一种基于重叠位图的路由查找算法

刘 斌 张楚文
(清华大学计算机系 北京 100084)

摘 要 路由查找是路由器的核心功能之一,可分为基于硬件和基于软件的查找算法两大类.前者使用专用的可并行硬件实现高速的查找性能,比如 FPGA 算法、GPU 算法和 TCAM 算法.后者可以部署在通用 CPU 上,具有更高的灵活性、更低的功耗和成本优势,并且可以利用 CPU 的 Cache 实现快速查找.因此,基于软件的路由查找算法已成为软件定义网络和网络功能虚拟化中的关键技术之一.尽管软件查找算法具有很大的优势,它也面临着许多新的挑战.首先,当今骨干网路由器的路由表项数目已达到 600 K,并且每年保持大约 15%的增长率,给路由查找和存储带来了巨大压力.同时,路由表更新速度也逐年稳定增长,并且峰值更新速率已超过 10 K/s,这就要求路由查找算法具有高速的更新性能.基础的树结构软件查找算法能够支持快速更新,但是过多的访存次数导致查找速度较低,而且其存储开销已经超过 16 MB,远远高于一般路由器中 CPU 的 Cache 大小,进一步影响了查找速度.以 Lulea 算法为代表的传统位图压缩方法虽然降低了数据结构的存储开销,但是会导致更新困难,复杂而低效的更新操作也会在一定程度上影响查找性能.本文提出了一种基于重叠位图压缩的软件路由查找算法,它通过层次遍历构造重叠式位图结构,比具有高压缩率的 Lulea 算法占用更小的存储空间(提高 Cache 的命中率,从而进一步提高查找速度).而且,本算法使用位图分割和多种更新优化技术实现快速的增量更新.实验结果表明本算法能够把包含 600 K 条前缀的路由表压缩到 2.3 MB,平均比 Lulea 算法减少 26%的存储空间,只有树结构的 1/8 左右.而且本算法具有良好的拓展性,从 2008 年的 5.06 字节/前缀降低到 2016 年的 3.94 字节/前缀.实际流量下,本算法平均查找速度达到 111.41 M/s,是 Lulea 算法查找速度的 2.5 倍.同时,在保证 10～100 K/s 的更新速率前提下,实现 90～100 M/s 的查找速度.

关键词 路由查找;位图压缩;增量更新;层次遍历;重叠位图
中图法分类号 TN915 **DOI 号** 10.11897/SP.J.1016.2018.02106

An Overlay Bitmap-based Routing Lookup Scheme

LIU Bin ZHANG Chu-Wen
(*Department of Computer Science, Tsinghua University, Beijing* 100084)

Abstract Routing lookup is one of the key functions of network routers, which can be divided into hardware-based and software-based algorithm. The former, such as FPGA-based, TCAM-based and GPU-based algorithms, uses dedicated and parallelizable hardware to achieve high lookup speed. The latter, which can be deployed on a commodity CPU, is more flexible, more energy-saving and cheaper, and the lookup performance can also be improved by exploiting the CPU cache. Therefore, it has become one of the key technologies in SDN (software defined network) and NFV (network functional virtualization). While software-based routing lookup algorithm is attractive, it faces some new challenges. Nowadays, Backbone route tables maintain an annual growth rate of around 15% and the number of route table prefixes has reached up to 600 K. The drastic expansion of the route table brings enormous pressure on lookup speed and

收稿日期:2017-02-22;在线出版日期:2017-11-29.本课题得到国家自然科学基金(61432009,61373143,61602271)、中国博士后面上基金(016M591182)、教育部博士学科点专项科研基金(0130002110084)资助.**刘 斌**,男,1964 年生,博士,教授,博士生导师,中国计算机学会(CCF)专业会员,主要研究领域为高性能交换机/路由器、网络处理器、命名数据网络、软件定义网络等.E-mail:liub@mail.tsinghua.edu.cn.**张楚文**(通信作者),男,1992 年生,博士研究生,主要研究方向为高性能交换机/路由器、命名数据网络、车联网等.E-mail:chuwen1992@gmail.com.

