多维过滤规则无冲突的高速分组分类算法

杜德超, 姚庆栋
(浙江大学信息与电子工程学系, 浙江杭州 310027)

摘 要: 为了有效地实现防火墙及 QoS 路由等功能, 路由器等网络元素必须能高速地对分组分类. 对一维分组分类, 已有很多成熟方案, 而多维算法由于实现复杂, 还没有有效的分类算法. 本文对无过滤规则无冲突的数据库进行了研究, 提出了基于元组空间多维分组分类算法: 元组空间矢量位映射算法. 对多维和二维分类在最不利情况下分别进行了性能分析, 指出与已有的方案相比, 在存贮空间、查找时间等性能上, 本文提出的算法是效率最佳的. 本文的算法不仅可以由软件实现, 也很容易由硬件实现.

关键词: 无冲突过滤规则; 分类; 元组空间; 位映射
中图分类号: TP393 文献标识码: A 文章编号: 0372-2112 (2002) 11-1676-05

High-Speed Packet Classification for Multi-Dimensional Conflict-Free Filters

DU De-chao, YAO Qing-dong
(*Department of Information Science & Electric Engineering, Zhejiang University, Hangzhou, Zhejiang 310027, China*)

Abstract: Routers must perform packet classification at high speeds to support advanced functions such as firewalls and QoS routing. While several efficient solutions are known for the one-dimensional IP lookup problem, the multi-dimensional packet classification has proved to be far more difficult. Existing filter schemes with fast lookup time do not scale to large filter database. Based on tuple space search, a packet classification algorithm called bitmap vector of tuple space for multi-dimensional conflict-free filters is presented in this paper. The result of the performance analysis in two-dimension and multi-dimension shows that the scheme provides better worse case bounds about time and space complexity than what have existed, so the algorithm is more scalable and faster. The novel approach can be easily implemented both from software and hardware which made this algorithm more practical to application.

Key words: conflict-free packet filters; classification; tuple space; bitmap

1 引言

分组的分类(又称查找)算法是快速转发分组的主要瓶颈[2]. 如何快速地对分组进行分类, 近年来受到广泛的重视.

传统的路由器仅基于输入分组的目的地址转发分组, 然而越来越多的应用要求分组的转发基于分组头中的多个域: 如基于源、目的地址; 源端的端口号; 协议类型; 目的地 URLs; 服务类型等. 因此, 就需要有基于分组头中多个域的多维分组分类算法. 本文对无过滤规则冲突的数据库进行了研究, 提出了基于元组空间的矢量位映射查找算法. 与目前已有的方案相比, 在时间复杂性和空间效率上, 本文提出的算法是综合性能最优的. 本文算法很容易由硬件实现, 从而可以线速处理输入分组.

2 分类算法的有关概念

若分组头中有 K 个域(或维)与过滤规则相关, 每个过滤规则 F 将由 K 个元素组成: $(F[1], F[2], \cdots, F[K])$, 这时, 称 F 为 K 维过滤规则. 其中 $F[i]\ (1\leqslant i\leqslant K)$ 用比特串前缀或区域(range)表示.

由 N 个过滤规则 F 构成的数据库(或规则集)定义为 $\boldsymbol{F}=(F_1, F_2, \cdots, F_N)^T$. 其中, 第 $j(1\leqslant j\leqslant N)$ 个规则的第 $i(1\leqslant i\leqslant K)$ 维元素表示为 $F_j[i]$. 即 $\boldsymbol{F}$ 可以表示成下列矩阵形式:

$$\boldsymbol{F}=\begin{bmatrix}F_1\\F_2\\\vdots\\F_N\end{bmatrix}=\begin{bmatrix}F_1[1] & F_1[2] & \cdots & F_1[K]\\F_2[1] & F_2[2] & \cdots & F_2[K]\\\vdots & \vdots & \vdots & \vdots\\F_N[1] & F_N[2] & \cdots & F_N[K]\end{bmatrix}$$

为方便起见, 用 F_i 表示 $\boldsymbol{F}$ 的某一行向量, 即某一过滤规则; 用 $\boldsymbol{F}[i]$ 表示 $\boldsymbol{F}$ 的列向量①.

