文章编号:1004-5694(2004)01-0057-03

一种硬件哈希表压缩方法及其性能研究*

张 勇, 雷振明
(北京邮电大学 信息工程学院,北京 100876)

摘 要:在高速、大容量的基于流的报文分类应用中,采用硬件哈希表具有成本低、扩展性好等优点。但由于需要在硬件哈希表中保存流标识,而流标识的长度依不同应用可能长达几十字节,一方面需要较大的存储空间,另一方面也严重影响了哈希查表的性能。提出了一种硬件哈希表压缩方法,可以有效压缩保存在哈希表中流标识的长度,减小所需存储器容量,提高查表性能,同时实现复杂度低,具有较高的实用价值。

关键词:硬件哈希表;压缩;查表性能
中图分类号:TP391.13 文献标识码:A

0 引 言

随着因特网的发展,人们已经不再满足于仅仅能够提供尽力服务类型的互联网络,对于网络所能提供的业务质量提出了各种不同的要求。为解决这个问题,相应出现了区分服务[1,2],集成模型[3,4]等解决方案。在区分服务中的用户接入点,根据用户的SLA(service level agreement)对到达的报文进行分类,依照分类的结果对报文进行相应的处理。在集成模型中,需要在整个传输路径中对报文进行基于流的处理。此外,在TCP off-load、Webswitching、防火墙、基于流的记费管理等应用中都需要对到达的报文进行基于流的处理。其具体处理过程如下:当报文到达时,首先需要对到达的报文进行分类,即根据报文的流标识(flow ID)在流表中查找该报文所属的流,然后取出流表内保存的该流对应的处理参数,如流量管理参数、统计参数、优先级参数等,之后根据这些处理参数对报文进行相应的处理。

为了获得高的报文处理能力,通常采用硬件完成上述功能。在硬件实现中,采用基于SDRAM/DRAM的硬件哈希表进行报文分类具有性能高、成本低、扩展性好等优点。但采用硬件哈希表进行报文分类时,为了检测到冲突并执行相应的辟退算法,必须在哈希表中保存各个流的流标识,IPv4中的"源IP地址、目的IP地址、协议类型、源TCP端口、目的TCP端口"的流标识长度为104比特,IPv6中则增加到296比特,当在流标识中增加高层协议部分内容,如在Web switching需要增加Http请求部分的内容,此时,流标识的长度将大大增加,导致每个哈希表项所需存储空间急剧增加。

长的流标识一方面会增加保存每个流所需的存储空间,同时,还会严重影响哈希查表的性能。由于存储器数据总线宽度一定,因此读出长的流标识需要进行多次读操作。例如:320比特的流标识,数据总线宽度为64比特,则一个哈希表项的比较操作就需要5次存储器的读操作;而如果流标识的长度为64比特,则只需要1次读操作。这样,查表性能将相差5倍。

为了解决这个问题,本文提出了一种硬件哈希表压缩方法,可以有效降低保存每个哈希表项所需的存储空间大小,使其与协议无关,与流标识的长短无关,仅与需要同时支持的流的数目有关。代价是会带来一定的分类错误概率,但其错误概率可以降低到完全可以忽略的程度,从而使这种结构具有较高

* 收稿日期:2003-05-14
基金项目:国家重大自然科学基金资助项目(69896240)。
作者简介:张 勇(1976-),陕西人,博士研究生,主要从事高速路由器,宽带网络方面的研究;雷振明,教授,主要从事IP/ATM宽带网络方面的研究。

的实用价值。

1 硬件哈希表压缩方法

这种压缩方法主要是基于如下的动机:

1) 在哈希表中保存流标识的目的在于能够唯一的区分流标识空间中的每一个流。流标识空间中的元素数目很多,如流标识长度为 300 比特时,则可能的流标识数目可以达到2^{300}个。但一个系统同时支持的流的数目却相对有限,如同时支持 1M 个流。

2) 在支持的流固定不变的情况下,区分 1M 个流只需要 20 比特,但由于实际支持的流是在不断动态变化的,不断有新流加入哈希表,同时到期的流被从哈希表中删除掉,因此 20 比特将不能区分这动态变化的 1M 个流。