storage space. Meanwhile, the update rate grows steadily and reaches a peak update rate at 10 K updates per second, which requires the advent of algorithms with high-speed update performance. The binary trie is the basic software-based routing lookup algorithm. Although it can support fast updates, its lookup speed is slow because of too many times of memory access. And its storage space has exceeded 16 MB to store a route table of 600 K prefixes, much higher than the CPU cache size in the line-card of a general router, which will decrease the slow lookup speed further. The traditional bitmap-based routing lookup schemes represented by Lulea algorithm can compress storage redundancy and build small data structure to fit into caches. However, they usually aggravate the incremental update and cannot meet the requirement of update speed in real networks. Their inefficient update will also interrupt the normal lookup and thus decrease the lookup throughput. In this paper, we propose an overlay bitmap-based software routing lookup scheme. It constructs overlay bitmap structures through hierarchical traversal, which can eliminate the horizontal redundancy in traditional bitmap and achieve even smaller storage than the high compressed bitmap-based scheme Lulea (increasing the cache hit rate and thus boosting lookup speed further). In addition, it performs fast incremental update using bitmap segmentation and a variety of update optimization techniques. Experimental results show that our scheme can compress a route table of 600 K prefixes to a 2.3 MB data structure, reducing 26% storage space than Lulea algorithm on average, which is only about 1/8 of the binary trie. Besides, the storage space of our scheme is more scalable, decreasing from 5.06 bytes per prefix in 2008 to 3.94 bytes per prefix in 2016. Under the real network traffic, its average lookup speed can reach up to 111.41 MSPS (million searches per second), about 2.5 times as fast as that of Lulea algorithm, benefiting from the smaller memory footprint and higher cache hit rate. Tests on update performance show that it can sustain up to 1.82 million updates per second. And it can achieve a fast lookup up to 90—100 MSPS with the ability to handle 10—100 K updates per second at the same time.

Keywords　routing lookup; bitmap compression; incremental update; level traversal; overlay bitmap

1　引　言

路由查找是路由器根据路由表为网络流量选择路径的过程.路由查找算法的功能包括将路由表存储为高效的数据结构,并实现查找和更新.路由查找算法的性能在很大程度上决定了路由器的性能,故其研究意义重大.

路由查找算法可以分为基于硬件的查找算法和基于软件的查找算法两大类.前者利用硬件高速、专用和并行的特点,查找速度快,广泛应用于骨干路由器,如 TCAM 算法[1-3]、FPGA 算法[4]和 GPU 算法[5].但是基于硬件的查找算法具有功耗高、灵活度低的缺点,尤其随着软件定义网络(Software Defined Network,SDN)和网络功能虚拟化(Network Function Virtualization,NFV)的发展,要求路由算法具有灵活性和通用性.因此,运行在通用处理器上的、具有更高灵活性的软件路由查找算法受到学术界越来越多的关注.

软件路由查找算法大都源于基础的 Trie 树结构,其存储开销大且查找深度不可预测,当查找最深的层次时速度将大幅下降.层次压缩技术[6]和路径压缩技术[7]通过改变树的结构,减小了搜索深度,却增大了更新的难度.Bloom Filter[8]技术首先通过快速定位查找匹配的树层,然后在此层上做精确查找,但是执行 Bloom Filter 的计算开销很大,同时还存在 False-Positive 的问题.传统的位图压缩技术[9-10]能减少存储冗余,但往往使更新复杂化.近年来,许多基于 Tire 树的高效软件查找算法被提出.熵压缩算法[11]能够实现基于信息熵理论的最大压缩率,缺点是难以支持高速查找与更新.SAIL 算法[12]通过分割路由查找的过程,显著地减少访存次数,提升了查找速度.但当网络流量局部性较差时,较低的缓存命中会影响其性能.Poptrie[13]算法拓展了多径 Trie

树结构，利用位图和多种技术实现快速查找，但其牺牲了部分存储空间，增量更新过程也较为复杂.

路由查找算法的评价指标有存储开销、查找速度和更新速度等.此外，路由查找算法还应随着路由表增大具有良好的可扩展性.路由查找算法需要兼顾存储、查找和更新，但是广泛研究的路由查找算法在权衡各方面的性能时，往往以牺牲更新性能来换取存储和查找性能.基于位图压缩的 Lulea[9] 算法是典型的例子.Lulea 算法存储开销小，查找速度快，但是由于位图重建复杂，导致难以实现增量更新.LOOP 算法[14] 针对虚拟路由器中存在的多个虚拟路由表情况，对 Lulea 算法进行了拓展，采用了层次遍历构造位图，从而实现快速查找，但是没有考虑到更新的局部性，更新性能有待提高.

随着网络动态性的增强，路由表的更新速度正在逐步增大.根据路由报告①，目前路由表的最大更新速率已超过 10K 更新/s，该更新速率还有继续增大的趋势[15].因此，路由查找算法的更新性能愈发重要.在树结构算法、Lulea 算法和 LOOP 算法的基础上，本文提出一种新型的基于重叠位图压缩的软件路由查找算法.该算法避免采用传统的叶推送技术[16]，而是通过对树结构的层次遍历，聚合位图中存储相同信息的相邻比特位，从而进一步压缩存储.此外，该算法还利用位图分割减小更新粒度，克服位图压缩引发的更新困难.最后，根据更新的局部性，我们还采用了多种优化方法来进一步提高更新速度.

实验结果表明本文提出的算法比目前位图压缩率较高的 Lulea 算法还要节省 20%～30%的存储空间.本算法利用位图压缩技术构建出足够小的数据结构以放入 CPU 的 Cache 中，从而提高查找速度，比 Lulea 算法提升超过 150%.多种更新优化技术使本算法能够达到 1.8M/s 的平均更新速度.本算法除了具有良好的储存、查找和更新性能外，还具有良好的可扩展性，这是因为位图压缩技术使得存储开销随路由表的增长而增加缓慢.