由 K 个域组成的输入分组 $\boldsymbol{P}$ 表示为 $\boldsymbol{P}=(P[1], P[2], \cdots, P[K])$. 分组分类需要查找分组 $\boldsymbol{P}$ 的 K 个域, $\boldsymbol{P}$ 的每个域对应 $\boldsymbol{F}$ 的相应维. 若 $\boldsymbol{P}[i]$ 为分组 P 的第 $i(1\leqslant i\leqslant K)$ 个域, 则 $\boldsymbol{F}[i]$ 即为对应的 i 维. 如果分组 $\boldsymbol{P}$ 中每一个 $\boldsymbol{P}[i]$ 都与 $\boldsymbol{F}[i]$ 相匹配, 则称 $\boldsymbol{P}$ 匹配 $\boldsymbol{F}_i$. 例如, 具有源和目的地址的 IP 分组

收稿日期: 2001-11-05; 修回日期: 2002-03-30
基金项目: NSFC 项目基金(No. 60002003)
①本文中所有字母除特别说明外, 均默认前文的约定.

(128.112.234.2,128.122.34.51)与(128.112.*,128.122.*)匹配而与(128.112.*,128.132.*)不匹配.

设 F_1 和 F_2 同时与 P 匹配,如果 $F_1[i]$ 与 $P[i]$ 匹配前缀比 $F_2[i]$ 与 $P[i]$ 匹配前缀长,则说 $F_1[i]$ 是 $P[i]$ 的最佳匹配(BMP).

分组 P 的各个 $P[i]$ 可能最佳匹配不同 F 中的 $F[i]$,即同时有多个 F 与 P 匹配,这时就不能确定哪一个 F 是 P 的BMP.以2维 F 为例(见图1):图中 F_1=(01*,1011*)和 F_2=(0111*,10*)是 F 中的两个规则,若输入分组 P=(0111*,1011*),将出现 $F_1[2]$ 是 $P[2]$ 的BMP,而 $F_2[1]$ 是 $P[1]$ 的BMP.即分组与 F_1 和 F_2 的匹配存在冲突.解决冲突的方法主要有两种:

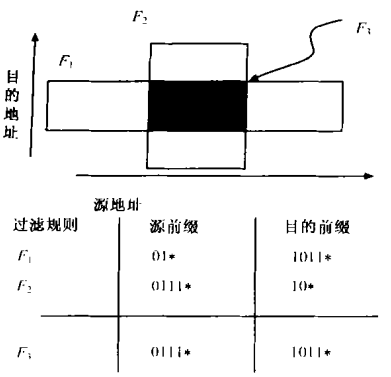

图1　两个冲突过滤规则

一种是按 F_i 在 F 中的存放顺序(也称优先级),上例中,若在 F 中 F_1 列在 F_2 前面,则取 F_1 为 P 的BMP;一种是Hari等[4]提出的在 F 中引入避免冲突的过滤规则,如在上例中,为解决 F_1 和 F_2 的冲突,引入了 F_3=(0111*,1011*),使 $P[1]$,$P[2]$分别为 $F_3[1]$ 和 $F_3[2]$的BMP,这样,F_3 就是 P 的BMP.本文中讨论的无冲突过滤规则的数据库,采用了Hari等提出的在 F 中引入避免冲突的过滤规则方法.因此,对 K 域无冲突过滤规则集 F,若 $F_i(\in F)$ 是分组 P 的最佳匹配,则对所有的域 $i(1\leqslant i\leqslant K)$,$F[i]$ 是 $P[i]$ 的最佳匹配(或最长匹配).

3　分类的相关算法

线性算法:对每个到达的分组,顺序地检查 F 中每个 F_i,直到找到一个与分组头中每个域都匹配的 F_i.它的算法简单,占用内存少 $O(N)$,Cashing技术常用于提高线性查找的性能.线性查找时间为 $O(N^K)$,扩展性不好.

Grid of Tries 和 Crossproducting:Grid of Tries 方法[7]是将多维分解为几个2维,再使用一个称作grid of tries的二维算法来解决多维问题.该算法扩展性不好,不容易扩展到多域.在该文中,作者还介绍了被称作Crossproducting的算法,在该方案中,在每个域中先进行最长匹配查找或区域查找,再将所得的结果组合(concatenate)起来构成crossproduct,据此映射到最佳匹配过滤规则.该算法的查找速度快,但需要占用 $O(N^K)$ 空间,容易导致内存爆炸,且对查找时间没有明确的保证.

