内存，尽量访问本端内存。

3.3总结

总的来说，为了得到千万级并发，DPDK使用如下技术来达到目的：使用PMD替代中断模式；将每一个进程单独绑定到一个核心上，并让CPU从这些核上隔离开来；批量操作来减少内存和PCI设备的访问；使用预取和对齐方式来提供CPU执行效率；减少多核之间的数据共享并使用无锁队列；使用大页面。

除了UDP服务器程序，DPDK还有很多的场景能应用得上。一些需要处理海量数据包的应用场景都可以用上，包括但不局限于以下场景：NAT设备，负载均衡设备，IPS/IDS检测系统，

四、其它性能优化技术

除了DPDK提供的一些是思想外，我们的程序性能还能怎样进一步提高性能呢?

4.1减少内存访问

运算指令的执行速度是非常快，大多数在一个CPU cycle内就能完成，甚至通过流水线一个cycle能完成多条指令。但在实际执行过程中，处理器需要花费大量的时间去存储器来取指令和数据，在获取到数据之前，处理器基本处于空闲状态。那么为了提高性能，缩短服务器响应时间，我们可以怎样来减少访存操作呢?

（1）少用数组和指针，多用局部变量。因为简单的局部变量会放到寄存器中，而数组和指针都必须通过内存访问才能获取数据;

（2）少用全局变量。全局变量被多个模块或函数使用，不会放到寄存器中。

（3）一次多访问一些数据。就好比我们出去买东西一样，一次多带一些东西更省时间。我们可以使用多操作数的指令，来提高计算效率，DPDK最新版本配合向量指令集（AVX）可以使CPU处理数据包性能提升10%以上。

（4）自己管理内存分配。频繁调用malloc和free函数是导致性能降低的重要原因，不仅仅是函数调用本身非常耗时，而且会导致大量内存碎片。由于空间比较分散，也进一步增大了cache misses的概率。

（5）进程间传递指针而非整个数据块。在高速处理数据包过程中特别需要注意，前端线程和后端线程尽量在同一个内存地址来操作数据包，而不应该进行多余拷贝，这也是Linux系统无法处理百万级并发响应的根本原因，有兴趣的可以搜索“零拷贝”的相关文章。

4.2Cache大小的影响

的访问延时。

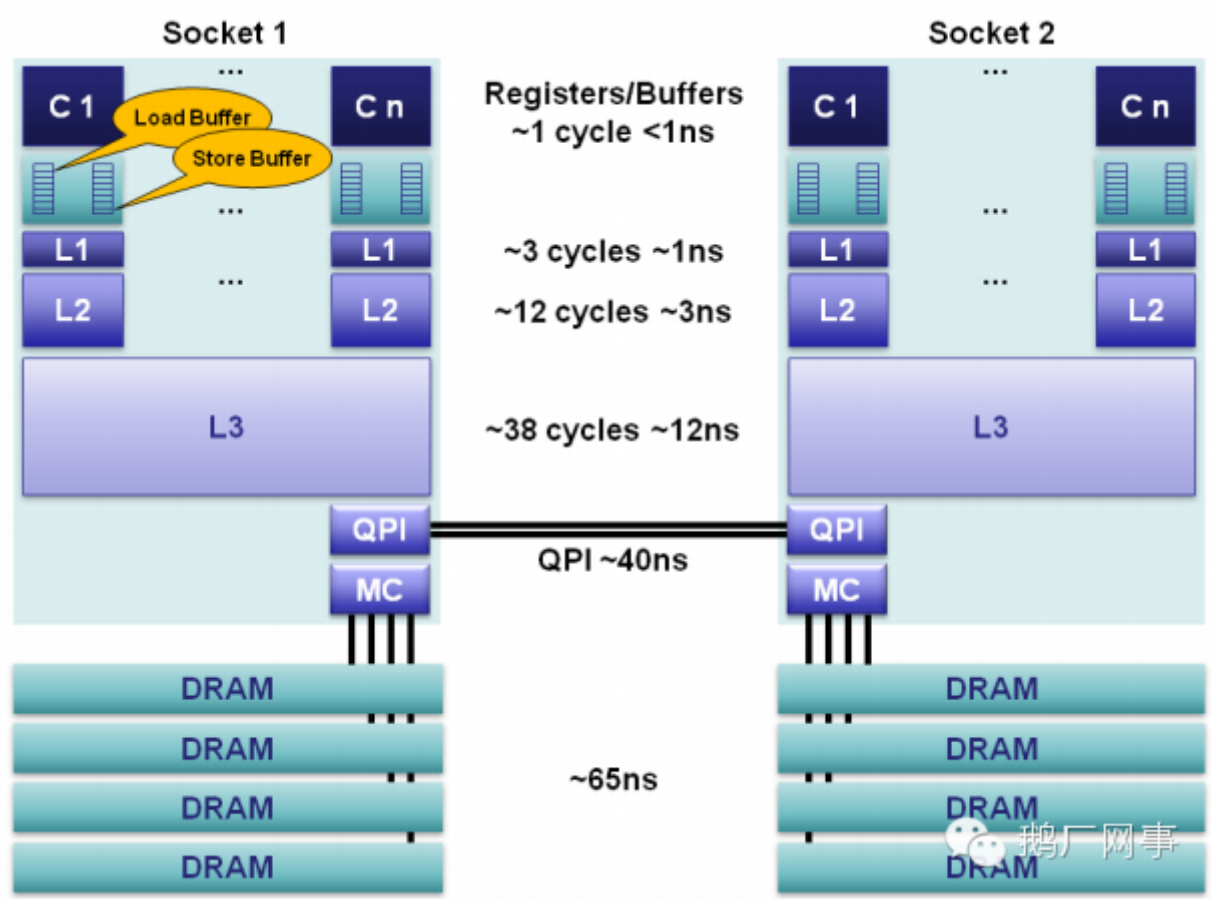


图5 三级Cache性能模型

Cache有效性得益于空间局部性（附近的数据也会被用到）和时间局部性（今后一段时间内会被多次访问）原理，通过合理的使用cache，能够使得应用程序性能得到大幅提升。下面举一个实际的例子来让大家理解cache大小对程序性能的影响。

```
static int arr[1024*1024*1024]={0};

int k=16;

void fun()

{

int i=0;

for (i=0;i<>

arr[i]*=3;

}

int main()

{

fun();

return 0;

}
```