本文首先在第 2 节描述背景知识，然后在第 3 节详解算法的三个关键技术：层次遍历、位图分割和更新特性优化.第 4 节介绍实现、查找与更新过程，接着在第 5 节进行实验和性能评价，最后在第 6 节做总结.

2 背景介绍

2.1 Trie 树结构算法

典型的路由表如表 1 所示，每一条表项由前缀/前缀长度和转发端口组成.如表项[010 * /3，P2]表示前缀是 010 *，前缀长度是 3，转发端口是 P2，所有 IP 地址以 010 开头的数据包都能与这条表项匹配.路由查找要求满足最长前缀匹配(Longest Prefix Match，LPM)原则[16]，即当目的地址匹配多条前缀时，最长的前缀决定其转发端口.

表 1 典型路由表

前缀/长度	端口
* /0	P1
010 * /3	P2
0111 * /4	P1
1110 * /4	P1
01100 * /5	P3
11000 * /5	P3

Trie 树结构是一种传统的表示路由表的方式.树结构的构造是遍历整个路由表，为每条路由表项生成前缀结点的过程.从树根开始，逐位遍历前缀二进制串：若是 0，则转到左子树；若是 1，则转向右子树.前缀长度表示遍历的深度，遍历结束的结点就是前缀结点，它存储转发端口.图 1(a)是根据表 1 的路由表构造的树结构(只显示到第 4 层).使用树结构进行路由查找很简单：逐位遍历目的地址二进制串，0 表示往左子树查找，1 表示往右子树查找，目的地址匹配所有经过的前缀结点.如图 1(a)所示，地址(01001001)匹配最上面的结点 P1 和下面的结点 P2，而 P2 才是最终的转发端口.为了确保最长前缀匹配，许多路由查找算法采用叶推送技术将所有的内部结点的端口信息压到叶子结点.这样，一旦遇到叶子结点就能返回最终的转发端口.图 1(b)是图 1(a)经过叶推送处理后的树结构.

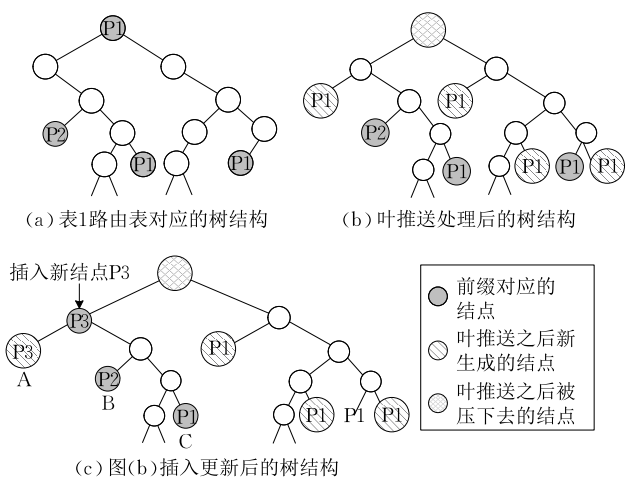

图 1 树结构及叶推送处理后的树结构

① BGP Routing Table Analysis Reports (AS65000)，http://bgp.potaroo.net，2017.2.18

叶推送技术有效避免路由查找的回溯，但是引发两个问题：(1) 叶推送导致额外的叶子结点产生，存储开销增大；(2) 为了正确地处理更新，所有的树结点需要额外的信息来指示此结点是表示前缀结点，还是叶推送处理后产生的新结点. 如图 1(c)所示，在叶推送处理的树中插入[0 * /1, P3]，叶推送产生的结点 A 就需要将端口信息由 P1 改为 P3，但是其后代中的前缀叶子结点 B 和 C 就不需要改变. 显然，更新前缀越短，叶推送导致的树结构变化越大，更新就会越复杂.

2.2　Lulea 及 LOOP 算法

Lulea 算法是对 Trie 树结构的位图式压缩，其采用水平切割、位图结构和多级索引表来压缩存储空间、提高查找速度.

考虑到前缀结点的分布特点和存储空间的开销，Lulea 算法把 32 层的 IPv4 树结构按照[16-8-8]的切割方式分成 3 个 Layer. 如图 2 所示，Layer 1 覆盖第 1 层到第 16 层，Layer 2 覆盖第 17 层到第 24 层，Layer 3 覆盖第 25 层到 32 层. “树分支”(Chunk)是夹在两切割之间的子树分支. 通过切割，整棵树被组织成许多树分支.