RFC算法: RFC(Recursive flow Classification)算法是Gupta等[6]提出,Crossproducting算法可视作它的特例.基本思想是分几个阶段(phase),在每个阶段,执行最佳匹配,将一个较大的集合Crossproducting到一个较小的集合上.在平均意义上,该算法极大地减少内存,但最坏的情况下,与Crossproducting算法一样,需要空间为 $O(N^K)$.

元组空间查找:(详见下文).

基于前缀长度的折半查找:对一维算法,Waldvogel等[2]提出了基于目的地址前缀长度的折半查找算法.当前缀长度是 W 时,其查找时间是 $O(\log W)$.在本文算法中,对每一维的最佳匹配采用了该算法.

硬件方案:充分利用硬件并行运行的特点,可极大地提高查找性能.特别是TCAM(Ternary Content Addressable Memroies)技术[8]的出现,可以很有效地利用硬件实现过滤查找.但CAMs的字宽是一定的,缺少灵活性,不能适应过滤规则的变化.

另一种有趣的硬件实现方案是Lakshman等[1]提出的Bit-Parallelism方法,查找速度依赖于存贮器的宽度.该算法对每一维都要读入 N 比特的位映射,需要占用空间为 $O(N^2)$,因此扩展性不好,只适用于中小规模的应用.

4　元组空间(Tuple Space)算法

本文的算法基于元组空间,在此对Srinivasan等[3]提出的元组空间算法作一介绍.

它是由Waldvogel等[2]提出的一维算法引入的.基本思想是虽然数据库中规则很多,但不同前缀长度的数量却是有限的.因此,各个域中前缀长度的组合是很少的.这样,就可以按前缀长度将前缀分组,每个组中前缀长度相同,这时就可以用前缀的比特串作为hash的关键字来查找数据库.Waldvogel等人[2]据此提出了折半查找算法,使时间复杂性减到 $\log(W)$,其中 W 为前缀长度.由于算法不依赖于数据库规模,所以扩展性很好.

将该思想扩大到多维,即产生了元组空间的概念[3].对多维过滤规则中每一维都用前缀的长度来表示,这样得到的结果的集合就是元组空间,用 T 表示,对应于规则数据库 F 有 $T=(T[1],T[2],\cdots,T[K])$.

IP分组的有些规则,如端口号,常用区间(range)的形式给出.为便于元组表达,必须将区间化为前缀表示.理论上,W 比特的区间至多可由 $2W-2$ 个前缀表示.Srinivasan等[3]提出了RangeID的概念,很好地解决了区间转化为前缀表示问题.

元组空间的查找算法是:在 F_i 中每一维 $F[i]$ 取所需的 $T[i]$ 比特,将这些比特组合(catenate)起来,构成该 F_i 的hash关键字,再用该关键字查hash表.

一般地,元组数比过滤规则数少很多.因此即使是在元组空间中采用线性查找,其性能也有很大提高.

通过对Mae-East[10]路由表中前缀数据库的研究,Srinivasan等人提出了Tuple Space Pruning算法.该算法在平均意义上速度很快,但它的性能与具体的数据库有关,最不利的情况下查找时间的复杂性是 $O(W^K)$.因此,在实际应用中,需要有更有效的分类算法.

5　元组空间矢量位映射算法与实现

对过滤规则无冲突的数据库 F,存在如下定理:

定理　设 $F_{bmp}\in F$ 是 P 的BMP,F_{bmp} 对应的元组是 $T_{bmp}\in T$,则在所有与 P 匹配的 F_i 中,必有

$$\sum_{i=1}^{K}T_{bmp}[i]=\max_{i=1}^{M}\left[\sum_{j=1}^{K}T_i[j]\right]$$

其中 $T_i \in T$ 是所有与 P 匹配的 F_i 对应的元组, M 是匹配的个数. (证略).

因此 P 与无冲突的 F 匹配的结果有三种:

(1) F 中没有规则与 P 相匹配;

(2) F 中有且只有一个规则与 P 相匹配;

(3) F 中有多个规则与 P 相匹配, 但只有一个与 P 最佳匹配, 且满足以上定理.