3) 理想的解决方案是区分这些流所需要的比特数只与流的数目有关,而与具体的流标识的长度无关。

在此基础上,笔者提出了如下的解决方案:选择一定的哈希函数H_{key},计算流标识KEY的哈希函数$H_{\text{key}}(KEY)$,将计算结果保存在哈希表中,用以区分系统中同时存在的流。因此,当报文到达时,首先我们从报文中提取出流标识,计算得到该流的哈希函数计算结果,用该哈希函数计算结果标识这个流,进行哈希查表的各个步骤,如判断表项是否被占用,查表是否结束等等。当新流加入哈希表时,需要将该哈希函数计算结果写入哈希表项中,替代以前方法中的流标识。当采用基于循环冗余校验(cyclic redundancy check,CRC)[5,6]的哈希函数时,由于其有专用的硬件逻辑电路实现并行CRC 计算,所以通常不会成为系统性能的瓶颈。

采用这种方法时,如果KEY_1和KEY_2是2 个不同的流标识,且满足$H_{\text{key}}(KEY_1)=H_{\text{key}}(KEY_2)$,则这种方法将不能区分$KEY_1$和$KEY_2$,将导致部分报文被按照错误的处理参数进行处理。这种情况是不可避免的,下面我们研究如何将其降低到可以完全忽略不计的程度。

假设哈希表中同时支持的流的数目为m个,哈希函数H_{key}将KEY的宽度由R比特压缩到r比特,且哈希函数可以将随机到达的流均匀映射到$0\sim 2^r-1$的哈希计算结果中。则m个流的到达序列中,不具有相同的哈希计算结果的概率$P_{\text{no_conflict}}$可以近似表示为

$$P_{\text{no\_conflict}}=\frac{2^r(2^r-1)(2^r-2)\cdots(2^r-m+1)}{(2^r)^m}>$$

$$(1-(m-1)2^{-r})^{m-1}\approx 1-(m-1)^2 2^{-r} \quad (1)$$

$$P_{\text{conflict}}=1-P_{\text{no\_conflict}}<(m-1)^2 2^{-r}\approx m^2 2^{-r} \quad (2)$$

当$m=2^{20}$,$r=64$ bit 时,P_{conflict}近似为6×10^{-8};当$r=96$ bit 时,P_{conflict}近似为10^{-17};当$r=128$ bit 时,P_{conflict}近似为3×10^{-27}。

当不同流具有相同哈希计算结果的概率为P_{conflict}时,可大致估计一下冲突发生的平均间隔。假设每个新加入的流以等间隔T加入到哈希表中,且每个流的持续时间为$m\cdot T$,则平均每$(m\cdot T)/(m\cdot P_{\text{conflict}})$s 发生一次冲突事件;当$m\cdot T=50$ s、$m=1\,000\,000$、$P_{\text{conflict}}=10^{-17}$时,则平均$6\cdot 10^{12}$ s 才会发生一次冲突,即平均近 20 万年才会发生一次冲突事件。结果是导致持续 50 s 的报文处理错误,因此对于通常的应用,这样的结果是可以接受的,如果需要得到更低的冲突概率,则可以相应增加r的值,如将r从上面例子中的 96 bit 增加到 128 bit,则平均$1.7\cdot 10^{22}$ s 才会发生一次冲突。

下面提出一种方法,可以进一步降低一定冲突概率所需的r值。

通过以上说明,可以知道,当采用上述哈希表压缩方法时,对于到达的流标识,我们需要计算 2 个哈希函数值:一个作为该流的哈希地址,记其哈希函数为H_{addr},它的哈希函数值宽度与哈希表的大小有关,记为t bit,且有哈希表的大小$n=2^t$;另外一个就是前面为了区分流所需要的r bit 的哈希值。在前面的分析中,笔者认为如果 2 个不同的流标识KEY_1和KEY_2满足$H_{\text{key}}(KEY_1)=H_{\text{key}}(KEY_2)$,则将发生冲突,但如果选择恰当的$H_{\text{addr}}$使得$H_{\text{addr}}(KEY_1)\neq H_{\text{addr}}(KEY_2)$,则仍然不会发生冲突,只有同时满足$H_{\text{key}}(KEY_1)=H_{\text{key}}(KEY_2)$和$H_{\text{addr}}(KEY_1)\neq H_{\text{addr}}(KEY_2)$时,才会发生冲突,这样就大大降低了冲突发生的概率。