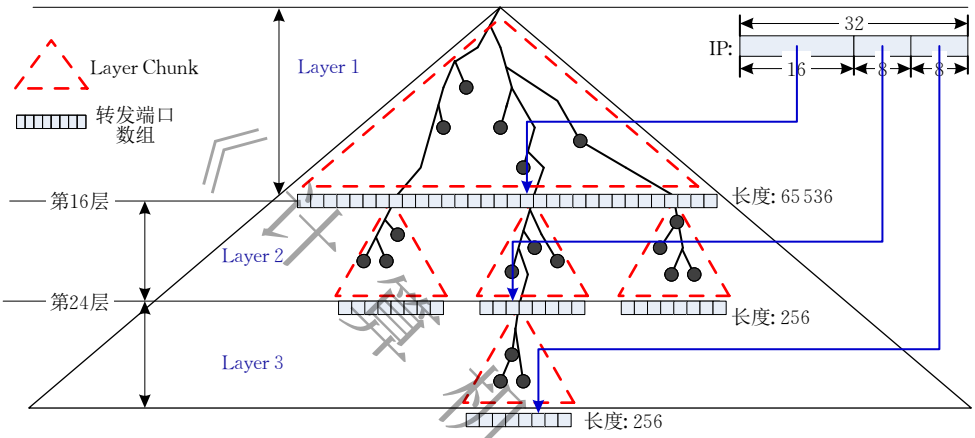

图 2　[16-8-8]切割方式

位图是一种高效的数据结构，其中每一位只有 0 和 1 两种取值. Lulea 算法首先利用叶推送把结点信息保存在叶子结点，然后通过遍历所有叶子结点构造对应的位图. Lulea 算法使用离线生成的多级索引表来实现快速查找. 此外，Lulea 算法还使用对稀疏数组进行特殊处理等方法来进一步减少存储开销.

虽然 Lulea 算法具有很高的压缩率，但仍存在一些问题：(1) 因为数据结构过于紧密，每当遇到增加或删除结点的更新时，Lulea 算法都需要重新生成整个路由表，这种做法难以应对现实网络中快速更新的需求；(2) Lulea 沿用了叶推送技术，由上文分析可知，这种技术增加了快速更新的复杂度；(3) 由于 Lulea 算法要求每一个 Genuine 结点(叶子结点“投影”到最底层后最左边的结点)必须为 1，可能出现连续的查找表项存储相同的内容，因此还存在进一步优化的空间. 而 LOOP 算法是对 Lulea 算法的多表拓展，其采用层次遍历构造位图，相比 Lulea 算法能进一步减少多表的存储空间. 但其没有考虑到更新的局部性，更新性能有待改进.

3　算法关键技术

通过对 Trie 树结构算法、Lulea 算法和 LOOP 算法的优缺点的分析，本文提出了一种基于重叠位图压缩的路由查找算法，采用：(1) 层次遍历和重叠式位图来减少存储开销；(2) 位图分割来支持快速更新需求；(3) 多种局部性更新优化算法来进一步提高更新速度. 实验结果显示，这三个关键技术显著地提高了更新速度，并且进一步减少了存储开销.

3.1　层次遍历和重叠式位图

本算法直接对原始 Trie 树结构采用层次遍历生成端口转发数组，然后扫描端口转发数组生成重叠式位图. 为了比较本算法和 Lulea 算法生成位图过程的区别，我们以表 1 所示路由表为例，分别阐述本算法和 Lulea 算法原理. 示例中，我们把 Layer 1 设置为 1 到 4 层，那么 Layer 1 最底层位图长度就是 $2^4=16$ 位. 因为表 1 路由表中的最后两项在 Layer 2 中，所以对应树结构的叶子结点中保存的不是下

一跳端口号而是跳转指针 N1 和 N2.

本算法将树结构的端口信息压到线性的转发端口数组(Forwarding Port Array,FPA)上,不采用叶推送技术,而是采用树的层次遍历来保证最长前缀匹配.实际上,每条前缀结点覆盖转发数组上的一段范围,即此段内的所有元素都应该记录这条前缀的端口信息,也就是说,最长前缀匹配其实可以转化为范围匹配.图 3 中,最上面的 P1 覆盖范围[0,15],而下面的 P2 覆盖范围[4,5].为了确保最长前缀匹配,P2 覆盖范围内的元素都应该记录 P2 的端口信息.当采用层次遍历二叉树来构建数组时,短的前缀先被访问.因此当两个前缀结点的范围重叠时,长的前缀将自动覆盖短的前缀.根据此规则,P2 会在遍历第 3 层时覆盖 P1.同样地,N1,P1,N2,P1 会在遍历第 4 层时覆盖上一层计算的值.当第 4 层访问完毕时,我们得到最终的转发端口数组.通过这种方式,树分支等价地转化成线性的数组结构.