不同K值的测试结果：

K取值	执行时间(s)
16	2.65
32	2.53
64	2.49
128	2.45
256	2.45
512	2.44
1024	2.44
2048	1.22
4096	0.61
8192	0.31

结论：k=16~1024时，代码执行时间和指令数无关，与cache关系更大；k>1024时，cache-misses时次数成倍减少而导致总执行时间成倍减少！因此循环执行时间长短由数组的内存访问次数决定的，而非整型数的乘法运算次数

表1 模拟cache大小对程序性能的影响

这里对上面的测试结果解释一下：在K<1024时，访问数组arr的步长不超过1024*4byte=4kb，而我们的测试机器l1><1024时候，cache从内存读取数据次数是一样的，所以执行时间差别不大；而当k>1024时候，访问数组步长为cache大小的倍数，当然访存次数也成倍较少（可以通过perf工具来跟踪分析），因此执行时间成倍减少。Cache大小可以通过命令lscpu、cat/proc/cpuinfo，或者在目录/sys/devices/system/cpu/cpu0/cache/中进行查看。

熟知cache的大小，了解程序运行的时间和空间上局部性原来，对于我们合理利用cache，提升性能非常重要。同时要少用静态变量，因为静态变量分配在全局数据段，在一个反复调用的函数内访问该变量会导致cache的频繁换入换出，而如果是使用堆栈上的局部变量，函数每次调用时CPU可以直接在缓存中命中它。最后，循环体要简单，指令cache也仅仅有几K，过长的循环体会导致多次从内存中读取指令，cache优势荡然无存。

4.3避免False Sharing

多线程中为了避免上锁，可以使用一个数组，每个线程独立使用数组中的一个项，互不冲突。从逻辑上看这样的设计非常完美，但实际中运行速度并没有太大改善，原因就在下面慢慢来解释了。

cache line是cache从主存copy数据的最小单元，cpu从不直接访问主存，而是通过cache间接访问主存，在访问主存之前会遍历一遍cache line来查找主存地址是否在某个cache line中，如果没找到则将内存copy到cache line中，然后从cache line获取数据。

多核CPU中每个核都拥有自己的L1/L2 cache，当运行多线程程序时，尽管算法上不需要共享变量，但实际执行中两个线程访问同一cache line的数据时就会引起冲突，每个线程在读取自己的数据时也会把别人的cacheline读进来，这时一个核修改改变量，CPU的cache一致性算法会迫使另一个核的cache中包含该变量所在的cache line无效，这就产生了false sharing（伪共享）问题。