理想情况下,选择恰当的H_{addr}和H_{key},可以产生$2^t\cdot 2^r$种不同的哈希结果组合,此时式(2)中的冲突概率可以降低到

$$P_{\text{conflict}}\approx m^2 2^{-r-1}=(m^2/n)2^{-r} \quad (3)$$

根据式(3)可以计算得到,当$m=2^{20}$,$n=2^{22}$,$r=64$ bit,P_{conflict}近似为10^{-14};当$r=96$ bit 时,P_{conflict}近似

为 3×10^{-24};当 $r = 128$ bit 时,$P_{conflict}$ 近似为 8×10^{-34}。

下面以基于取模运算的哈希函数为例对所需的 H_{addr} 和 H_{key} 的关系进行说明。假设 H_{addr} 采用模 a 运算,H_{key} 采用模 b 运算,则哈希地址的取值共有 a 种,分别为从 0 到 $a-1$,压缩的流标识的取值共有 b 种,分别为从 0 到 $b-1$,如果我们选择使得 a、b 互质,则模 a 和模 b 的取值共有 $a \cdot b$ 种,且依次地从 0 到 $a \cdot b - 1$ 对应不同的模 a 模 b 组合,此时便可以达到式(3)中的性能。

但在实际应用中,更常采用的是基于CRC的哈希函数,基于CRC的哈希函数是通过不同的生成多项式达到不同的映射关系,因此,如何选择哈希地址CRC多项式与压缩CRC地址多项式,是需要进一步研究的一个问题。此外在实现中,虽然CRC有简单的硬件电路可以方便的实现,但随着CRC哈希函数计算结果宽度的增加,相应硬件电路实现难度也有所增加,因此,在前面问题研究的基础上,可以方便地将长的CRC哈希函数拆成几个短的CRC哈希函数,如64 bit的CRC可以通过2个一定关系的32 bitCRC实现,如96 bit的CRC可以通过3个一定关系的32 bitCRC实现,使各个哈希函数计算结果的组合仍然具有采用单个长CRC哈希函数的效果,将更加有利于提高这种结构的实用性。

参考文献:

[1] BLAKE S, BLACK D, CARLSON M, et al. An architecture for differentiated services [R], RFC 2475, 1998.

[2] XIAO X, LI L M. Internet QoS: A big picture[J]. IEEE Network, 1999, 13(2): 8-18.

[3] BRADEN R, CLARK D, SHENKER S. Integrated services in the Internet architecture: an overview [R]. RFC 1633, 1994.

[4] BRADEN R, Ed., ZHANG L, et al. Resource reservation protocol (RSVP) version 1 functional specification[R]. RFC 2205, 1997.

[5] ALBERTENGO G, SISTO R. Parallel CRC generation[J]. IEEE Micro. 1990, 10(5): 63-71.

[6] SHIEH Ming-Der. A systematic approach for parallel CRC computations [J]. Journal of Information Science and Engineering, 1999, 17(3): 445-461.

(编辑:刘 勇)

Study of a hardware Hash list compression method and its performance analysis

ZHANG Yong, LEI Zhen-ming

(*Beijing University of Posts and Telecommunications, Beijing* 100876, *P. R. China*)

Abstract: In high speed and large volume flow of packet classification, hardware Hash table has the advantage of lower cost and better performance. In these applications, the flow ID should be stored in the hardware hash table. While in some applications, the flow ID can be very long, e.g. 300 bits, and it requires a much larger memory to hold these flow ID. At the same time, the Hash table look-up performance is degraded remarkably. In this paper, we proposed a hardware Hash list compression method. It can effectively reduce the flow ID length and Hash table memory size needed and improves the table look-up performance. The implementation of this method is very simple and it has much merit in practical applications.

Key words: hardware Hash list; compression; list lookup performance