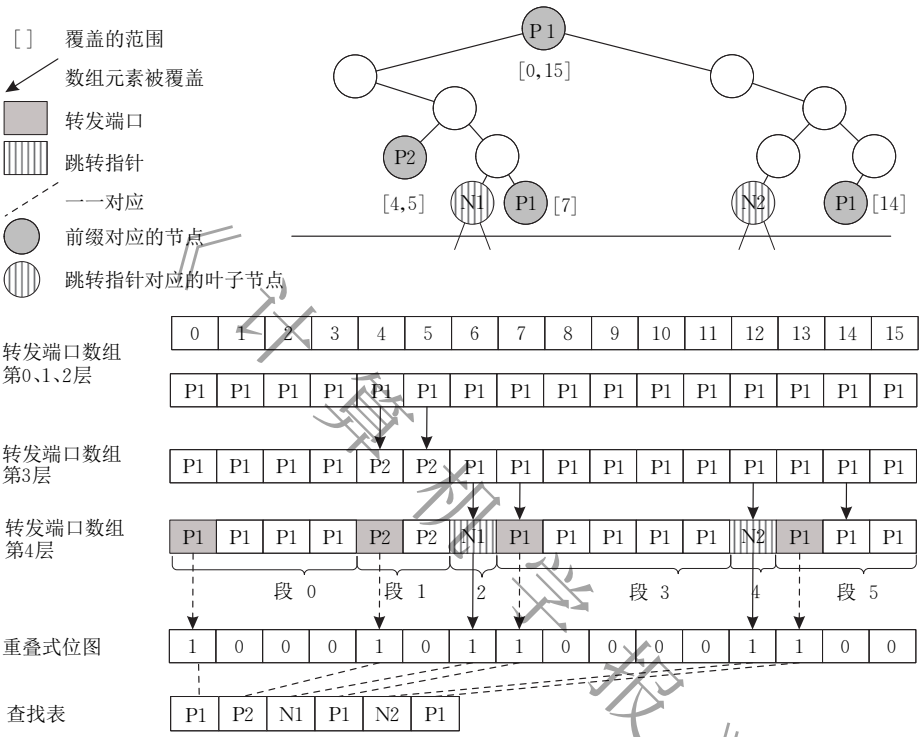

图 3 重叠式位图生成过程

如上所述,数组能够取代树结构实现快速路由查找.但是,直接数组存储方法开销大,信息冗余大:连续的元素往往存储在相同的端口.为了消除冗余,我们采用位图压缩机制将数组转化成重叠式位图及其对应的查找表.位图的每一位依次对应数组的每一个元素,而查找表依次存储每一段的首元素.

对于数组中的每一段,查找表只存储段的首元素.位图将对应于每段首元素的位编码为 1,其他的位编码为 0.因此,位图和查找表之间存在着一一对应的关系:位图的第 N 个 1 对应于查找表的第 N 项,如图 3 所示.这样,位图及其查找表就等价地表示了数组结构.对于给定的目的地址,查找过程首先定位位图的某一位,计算出位图中此位之前(包含)1 出现的次数 N,在查找表中找到对应的查找表项.这样只需要一次访存操作(根据 IP 地址查找位图)和简单的计算(移位运算确定某一位之前 1 个数),就可以得到查找表的地址.

Lulea 算法的位图和查找表的原理与本文算法相同,但是其生成过程有所差别.如图 4 所示,Lulea 算法首先根据路由表生成原始树结构,然后采用叶推送技术生成只有叶子结点包含转发信息的树结构.之后遍历所有的叶子结点,把其“投影”到 Layer 1 最底层的 16 位位图上,每一个叶子结点覆盖范围的最左端的那一位赋值 1,其它赋值 0,得到图中所示的 Lulea 算法位图.最后按照位图顺序,依次把叶子结点的下一跳端口号或者跳转表指针存入查找表.

同 Lulea 算法相比,本算法的位图构造不仅减小存储而且易于更新.原因如下:(1)本算法采用层次遍历产生重叠式位图,因为允许合并,其包含更少的 1,存储开销更小;(2)Lulea 采用叶推送以保证最长前缀匹配.然而,叶推送操作改变树结构,使得增量更新变得复杂.本算法的一个重要特征是保持树