因此, 若将 F 对应的规则元组空间 T 按 $\sum_{i=1}^{K} T[i]$ 由大到小的顺序排列, 查找时, 第一个匹配的必然是 BMP. 因此, 简单的线性查找就可实现多维的最佳匹配. 实现算法用伪码表示为:

```
Function LinearSearch( P)
将 BMP 初始化为 nil;
i:= N;
While( BMP= nil) and( i> = 0) do
    For k= 1 to K do
    分别从 P[ k] 中取 Ti[ k] 个比特构成关键字 KEY;
    end for
    BMP:= Search( KEY, HashTable( Ti) );
    i:= i- 1;
Endwhile
```

其中 N 为 T 中元组数. 该算法的时间复杂性为 $O(N)$, 占用空间复杂性为 $O(N')$ (N' 是 F 中规则数).

该算法在查找时间上不理想, 虽然元组数比原有数据库的规则数少很多.

5.1 Pruned tuple space 算法的修正

为了改进元组空间中的搜索算法, Srinivasan 等[3]提出了 Pruned tuple space 算法. 它的原理是: 为数据库 F 中每一个前缀 $F_i[k]$ 建立一 Tuple list, 这是一个所有与 $F_i[k]$ 匹配的元组列表. 当查找时, 每一维 k 先独立地查找到最佳匹配的 $F[k]$, 再查找它的 Tuple list, 若对所有的 k ($1 \leqslant k \leqslant K$), 各 $F[k]$ 的 Tuple list 中对应有相同的元组, 则该元组就是所查找的. 该算法实际上只能是在 F_i 无冲突的条件下成立, 即对所有的 k, 各 $F[k]$ 的 Tuple list 中不能有两个或两个以上相同元组. 当 F_i 有冲突时, 由于元组隐藏了 F_i 的头信息, 基于优先级的冲突解决方法在该算法中无法实现, 只能采用引入避免冲突过滤规则的方法[4]. 根据本文的定理, 对该 Pruned tuple space 算法的修正如下: 对所有的 k, 若各 $F[k]$ 的 tuple list 中有多个相同的元组, 则 $\sum_{i=1}^{K} T[i]$ 最大的元组是所需的元组.

为说明该算法, 还是以本文前面的图 1 为例, 各 F_i 前缀对应的 Tuple list 如表 1 所示. 表中加 * 号的元组表示引入避免冲突过滤规则 F_3 时增加的元组. 当输入分组 P = (0111, 1001) 时, $P[1]$ = 0111 与 $F_1[1]$、$F_2[1]$ 及 $F_3[1]$ 的最长匹配是 0111, 对应的 Tuple list 1= [(2, 4), (4, 2), (4, 4)]; $P[2]$ = 1011 与 $F_1[2]$、$F_2[2]$ 及 $F_3[2]$ 的最长匹配是 10, 对应的 Tuple list 2 = [(4, 2)]. 由于 Tuple list1 与 Tuple list2 有共同的唯一元组 (4, 2), 所以它就是要查找的最佳元组. 当输入分组 P = (0111, 1011) 时, 若没有引入 F_3, 得到的 Tuple list1= [(2, 4), (4, 2)] 和 Tuple list2= [(4, 2)]. 两个 Tuple list 中有二个相同的元组, 存在冲突, 无法取舍. 若引入 F_3, 对同样的分组 P, 得到 Tuple list1= [(2, 4), (4, 2), (4, 4)] 和 Tuple list2= [(2, 4), (4, 2), (4, 4)]. 两个 Tuple list 中有三个相同的元组, 这时, 按修正算法, 就可取 $\sum_{i=1}^{K} T[i]$ 为最大值的元组 (4, 4) 为最佳元组.

表 1 Tuple list 示例

前 缀	$F_1[1]$ = 01*	$F_1[2]$ = 1011*
Tuple list	(2, 4)	(2, 4), (4, 2), (4, 4)*
前 缀	$F_2[1]$ = 0111*	$F_2[2]$ = 10*
Tuple list	(2, 4), (4, 2), (4, 4)*	(4, 2)
前 缀	$F_3[1]$ = 0111*	$F_3[2]$ = 1011*
Tuple list	(2, 4), (4, 2), (4, 4)	(2, 4), (4, 2), (4, 4)

在最不利的情况下, 该算法的时间复杂性仍然是 $O(W^K)$. Srinivasan 等人对 Mae-East[10] 路由器的路由地址前缀的调查发现, Tuple list 中元组数一般不超过 6 个, 因此线性查找各个 Tuple list 中相同元组的速度非常快. 据此, 他们认为在实用中该算法很有效.