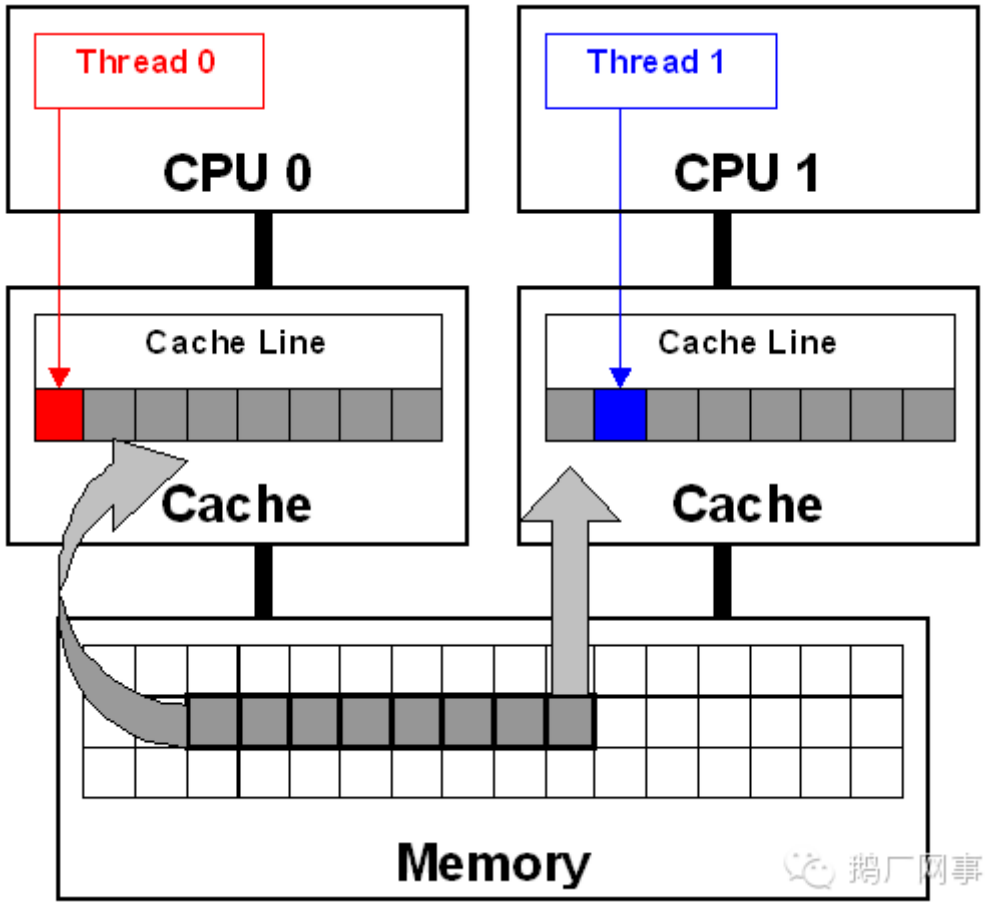


图6 false sharing示意图

```
private static int[] s_counter = new int[1024];

private void UpdateCounter(int position)

{

for (int j = 0; j < 100000000;="" j++)="">

{

s_counter[position] = s_counter[position] + 3;

}

}
```

图7 false sharing测试代码

下面总结一下文章《Avoiding and IdentifyingFalse Sharing Among Threads》里面的建议：访问全局变量和动态分配内存是falsesharing问题产生的根源，当然访问在内存中相邻的但完全不同的全局变量也可能会导致false sharing，多使用线程本地变量是解决false sharing的根源办法。

当然，在平时编程中避免不了要使用全局变量时，这时可以将多个线程访问的变量放置在不同的cache line中，这样通过牺牲一些内存空间来换取高性能。

4.4对齐

Channel是指内存上北桥上面的独立内存接口，一个内存通道一般由64位数据总线和8位控制总线构成，不同通道可以并行工作，当有两个channel同时工作就是我们平时所说“双通道”，其效果等价于128位数据总线的带宽。Rank是指DIMM上通过一部分或者所有内存颗粒产生的一个64位的area，同一条内存上的不同rank因为共享数据总线而不能被同时访问，但内存可以利用交错的片选信号来访问不同rank中数据。

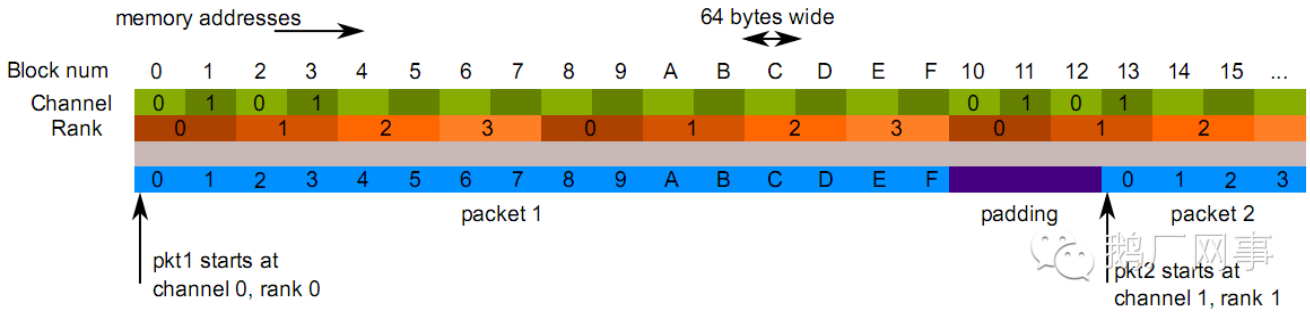


图8双通道、4R的DIMM

在以上的情形中，假定数据包是64bytes大小，那么最优的对齐方式是在每个数据包间填充12bytes。

其次谈到的是字节对齐：众所周知，内存最小的存储单元为字节，在32位CPU中，寄存器也是32位的，为了保证访问更加高效，在32位系统中变量存储的起始地址默认是4的倍数（64位系统则是8的倍数），定义一个32位变量时，只需要一次内存访问即可将变量加载到寄存器中，这些工作都是编译器完成的，不需人工干预，当然我们可以使用__attribute__((aligned(n)))来改变对齐的默认值。

最后谈到的是cache对齐，这也是程序开发中需要关注的。Cache line是CPU从内存加载数据

照cache line边界对齐，就会多读写一次内存和cache了。

4.5了解NUMA

为了解决单核带来的CPU性能不足，出现了SMP，但传统的SMP系统中，所有处理器共享系统总线，当处理器数目越来越多时，系统总线竞争加大，系统总线称为新的瓶颈。
NUMA（非统一内存访问）技术解决了SMP系统可扩展性问题，已成为当今高性能服务器的主流体系结构之一。

NUMA系统节点一般是由一组CPU和本地内存组成。NUMA调度器负责将进程在同一节点的CPU间调度，除非负载太高，才迁移到其它节点，但这会导致数据访问延时增大。下图是2颗CPU支持NUMA架构的示意图，每颗CPU物理上有4个核心。

