Report On Affinity Accept

Improving Network Connection Locality on Multicore Systems

这篇论文我目前只看完了前四个section，后面就是实验验证了。看得比较困难。

本文主要阐述了在多核系统的通信中，如何提高TCP端口接收和处理效率的问题。

**本文主要创新点如下：**

（1）设计了Affinity-Accept的方式来执行TCP端口的数据接收问题，主要有两个要点；

（2）在数据接收部分，设计了可以根据任务的相关性（Affinity）来分配任务到相应的核的以太网接口；

（3）在核端，针对于不同的核在性能和任务调度的不平衡，设计了用于平衡核的负载，来提高运行效率的方法。

（4）实验结果看来，在48核AMD系统中，Affinity-Accept方法TCP栈中降低了30%的运行时间，提高了24%的吞吐率。

**本文的解决思路如下：**

首先，文章针对多核架构下的TCP协议栈，提出了最优的策略（性能最好）就是每一个任务都可以在一个独立的核中完成包括数据接收，数据处理和数据发送等操作。这是因为核间（inter core）的操作执行时间长，性能衰减很快。此外，多个核处理同一个任务时，会导致cache一致性问题，严重的影响了性能。但是由于，TCP端口的结构，不同的核以及应用的不同，任务很难保证在同一个核上完成处理。传统的方法有三个不足：

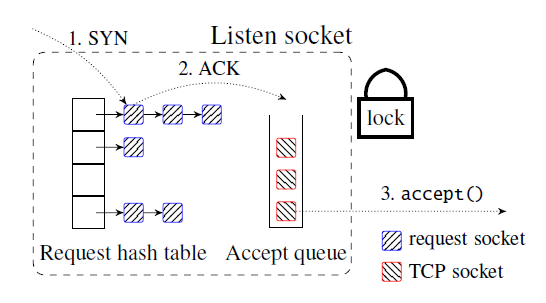
（1）接收数据的核是被中断触发的核，并一定是处理数据任务的核。

（2）独立完成任务可能会和该核现有的调度方法有冲突（会累积大量的任务）

（3）不同的应用需求不同。

为了实现这个目标，本文设计了Affinity-Accept的方法，在接口端（port）增加用于分配任务的NCI，并为每一个核单独设计一个队列，作为NCI的输出对象；在处理器端，每一个处理器针对不同的负载则设计了平衡负载的调度方法。

首先是连接（Connection）的分配问题。本文中TCP采用三次握手的方法来建立一个连接。在这个过程中，服务器（server）和终端（client）分别会各有一次SYN数据包和ACK信号。在这个基础上，文中设计了一种可以跟踪数据包，分析并转发的架构。如下图所示。



这是一个通信接口，由一个锁来保护。其中当SYN包进入的时候，数据被存入哈希表（hash）。当ACK进入时，则意味这连接已经建立了，数据有效，此时根据包中ID元组（实际上选择了5元组的一部分，目的是减少数据量，节约存储空间）的哈希值来选择存入接收队列（Accept Queue）中。进入接收队列的数据即是待处理的有效数据。此时以太网接口（NCI）的任务已经完成，将有处理器决定调用系统调用accept()的时候，将数据加载到内核中。

（Q：根据前后的语义，感觉文中描述的这个TCP接口是client端的。我不确定到底是client还是server，这两者有什么本质的区别。）

这一部分面临的挑战有三个：（1）由于端口由一个lock来保护，因此每次只能有一个core来访问accept queue，这一点没有发挥出多核的优势。（2）数据包需要不同的核分别对Accept Queue读写，因此在完成任务的时候不同的核之间需要相互访问cache，产生cache一致性问题；还会导致跨核寻址操作，代价很大。

为了解决这个问题，Affinity-Accept的主要改进有两处：

（1）Accept Queue被拆分成多个，每一个核都对应着一组收发DMA。在数据的转发过程中，由于数据是由以太网接口（NCI）根据其hash值来选择其对应的核，而接收队列的硬件本质是每一个核对应的自己的DMA。因此每一个内核加载对应数据的操作是本地操作，性能很高

（2）将TCP端口中的操作分成多个小的部分，并由多个更细粒度的锁（lock）来保护。这样就允许不同的核分别来完成port的数据收发操作。这有利于多核充分利用。

除了连接的分配问题（实质上就是3.1 Connection Routing in the NIC描述的问题），第二部分是处理器加载数据的问题。在Affinity-Accept中，与传统的方式不同，所有的数据包都会被分配到对应处理器上，因此处理器在选择数据进行加载的时候也会优先选择本地正在处理的任务的数据，以提高效率。

以上就是Affinity-Accept的主要内容。但是，在实现的这个新的架构的同时，也带来其他的问题。最重要的就是负载调度的问题。由于数据包不再是均匀的分配给不同的核，所以，需要设计算法来面对单核负载过重的问题。主要包含两种情况：

（1）突发性负载，导致短暂峰值（Short-term Load Spike）。

在这种情况下，部分数据包需要挪到空闲的处理器上处理，即连接盗窃（Connection Stealing）。解决办法就是对处理器设计一个状态busy/non-busy。其中，这两个状态的判断是依据对应的Accept Queue的长度，的转移是一个滞回的过程。给定一个阈值“最大接收队列长度”(MAQL)，当前队列长度（Instantaneous Queue Length）大于75%的阈值时，进入busy状态；而当队列长度均值（EWMA）小于10%的阈值时，处理器进入non-busy状态。

在处理器调用accept()时，busy状态下的处理器只接收本地的数据；non-busy的处理器则会按照5:1的比例接收本地数据和busy处理器的数据。（这个比例只是文中给出的数据，并没有理论依据）这样就可以降低busy的处理器的负载，但是会引入核间处理的代价。对于non-busy的处理器来说，选择non-busy的处理器是根据一个存放着busy处理器的环形链表来选择，因此并不是最优的。（这个方法文中表示需要在未来的工作中研究）

（2）对于长期的重负载（Long Term Load Imbalance）

对此采用的方法是数据流组搬移（Flow Group Migration）。这个方法是建立在Connection Stealing的基础上的。再一个较长的时间段中（100ms，也是实验数据，没有理论依据），non-busy的处理器会统计最多的busy核的并从中搬移一部分数据流组（Flow Group）到本地。每一个数据流组即是对应着一组hash数值，因此可以理解成将应用处理全部搬移到本地。（这个算法也只是一个为了应对当前问题的定性描述，并没有进行优化或者理论分析。）

最终，本文还讨论了不同的应用，会调用不同频率的系统调用对Affinity-Accept的影响。

如前所述，实验部分还没有看。如果能的话，这个文章看得比较吃力，希望做看的另一篇文章的PPT来讲。

**本文结果的验证指标：**

即性能的验证指标，包括任务的运行时间和TCP端口的吞吐率。

**本文的不足之处：**

（1） 如思路中所指出的几处红字部分，文中有几个工作没有给出合理的推测，也没有给出合理的分析。

（2）关于处理器的busy和non-busy状态，由于两个的评估对象分别是瞬时的队列长度和平均的队列长度，因此可能会在由busy进入non-busy的过程中产生冲突，即：平均队列长度很短，可以进入non-busy，但是临时队列长度很长，应该进入或者保持busy。这一点没有讲清楚。