5.2 元组空间矢量位映射算法

为了提高在最不利情况下的查找时间, 本文提出一种元组空间矢量位映射算法. 它是对在 Tuple list 中线性查找相同元组的改进, 以期望提高最不利时的查找速度. 设元组空间 T 按 $\sum_{i=1}^{K} T[i]$ 由大到小的顺序排列, 对于 $\sum_{i=1}^{K} T[i]$ 相等的 T_i、T_j, 可任意顺序. 查找前, 为数据库 F 中每一个前缀 $F_i[k]$ 建立一个元组空间 T 的位映射矢量 $R_{i,k}$. 其中 $R_{i,k}[j]$ 是 $T_j \in T$ 的位映射. T 中每个元组 T_j 映射到 $R_{i,k}$ 位矢量中对应的每一比特 $R_{i,k}[j]$. 映射原则是: 当 $F_i[k]$ 的 Tuple list 中的元组与 T 中元组 T_j 相同时, 矢量 $R_{i,k}$ 对应的 $R_{i,k}[j]$ 位置为 1, 其他位清零. 这样, F 中每个前缀 $F_i[k]$ 都对应一个位矢量 $R_{i,k}$. 查找时, 对每个维 k ($1 \leqslant k \leqslant K$), 先使 $P[k]$ 在 $F[k]$ 中找到最佳匹配的 $F_{bmp}[k]$, 根据 $F_{bmp}[k]$ 找到对应的位映射矢量 $R_{bmp,k}[j]$. 再按 j ($1 \leqslant j \leqslant N$) 由小到大的顺序, 对各个维 k ($1 \leqslant k \leqslant K$) 作逻辑"与"运算, 若对某个 $j = J$, 有 $\prod_{k=1}^{K} R_{bmp,k}[J] = 1$, 则表示对应 T_J 是 P 的 BMP 元组.

仍以本文图 1 为例. F 中每个前缀对应的 Tuple list 如表 1 所示, 按所需顺序排好的元组 T 如图 2 所示. 每个 $F_i[k]$ 的位映射矢量 $R_{i,k}$ 见表 2. 当输入分组 P = (0111, 1001) 时, $P[1]$ = 0111 与 $F_1[1]$、$F_2[1]$ 及 $F_3[1]$ 的最佳匹配是 0111, 对应的位矢量是 $R_{bmp,1} = [1, 1, 1]^T$; $P[2]$ = 1011 与 $F_1[2]$、$F_2[2]$ 及 $F_3[2]$ 的最长匹配是 10, 对应的位矢量是 $R_{bmp,2} = [0, 1, 0]^T$. 将所得的两个位矢量对应行逻辑"与", 即发现当 j = 2 时有 $\prod_{k=1}^{2} R_{bmp,k}[2] = 1$. j = 2 在 T 中对应的元组是 T_2 = (4, 2). 因此它就是所要查找的元组. 当输入分组 P = (0111, 1011) 时, 在两维空间中同样可得两个位矢量 $R_{bmp,1} = [1, 1, 1]^T$ 和 $R_{bmp,2} = [1, 1, 1]^T$.

$$T = \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} = \begin{bmatrix} (4, 4) \\ (4, 2) \\ (2, 4) \end{bmatrix}$$

图 2 元组空间示例

两个位矢量对应位相"与", 发现当 $j=1$ 时, 结果为 1. 因此 T 中 $T_1=(4,4)$ 即是查找的元组.

表 2　算法示例

前　缀	$F_1[1]=01$	$F_1[2]=1011$
位矢量	$R_{11}=\begin{bmatrix}0\\0\\1\end{bmatrix}$	$R_{12}=\begin{bmatrix}1\\1\\1\end{bmatrix}$
前　缀	$F_2[1]=0111$	$F_2[2]=10$
位矢量	$R_{21}=\begin{bmatrix}1\\1\\1\end{bmatrix}$	$R_{22}=\begin{bmatrix}0\\1\\0\end{bmatrix}$
前　缀	$F_3[1]=0111$	$F_3[2]=1011$
位矢量	$R_{31}=\begin{bmatrix}1\\1\\1\end{bmatrix}$	$R_{32}=\begin{bmatrix}1\\1\\1\end{bmatrix}$

以上为了叙述方便, 给每个 $F_i[k]$ 创建一个位矢量. 实际上, 每一维最佳匹配的结果只有一个, 对应一个位矢量. 因此每一维中具有相同前缀的 $F_i[k]$、$F_j[k]$ $(i\neq j)$ 只需对应一个位矢量 R_k. 即 R_k 仅与前缀有关, 而与 F 中位置 i $(1\leqslant i\leqslant N')$ 无关. 这样可以减少存贮空间.