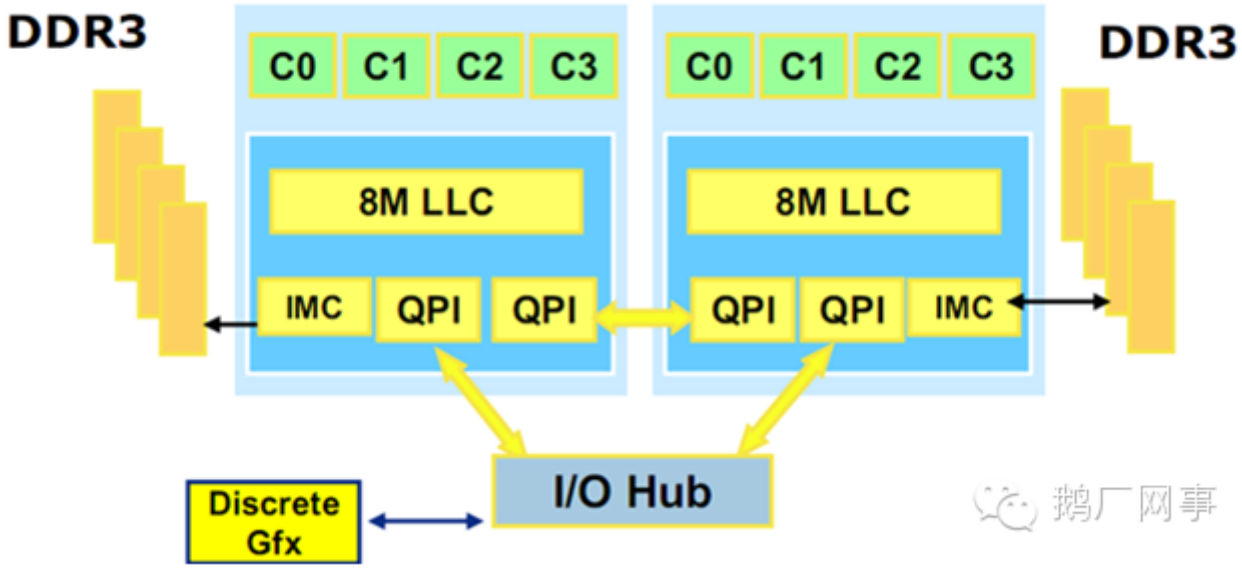


图9 NMUA架构示意图

需要仔细斟酌和合理的使用内存，避免CPU访问远端内存产生不必要的延时，也要留心在高速处理数据包时受到QPI总线带宽的瓶颈。由于业务逻辑目前还无法做到如此简洁，QPI并未成为系统瓶颈，但腾讯Bobcat项目在处理在转发高达10G流量时就碰到了这个问题！

注意，并不是公司的所有服务器都支持NUMA的，这个功能需要操作系统、CPU和主板同时支持，可以通过numactl --show来查看numa是否有效，目前L2机型是支持了NUMA架构的，普通的C1、B6等服务器则没有支持NUMA特性。

4.6减少进程上下文切换

进程上下文切换（context switch，简称CS）对程序性能的影响往往会被大家忽视，但它其实一直在默默扼杀着程序的性能！上下文切换是指CPU控制权由运行任务转移到另一个就绪任务所发生的事件：此时需要保存进程状态和寄存器值等，不仅浪费了CPU的时钟周期，还会导致cache中进程相关数据失效等。

那么如何来减少进程上下文切换呢？我们首先需要了解哪些场景会触发CS操作。首先就介绍的就是不可控的场景：进程时间片到期；更高优先级进程抢占CPU。其次是可控场景：休眠当前进程(pthread_cond_wait)；唤醒其它进程(pthread_cond_signal)；加锁函数、互斥量、信号量、select、sleep等非常多函数都是可控的。

对于可控场景是在应用编程需要考虑的问题，只要程序逻辑设计合理就能较少CS的次数。对于不可控场景，首先想到的是适当减少活跃进程或线程数量，因此保证活跃进程数目不超过CPU个数是一个明智的选择；然后有些场景下，我们并不知道有多少个活跃线程的时候怎么来保证上下文切换次数最少呢？这是我们就需要使用线程池模型：让每个线程工作前都持有带计数器的信号量，在信号量达到最大值之前，每个线程被唤醒时仅进行一次上下文切换，当信号量达到最大值时，其它线程都不会再竞争资源了。

现代处理器都是通过多级流水来提高指令执行速度，为了保持流水线充满待执行指令，CPU必须提前获取指令。当程序中遇到分支或条件跳转语句时，问题就来了，处理器不确定下一条指令，这是就会使用分支预测逻辑来判断进入流水的下一条指令。

从P5处理器开始引入了分组预测机制，如果预测的一个分支指令加入流水线，之后却发现它是错误的分支，处理器要回退该错误预测执行的工作，再用正确的指令填充流水线。这样一个错误的预测会严重浪费时钟周期，导致程序性能下降。《计算机体系结构：量化研究方法》指出分支指令产生的性能影响为10%~30%，流水线越长，性能影响越大。Core i7和Xen等较新的处理器当分支预测失效时无需刷新全部流水，当错误指令加载和计算仍会导致一部分开销。

下面给出一个分支预测例子的结论：对一个已排序好的数组中每个元素进行分支判断，预测成功率在90%以上，而对于随机数组进行预测，成功率只有50%。当然，我们还可以通过将一些简单的分支逻辑转换为位运算来避免预测失效，虽然比较晦涩难懂，例如：

改进前	**改进后**
if (data[c] >= 128) // 条件分支	int t = (data[c] - 128) >> 31;
sum += data[c];	sum += (~t & data[c]);

表2 一种减少分支预测的晦涩方法

具体的测试结果，可以搜索文章《Whyis processing a sorted array faster than an unsorted array》查阅。

用likely、unlikely预处理指令，来指示编译器在生成汇编代码时候对指令进行优化，加快执行速度。

这两个宏在内核中定义如下：

#definelikely(x) __builtin_expect((x),1)

#defineunlikely(x) __builtin_expect((x),0)

分支预测中最核心的是分支目标缓冲区（Branch Target Buffer，简称BTB），每条分支指令执行后，都会BTB都会记录指令的地址及它的跳转信息。BTB一般比较小，并且采用Hash表的方式存入，在CPU取值时，直接将PC指针和BTB中记录对比来查找，如果找到了，就直接使用预测的跳转地址，如果没有记录，必须通过cache或内存取下一条指令。下面的一个测试程序演示了BTB满后导致分支预测失败数量迅速增长，测试在tlinux下使用了perf工具来跟踪结果：