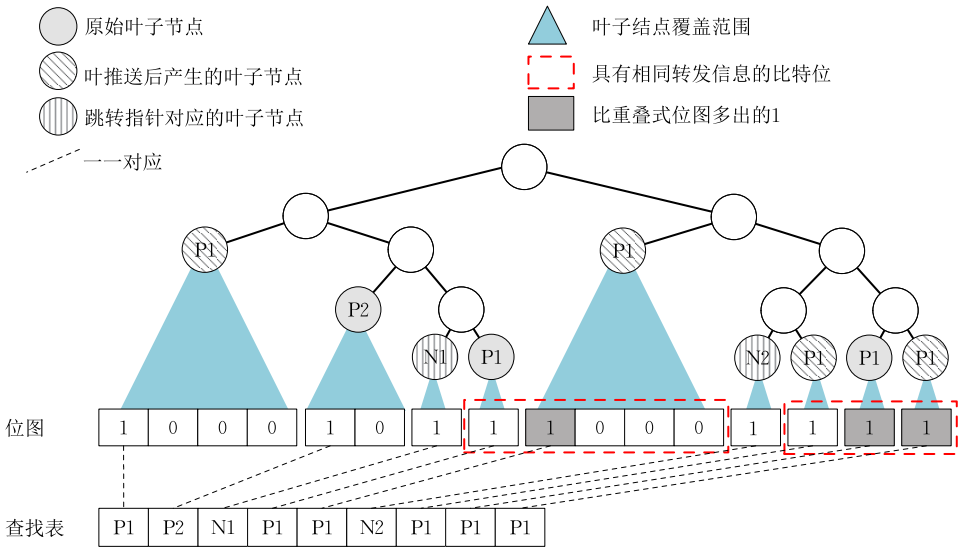

图 4　Lulea 算法位图生成过程

结构不变，因此当更新到来时，新的位图更易生成.

在获得更小存储空间和更优更新性能的同时，本算法牺牲了一定的查找性能. 因为 Lulea 算法的位图是由叶子结点生成的，有一定规律可循，16 位的位图其实只有 676 种有效图案. 所以 Lulea 算法离线计算出这 676 种位图前 n 位（$n=1,2,\cdots,16$）累积的 1 的个数，存储时只保存位图的编号，查找时只需要根据编号查找这个表格即可. 而重叠式位图存储时只能保存实际的位图，查找时采用移位运算＋查表的方式，查找性能会略有降低，但是，由于本算法能把原始路由表压缩的更小，因而有利于在 Cache 中查找，故整体性能反而提高，详见第 5 节实验结果.

3.2　位图分割

层次遍历和重叠式位图虽然能够进一步减少存储空间和在一定程度上提高更新速度，但是根本上不能解决表项级联移动问题. 举例来说，在图 3 中，假定更新是“删除 P2 结点”，所需操作如下：首先位图的第 4 位从 1 变为 0，然后查找表的第 2 项到第 5 项共 4 项依次前移一个位置来维持对应关系. 但是，前移过程与路由查找是互斥的，故引发查找中断. 当位图过长时，分组的实时查找将难以承受被中断的时间间隔. 一种避免表项级联移动的简单方法是：一旦发生更新就重建整个数据结构，先继续使用旧的数据结构进行查找，直到重建完新的数据结构且将新结构下载到线卡为止. Lulea 算法就是重建整张路由表. 但是这种方法要求路由器线卡上配置有两套物理路由表存储器，不仅与减少存储空间相悖，而且也会使得更新延迟加长. 因此，这一方案不可取. 最理想的方案就是实现实时的路由增量更新. 下面我们就提出相应的解决方案.

我们将转发端口数组划分为等长的组，分别为各组建立相互独立的组结构. 依此，数据结构由相互分离且独立的组结构组成. 更新可以以细粒度的组为单位进行，只需要重建和替换被改变的组结构，而保持其它的组结构不变. 组的划分是灵活的：组越小，更新粒度越小，更新速度越快，但需要存储更多的组指针信息；组越大，组指针数越少，但更新的级联影响就越大，更新速度越慢. 组结构的选择也是灵活可变的：线性的数组结构存储开销大，但更新方便；位图结构压缩了存储开销，更新也变得相对复杂. 具体的实现可以根据路由表的特性和更新本身的特性来确定组的划分和组结构的选择，这是存储与更新之间的权衡. 图 5 是组划分的示例，转发端口数组被分割为 4 组，分别包含 $k0,k1,k2,k3$ 个元素，每组有独立的组结构. 在处理更新时，算法保持原先的组结构不变，重建修改的组结构，然后修改指针指向新的组结构.

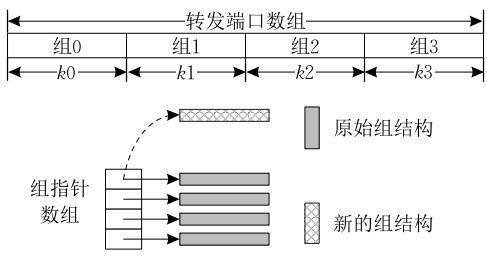

图 5　位图分割示例

更进一步，分组其实是把原来的统一编址变成局部编址，这就需要在原来 Lulea 算法的两级索引

表前再加一级索引表，也就是组号索引表. 查找时，先根据 IPv4 地址查找组号索引表得到组指针，然后用这个指针，去查找相应组内的 Code Word 和查找表. 多加一次查表操作，虽然增加了访存时延，但相对于更新效率的提高是值得的；虽然付出了组号索引表的存储空间，但考虑到指针结构占用空间很小以及重叠式位图算法带来的存储减少，整个算法的存储空间并不一定会增加. 事实上，由后文的实验可知，重叠式位图带来的节省的存储空间相当可观的，其整体存储效率相对 Lulea 反而更高.