每一维最佳匹配是一维算法, 采用 Waldvogel 等[2] 的折半查找, 算法复杂性是 $O(\log W)$.

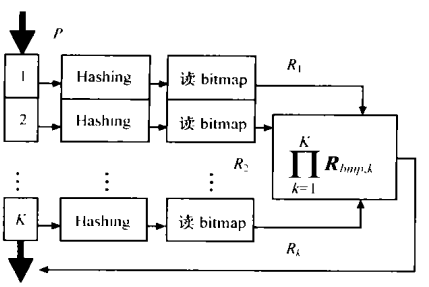

图 3　硬件实现示意图

由于一维的折半查找实际上是查 hash 表, 而位矢量的按位"与"运算只需"与门", 因此该算法可以非常简单地用硬件实现(如 FPGA, CPLD). 每一维都独立地并行查找、寻找位矢量, 可以充分利用硬件并行执行的特点加速查找.

6　性能比较

只有最坏情况下的性能得到保证, 才不会出现在分组分类处理之前的排队现象(这时, 一些重要的分组就可能被丢弃). 下面将本文算法与常见的几种多维和二维算法在最不利情况下的性能作一比较.

6.1　多维算法性能比较

表 3 中列出了几种多维算法在最不利的情况下的性能. 表中 W 是前缀长度, N' 是 F 中规则数, N 是 T 中元组数. 存在如下关系: $W<N'<N$. 可以看出:

(1) 由于 N 远小于 N', 所以 $O(N'^2)\leqslant O(NN')\leqslant O(N'\log(N'))$. 本文算法空间复杂性介于 Lakshman 等[1] 提出的两种算法之间. 由于 $N\leqslant W^K$, 当 N' 很大时, N 并不随 N' 增大而增大, 因此不会出现象 $O(N'^2)$ 那样的内存爆炸问题. 本文算法的时间复杂性比 Lakshman 等的算法改善很多.

(2) Tuple Space 算法由于查找时间为 $O(W^K)$, 所以不适于多维查找, 只能作为一种算法提出. Pruned tuple space 算法是基于对数据库的认识, 认为同时与同一前缀匹配的不同前缀个数很少. 与以上两种 tuple space 算法复杂性相比, 本文的算法复杂性是 $O(\log W)$ 加上 N 比特的"与"运算. 即使不采用硬件实现, 由于本文的 N 比特实际的存贮器访问次数为 $N/L*8<W^K/(L*8)<W^K$, 其中 L 为 cache 的 slot 字节数, 所以远小于 Srinivasn 等[3] 的时间复杂性.

(3) Crossproducting 算法具有极好的查找速度, 但由于其占用空间为 $O(N'^K)$, 使得该算法实际上不可行.

表 3　多维算法性能比较

算　法		时间复杂性	空间复杂性
Lakshman 等[1]		$O(\log(N')+N')$	$O(N'^2)$
		$O(\log(N')+N'\log(N')+N')$	$O(N'\log(N'))$
本文算法(位矢量)		$O(\log(W)+N)$	$O(NN')$
Srinivasn 等[3]	Tuple space	$O(W^K)$	$O(N)'$
	Pruned tuple space	$O(W^K)$	$O(NN')$
Crossproducting		$O(\log(W))$	$O(N'^K)$

6.2　二维算法性能比较

基于源地址/目的地址的二维算法在实际中应用很多[5], 在 VPN、多播等应用中都会用到, 同时由于二维具有一定的特殊性, 单独研究它的算法很多. 在此将几种常见的二维算法作一比较(见表 4).