代码示例

```
int f(void)
{
int num=10;
int i,res;
for (i=0;i<>
res+=1;
```

测试过程

不同num值的测试结果：

num	branches	branch-misses	task-clock(ms)
10	150,134,871	3,892(0.00%)	524.250013
20	250,188,217	5,183(0.00%)	829.384594

```
int main()
{
int i=1;
for (i=0;i<>
f();
return 0;
}
```

40	450,317,966	10,004,191(2.22%)	1467.548130
50	550,393,674	10,004,542(1.82%)	1780.274314

注：branches表示分支命中次数；branch-misse表示分支预测失效次数；task-clock表示任务执行的时间，是真正占用处理器的时间。

结论：当num增加到40的时候，分支预测率急剧上升，说明BTB的大小介于30-40之间，很可能是32个！

表3 BTB对分支预测成功率的影响

4.8利用流水线并发

下面我们从指令级并发的角度来考察从cache对程序性能的影响。

```
int[] a = new int[2];
for (int i=0; i循环1
for (int i=0; i循环2
```

表4 流水线对程序性能的影响

在我们的测试机上运行这两个循环，第一个循环需要1.6s，第二个循环只要0.8s，这是为什么呢？因为第一个循环体内，操作相互依赖，必须等第一次a[0]++执行完后才能执行后续操

这个原因其实就是和CPU中的流水线有关，像Pentium处理器就有U/V两条流水，并且可以独自独立读写缓存，循环2可以将两条指令安排在不同流水线上执行，性能得到极大提升。另外两条流水线是非对称的，简单指令（mpv,add,push,inc,cmp,lea等）可以在两条流水上并行执行、位操作和跳转操作并发的前提是在特定流水线上工作、而某些复杂指令却只能独占CPU。

需要补充说明，因为上面举得例子非常简单，而在实际代码编写过程中，往往使用的变量非常多，逻辑也比较复杂，数据在内存中的分布也会影响cache的命中次数，并且分支预测导致的CPU中指令乱序执行，非常难去分析执行流程。因此在程序设计中，要做到指令级并行，需要有意识的注意指令间的配对，尽量使用简单指令，还要在顺序上减少上下文的依赖。

4.9适度预取

为了利用空间局部性，同时也为了覆盖数据从内存传输到CPU的延迟，可以在数据被用到之前就将其调入缓存，这一技术称为预取Prefetch，加载整个cache即是一种预取。CPU在进行计算过程中可以并行的对数据进行预取操作，因此预取使得数据/指令加载与CPU执行指令可以并行进行。

预取可以通过硬件或软件控制。典型的硬件指令预取会在缓存因失效从内存载入一个块的同时，把该块之后紧邻的一个块也传输过来。第二个块不会直接进入缓存，而是被排入指令流缓冲器（Instruction Stream Buffer）中。之后，当第二个内存访问指令到来时，会并行尝试从缓存和流缓冲器中读取。如果该数据恰好在流缓冲器中，则取消缓存访问指令，并将返回流缓冲器中的数据。同时，发出起一次新的预取。如果数据并不在流缓冲器中，则需要将缓冲器清空。

代码，并把预取指令适当地插入其中。这类指令直接把目标预取数据载入缓存。如果我们在编程中能显示的调用预取指令，就能大大提高效率。如果读取的内容仅仅被访问一次，prefetch也没有意义。

在使用预取指令时，必须考虑调用时机和实施强度。如果过早地进行预取，则有可能在预取数据被用到之前就已经因为冲突置换被清除。如果预取得太多或太频繁，则预取数据有可能将那些更加确实地会被用到的数据取代出缓存，反而会增加开销。

一开始在处理进程开发过程中增加了大量预取操作，但是性能反而下降了，因为在处理进程中对于每个数据包分析逻辑比较复杂，数据预取填充的cache很快就被业务逻辑指令和数据替换了无数遍吗，因此预取一定要得当。

下面给出DPDK封装的一些预取指令操作以供参考，在自研UDP服务器中对rte_prefetch0()函数的调用进行了合理安排，性能提高不少。

函数实现	**功能**

static inline void rte_prefetch0(volatile void *p){
asm volatile ('prefetcht0 %[p]' : [p] '+m' (*(volatile char *)p));
}
	预取数据到所有级别的缓存中

static inline void rte_prefetch1(volatile void *p){	预取数据到除L0外所有级别缓存

}

static inline void rte_prefetch2(volatile void *p){
asm volatile ('prefetcht2 %[p]' : [p] '+m' (*(volatile char *)p));
}
	预取数据到除L0和L1外所有级别缓存中

表5 DPDK预取操作的实现

4.10合理编写循环

先从一个简单的例子入手：

函数F1实现	函数F2实现

int i=0,j=0;
int i=0,j=0;
static a[1000][500000] = {0};
for (j=0;j<>
for (i=0;i<>
a[i][j] = 2*a[i][j];
	int i=0,j=0;
	static a[1000][500000] = {0};
	for (i=0;i<>
	for (j=0;j<>
	a[i][j] = 2*a[i][j];
	}

```
}
```

表6 两种循环编写方式对比

运行左边的程序需要10s，运行右边的程序只需不到4s。产生这个现象的原因就是右边的例子中因为a[i][0]访问cache失效后，会从内存中读取至少一条cache line数据（64bytes），因此cache中充满了a[i][0]~a[i][15]的结果，这样后面15次循环都可以命中cache了。