3.3 更新优化

在位图分割的基础上，我们提出了两种优化算法来进一步提高更新速度：

(1) 条件优化. 在处理更新时，最耗费 CPU 运行周期的操作就是层次遍历. 故若能减少层次遍历的次数，就能提高更新速度. 本文发现，对于插入或修改的前缀，如果长度为 16 或 24 且对应的树结点是叶子，组结构的重建可以以旧的组结构为基础进行，不需要再经过层次遍历重构组结构.

如图 6 所示，我们以前文的树结构为例，说明条件优化的原理. 当处理一条更新前缀[0111/4 P3]，按照原始的更新流程，需要先修改树结构，然后层次遍历生成端口转发数组，最后重构整个组结构. 条件优化利用位于 layer 之间的叶子结点，也就是 level 16 或 level 24 上的叶子结点在位图中只占用一位且不影响下层 layer 的特性，先通过查找过程确定待修改的组结构，然后根据具体的位图和相邻端口决定插入或删除若干表项并修改位图完成更新.

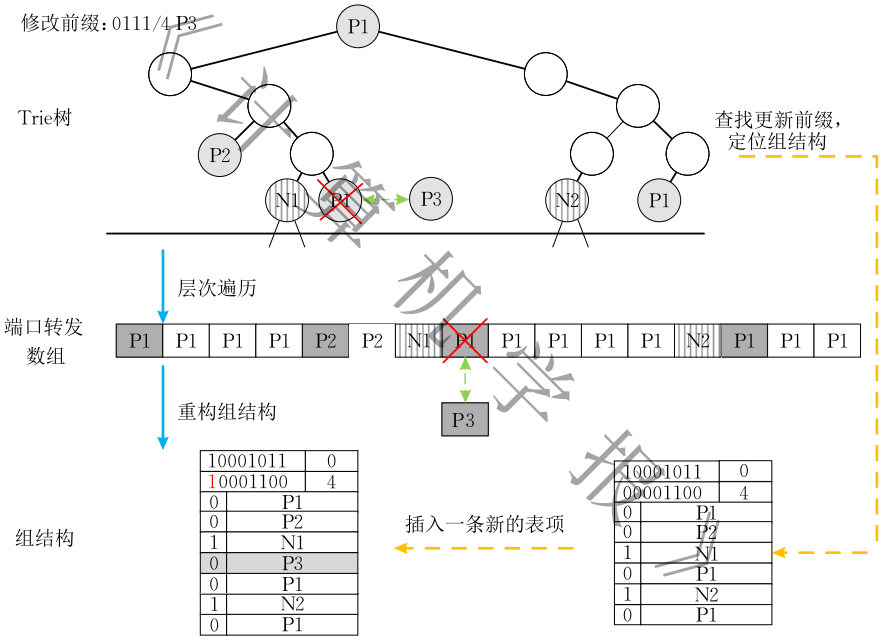

图 6 条件优化更新示例

图 6 中，查找 0111 的下一跳为 P1，其相邻前缀 0110 和 1000 的下一跳分别是 N1 和 P1. 因为更新后的 0111 下一跳是 P3，与原来的端口 P1 不同，结合邻居的端口情况，只需要在 P1 表项前插入一条 P3 表项，并置第二行位图的首位为 1 即可.

(2) 通用优化. 减少重构时层次遍历操作的另一个思想就是用存储换时间. 通用优化就是不释放转发端口数组的存储空间，用于下一次的更新，这样既减少申请和释放存储的时间，且仅需进行一层遍历，避免部分端口值的重新计算.

条件优化在没有增加存储空间的前提下，缩短了处理更新的时间，从而提高的更新速度，是更新算法首先要采用的优化技术. 通用优化牺牲部分存储空间来换取更新速度. 在算法的具体实现中，存储开销和更新性能是一种平衡，是否要采用通用优化要根据实际系统的要求来选择. 在下文中，如果没有特别指出，更新算法都只采用了条件优化.

综上所述，表 2 列出了三种算法的异同点.

表 2 三种算法异同点

名称	相同点	不同点
Lulea	位图结构、[16-8-8]划分、多级索引表	单表、叶推送、无更新优化
LOOP		多表、层次遍历、有更新优化
本算法		单表、层次遍历、重叠式位图、多种更新优化

4 算法实现

如上所述，树结构有 3 种等价的表示形式：分支

树、转发端口数组和位图/查找表，三种形式的存储开销和查找速度各不相同. 通过权衡存储与查找的性能，不同 Layer 采用不同的结构，最终统一成本算法的数据结构.

4.1　算法示例

图 7 是实现示例，切割方式是[16-8-8]. Layer 1 只有一个 16 层的树分支，它的转发端口数组的长度为 $2^{16}=65536$. 在 Layer 2 和 Layer 3 的所有树分支只有 8 层高，转发端口数组的长度都是 $2^8=256$. 为了构建相互独立的组结构，我们将数组划分成等长的组，每组包含 64 个数组元素，即“组大小”是 64. 长度为 32 的 IP 地址按照切割方式分为 3 段. 每一段的高位(*group*1；*group*2；*group*3)用于指定组结构，而低位(*bit*1；*cluster*2，*bit*2；*cluster*3，*bit*3)用于定位组内的元素.