表 4　二维算法性能比较

算　法	时间复杂性	空间复杂性
Rectangle search[3]	$O(W)$	$O(N'W)$
Grid of tries[7]	$O(W)$	$O(N'W)$
Two Dimensional Conflict-Free[5]	$O(\log^2(W))$	$O(N'\log^2(W))$
本文的算法	$O(\log(W)+N)$	$O(NN')$

表 5　二维算法最多需要访问存贮器次数

算　法	存贮器访问次数
Rectangle search[3]	$2W-1=63$
Grid of tries[7]	$W=32$
Two Dimensional Conflict Free[5]	$\log^2(W)=25$
本文的算法	$2(\log(W)+N)=18$

在软件实现中, 算法执行速度主要与程序访问存贮器的次数多少有关. 为了更清楚地比较各算法性能, 以每种算法在最不利情况下, 需要读取存贮器的次数为参数, 来比较各算法的时间复杂性. 设 $W=32$(IPV4), 这样二维空间中最多的元组数 $N=W^2=1024$. 设 cache 的 slot 为 32 个字节. 则 $N=1024$ 比特需要读存贮器的次数为 $1024/(32*8)=4$ 次. 因此各二维

算法需要的数据存贮器访问次数如表 5 所示.

在空间占用上, 由于 $N \leqslant W^2 = 1024$, 所以本文算法需要中用的空间 $O(NN') = O(N')$.

可以看出, 在二维空间上本文算法即使采用软件实现, 代价也是很小的.

本文是在无过滤规则冲突的数据库的基础上研究的. 对于已有的数据库, 可以按 Hari[4] 等介绍的方法检查是否存在冲突并化为无冲突的数据库. 由于存在规则冲突的数据库可能存在安全隐患等问题[4], 因此设计良好的数据库都是尽量避免冲突的. Gupta 等[6] 报导了大小为 1734 的数据库只有 2581 个冲突. 而理论上最差情况下可达 10^{13}. 因此一般来说, 无冲突数据库与原有数据库规模相当.

7 结论

多域分组分类算法近年来得到了广泛的研究. 我们对无过滤规则冲突的数据库进行了研究. 提出了基于元组空间矢量位映射算法, 指出并修正了 Srinivasn 等[3] 提出的 Pruned turple space 算法存在的问题. 与目前流行的几种分类算法相比较表明, 由于将多维查找简化为多个并行的一维查找, 因此具有算法简单、查找速度快的特点; 在占用空间上, 由于本文基于元组空间, 所以具有很好的扩展性. 本文算法的另一个突出特点是可以很容易用硬件实现, 因此非常适合在高速分组分类中应用.

参考文献:

[1] T V Lakshman, D Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching[A]. Proc. of ACM Sigcomm[C]. Vancouver, Canada: 1998. 101-202.

[2] M Waldvogel, G Varghese, J Turner, B Plattner. Scalable high speed IP routing lookups[A]. Proc. of Sigcomm[C]. Cannes, France, 1997. 25-35.

[3] V Srinivasan, S Suri, G Varghese. Packet classification using turple space search[A]. Proc. of Sigcomm[C]. Cambridge, Massachusetts, 1999. 135-1466.

[4] A Hari, S Suri, G Parulkar. Detecting and resolving packet filter conflicts[J]. Porc. of IEEE INFOCOMM, 2000. 1203-1213.

[5] P Warkhede, S Suri, G Varghese. Fast packet classification for two-dimensional conflict-freee filters[J]. IEEE INFOCOM, 2001. 1434-1443.

[6] P Gupta, N McKeown. Packet classification on multiple fields[J]. ACM Computer Review, 1999, 29(4): 146-160.

[7] V Srinivasan, G Varghese, S Suri, M Waldvogel. Fast and scalable layer four switching[A]. Proc. ACM Sigcomm[C]. Vancouver, Canada, 1998. 203-214.

[8] Anthony J McAuley, Paul Francis. Fast routing table lookup using CAMs[J]. IEEE INFOCOM, 1993, 3: 1382-1391.

[9] P Gupta, N McKeown. Algorithms for packet classification[J]. IEEE Network, 2001, 4: 24-32.

[10] http://nic.merit.edu/impa[DB/OL]. 2001, 8.

作者简介:

杜德超 男, 1965 年生于安徽无为, 浙江大学信电系博士研究生, 研究方向: 网络流量工程与 QoS 等.

姚庆栋 男, 1932 年生于浙江瑞安, 浙江大学信电系教授, 博士生导师, 研究方向: 数字通信、信息处理、VLSI 等.