下面是我们分别对左右程序进行cache命中率跟踪的对比结果：

#perf stat -e cache-references -ecache-misses -e L1-dcache-loads -e L1-dcache-load-misses -e dTLB-loads -edTLB-load-misses -e instructions program

```
Performance counter stats for './t.o':

      174,921,321 cache-references                                  [71.43%]
       33,226,050 cache-misses              #   18.995 % of all cache refs     [71.41%]
    3,820,803,746 L1-dcache-loads                                   [71.38%]
    1,099,969,105 L1-dcache-load-misses     #   28.79% of all L1-dcache hits   [71.45%]
    3,820,107,116 dTLB-loads                                        [57.20%]
      501,119,127 dTLB-load-misses          #   13.12% of all dTLB cache hits  [57.15%]
   10,182,385,158 instructions              #    0.00  insns per cycl          [71.42%]

     10.083969612 seconds time elapsed
```


图10 左侧实例程序性能测试结果

```
 Performance counter stats for './t2.o':
    33,795,594 cache-references                                   [71.38%]
    31,530,470 cache-misses             #   93.298 % of all cache refs    [71.48%]
 3,819,101,479 L1-dcache-loads                                    [71.50%]
    36,010,100 L1-dcache-load-misses    #    0.94% of all L1-dcache hits  [71.49%]
 3,817,421,646 dTLB-loads                                         [57.17%]
     2,704,022 dTLB-load-misses         #    0.07% of all dTLB cache hits [57.07%]
10,169,078,725 instructions             #    0.00  insns per cycle        [71.31%]

   3.845269998 seconds time elapsed
```


图11 右侧实例程序性能测试结果

因此，数据cache大小非常有限，循环中数组访问的顺序非常重要，对性能的影响不容小觑。

另外，如果两个循环体可以合并到一个循环而不影响程序结果，则应该合并。因为通过合并，原来第二个循环中的指令会在指令cache中被命中。

其次，指令cache和数据cache一样都非常小，循环中编写的指令一定要精简，非循环内部的操作可以放到外面去，否则一旦循环体中指令长度超过cache大小就会导致不必要的置换。

再次，需要考虑cache的置换策略，例如cache使用的是LRU算法的话，在编写多层嵌套循环时需要考虑被置换出去的数据越少越好。

最后，如果能少用循环的话就更好了！

4.11其它优化建议

尽量减少函数调用，每一次调用都要进行压栈、保存寄存器和执行指令跳转等都会耗费不少时间，可以将一些小的函数写成内联，或直接用宏或语句代替。

对于提高性能空间换时间也是一个非常重要的设计理念，Bloom Filter就是一个典型的例子，还有一些位图、hash的思想也是不错的选择。

在高性能程序设计时，要减少过保护，除了会影响程序执行的一些关键路径和参数要进行校验外，其它参数不一定非得要检查，毕竟错误情况是少数。

尽量用整型代替浮点数，少用乘除、求余运算，这些操作会占用更多的CPU周期，能提前计算的表达式要提前计算出结果。

充分利用编译器选项，-O3帮助你进行文件内部最深层次的优化，使用其它编译器如icc能编译出在x86平台下运行更快的程序。

延时计算，最近用不上的变量就不要去初始化，操作系统为了提高性能也使用COW(copy-on-write)策略，在fork子进程的时候不会立即复制进程的所有页表。

当然，当你花费大量的时间也就是为了利用硬件的某个特性，冥思苦想之后或许获取了一点点性能的提升，其实更重要的是在程序设计时候换一种思路，也许就会柳暗花明，从逻辑上或算法上优化获得的效果可能远远大于针对硬件特性优化的效果，在程序设计之初一定要深入理解业务逻辑，最大程度上简化流程，针对硬件特性的优化一定要是在程序逻辑优化之后进行才能更有效。

五、应用实践

在高性能服务器程序开发中，通过网卡中断收发报文是第一道瓶颈，为了避免中断方式的网卡驱动，可以通过卸载Linux自带的网卡驱动，代而使用DPDK提供的PMD模式的驱动，避免了

相比较Linux默认的4K页表，在应用程序中使用1G的大页表，可以大大减少缺页中断，提升内存的访问速度。

在CPU亲缘性设置方面，将Linux运行在单独的内核上，实际的数据收发包或者处理逻辑绑定在单独的内核上，避免进程间竞争和上下文切换。

5.1 高性能网关设备

为了方便在不同数据中心间进行数据传输，一般通过建设专线来满足要求。但是专线建设成本高，时间长，当容量增长过快时扩容缓慢，如果在基础架构侧能够利用公网来进行一部分数据的传输，就能缓解专线的压力了。另一方面，基于网络的容灾需求，大部分专线利用率都低于50%，同时专线是按带宽收费的，浪费了一半以上的成本，如何提高专线利用率的同时还能够低成本的保证网络容灾需求，成为一个必须解决的问题。大流量的公网传输平台在这种背景下应用而生，但要达到此目的，一定要避免成为数据转发的瓶颈。因此必须保证数据报文转发的高性能。

Bobcat是自研的利用公网来传输业务数据的一种网关设备，它具备报文收发，寻址，报文封装和校验等一些列功能。在实现上，bobcat仅使用一个CPU核心来运行Linux操作系统，依赖于Linux完整的协议栈来实现管理功能。其它CPU核心都用来做IPP（Ingress Packet Processor）、EPP（Egress Packet Processor）以及报文转发功能。

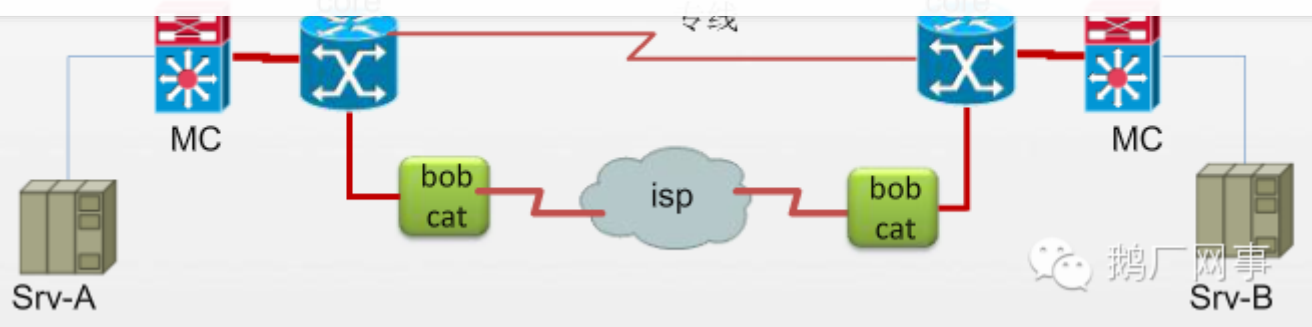


图12 报文在Bobcat中的传输示意图

具体工作流程如下：svr将包传递到城域网核心，根据路由来决定是否跨城；包到达了Wan，则根据DSCP值来区分专线和大流量平台；在包到达Bobcat平台后，将进行目标Bobcat的查表，头部封装，同时打散成多个传输通道给于传递；bobcat收到传递过来的报文则进行头部校验，需要保证不给篡改和重复的序列号；并向接收端的Wan核心转发；wan core这根据目标地址传递man core。

由于引入多核，Bobcat需要处理报文发送的时序问题，多核的并行处理会导致EPP后的报文乱序，因此要对报文进行重排序，确保从物理端口发出去的顺序和收到的顺序一致，因此这里EPP仅有单核来处理，避免重排序的复杂性，当然系统的整体性能也取决于EPP的转发能力。在实际测试中，Bobcat在转发64bytes小包时几乎可以达到10Gbps的线速。

5.2 高性能服务器程序

基于UDP的无状态特性，可以很方便的使用DPDK来提供底层的收发数据包框架，自己实现协议的解析和处理过程，而不用借助于繁重的Linux内核协议栈。

亲缘性设置方面，在一台双CPU的12颗物理核的机器上，因为开启了超线程，逻辑核实际是24个，但由于两颗CPU之间的通信需要使用QPI，会增大报文处理时延。因此，只利用其一颗

用来收发网卡数据包，并均匀的将数据包放到和处理进程公用的9个无锁队列中。

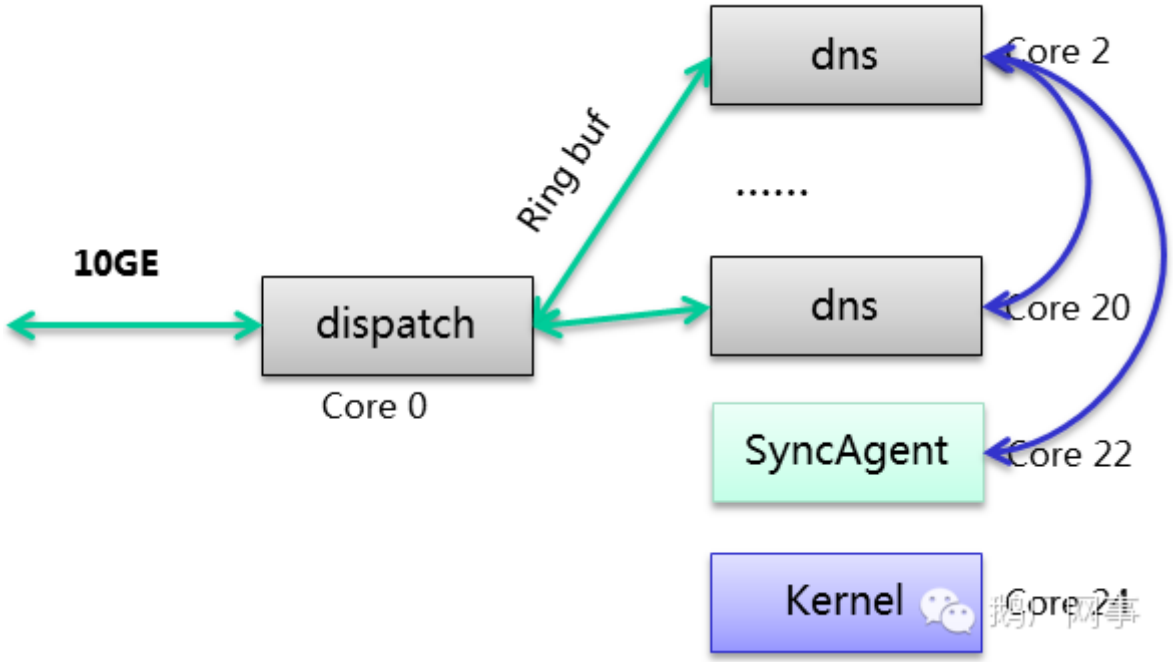


图13 一种UDP服务程序的亲缘性设置方案

另外自己来处理数据包，从以太网数据帧开始向上层协议分析，并进行必要的校验，只过滤出需要处理的UDP报文，不去使用复杂且低效的Linux协议栈来处理报文。

数据包的读取都是使用批量操作，当网卡队列在收到32个数据包后，一次性将数据包传输到内存中；同时使用HPET时钟（RDTSC指令也行）定时处理不够32数据包的情形，避免响应延时。

buf里，供处理进程来获取，避免了加锁的等待延时。使用了预取操作，主要用在了从网卡队列读取数据包进行检验后传递到内存的过程中，这样在对数据帧校验过程中的同时，也并行地将下一个数据包放到cache当中，节省了数据传输延时。

合理的使用分支预测，在大概率条件语句前加上likely，反之加unlikely。

预分配内存，依据队列大小，为每个进程预先独立分配UDP报文处理所需要的空间，在构造UDP应答包结构的时候直接通过空闲指针链表获取即可。避免拷贝数据包。Dispatch进程将数据报文通过DMA方式传递到内存之后，只是将报文地址放入到ring buf，UDP处理进程可以在原地址处直接来解析数据包并就地修改，然后通知Dispatch是否要由网卡发出去，或者丢弃。

数据包头部和尾部预留空间，这样在修改数据包的内容或填充新的头部、尾部的时候不需要重新申请空间了，而直接在原处修改即可。减少内存访问是优化的重点，也是难点，上面的很多方法其实都可以减少内存访问，但最重要的是要在程序中避免复杂变量的拷贝，多使用指针，这也需要非常小心的编写代码。也不能为了减少内存拷贝而把所有字符串复制都修改为指针，这将导致极难维护和调试。

通过硬件指令加速hash计算，甚者直接使用intel的crc指令来计算hash，比传统纯软件hash算法性能大幅提升。避免过渡校验：一个未被修改的参数从头到尾只需要校验一次，事先能确保字符串以\0结尾后面也可以减少一些判断机制。

还有很多其它的用来提高程序性能的编程方法和技巧可能没列举出来。总之，在设计和开发系统之前需要对硬件和操作系统有较深入的了解，并保持着以追求性能为目标的心态来编写每个函数和模块，那么写出来的程序性能也不会差了。在高性能服务程序设计过程中，也不可能把

六、参考文献

[1]. *www.kegel.com/c10k.html*，Internet

[2]. *c10m.robertgraham.com*，Internet

[3]. *I ntel Data Plane Development kit：Software Architecture Specification*， Intel

[4]. *Avoiding and Identifying False Sharing Among Threads*， Intel

[5].《大话处理器》，清华大学出版社

上一篇：分布式系统一致性问题解决实战

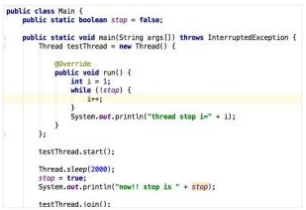
下一篇：线上服务CPU100%问题快速定位实战

类似文章

更多

ODP/DPDK代码级性能优化小结Tips

ODP/DPDK代码级性能优化小结TipsODP/DPDK代码级性能优化总结Tips ODP/DPDK代码级性能优化总结Tips.所以要提高性能，一定要减小cache miss! 用Perf经常记录cache miss百分比。DPDK配置：这些是gdb单步出...



Java 并发原理无废话指南

解释这个程序就用到了“多级存储”，在x86架构的CPU中对数据的的访问都是经过寄存器，如果数据在内存中CPU会先加载到寄存器然后在读取；...

cpu的一级缓存是什么？~电脑专业术语普及

缓存对CPU的性能影响很大，主要是因为CPU的数据交换顺序和CPU与缓存间的带宽引起的。一级缓存可以分为一级数据缓存(Data Cache，D-Cache...

为什么Excel水平的高低，决定着你的薪资

以前，在网上收集数据，看到数据后，还得一个个手动录入到Excel表格当中，如果在数据量很大的情况下，一个个录入，可能一天时间就过去了...

什么是云主机 云主机能干什么

云主机做什么好

1.3万阅读

台式机硬件常识(买机前一定要看)

CPU的缓存分两个，一个是内部缓存，也叫一级缓存（L1 Cache）：封闭在CPU芯片内部的高速缓存，用于暂时存储CPU运算时的部分指令和数据，存取速度与CPU主频一致。外部缓存，也叫二级数据缓存（L2 Cache...

linux内核中的内存屏障

linux内核中的内存屏障前言 之前读了关于顺序一致性和缓存一致性讨论的文章，感觉豁然开朗。编译器不是要打乱代码执行顺序吗，处理器不是要乱序执行吗，你插入一个内存屏障，就相当于告诉编译器，...

内存屏障浅析

内存屏障浅析。内存屏障的分类：编译器引起的内存屏障缓存引起的内存屏障乱序执行引起的内存屏障。1、编译器引起的内存屏障：就是CPU和外部设备访问内存的时候都需要经过总线的仲裁，有一个专门的硬件...

云主机做什么好

3.1万阅读

本书的使用指南（深入理解计算机系统（原书第2版））书评

本书的使用指南（深入理解计算机系统（原书第2版））书评 Chapter 1A Tour of Computer System一个对计算机系统总体的介绍，简单明了。计算机正是通过时间片技术使得每个进程在执行时仿佛独占CPU...

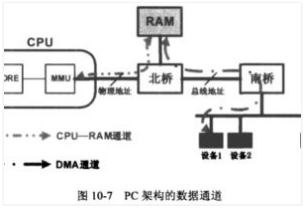


图 10-7 PC 架构的数据通道

DMA描述符及映射

DMA描述符及映射1.DMA通道。从图中可以看出，Linux内核中的DMA层为设备驱动程序提供标准的DMA映射接口，例如一致性映射类型的dma_alloc_...

电脑硬件基础知识之内存

内存：内存(Memory)也被称为内存储器，内存就是内存储器的简称，其作用是用于暂时存放CPU中的运算数据，以及与硬盘等外部存储器交换的数...

什么是云主机 云主机能干什么

云主机免费推荐吗

1.3万阅读