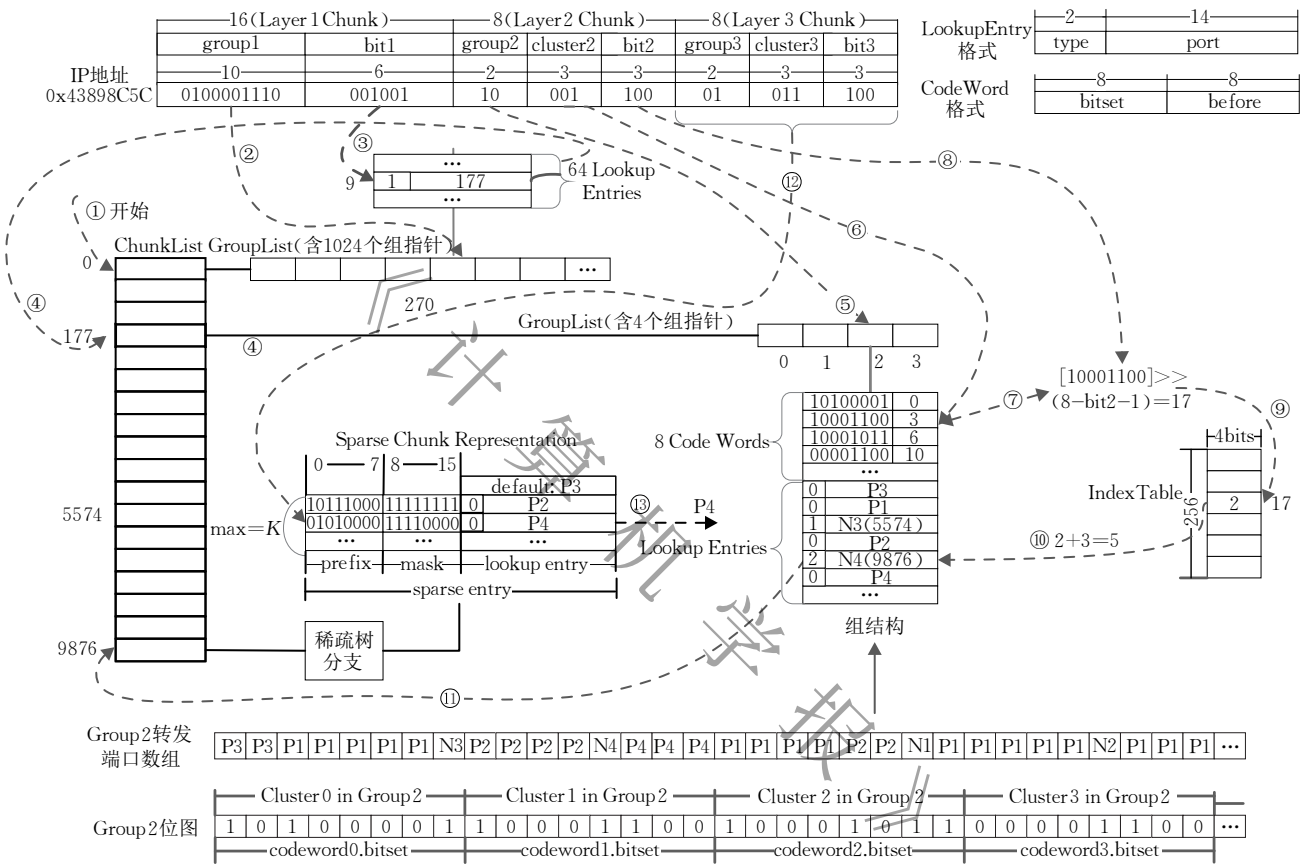

图 7　数据结构及查找示例

Layer 1 的树分支有 $2^{|group1|}=1024$ 组，而 Layer 2 和 Layer 3 的每一个树分支有 $2^{|group2|}=2^{|group3|}=4$ 组. 树分支的组结构相互独立，同一树分支中所有指向组结构的指针构成一个 *GroupList*. *ChunkList* 是指向 *GroupList* 的指针数组，它的每一个元素对应着一个树分支. 显然，*ChunkList*[0]指向 Layer1 的 *GroupList*，每次查找先从 *ChunkList*[0]开始. 在图 7 中，查找表项(*Lookup Entry*)由两部分组成：*type*(2 位)和 *port*(14 位)，共占 2 字节. 查找表项有 3 种类型：(1)*type*=0，port 存储端口信息；(2)*type*=1，*port* 指定 *ChunkList* 的元素，下一层树分支不是稀疏的；(3)*type*=2，*port* 指定的树分支是稀疏的，稀疏树分支的存储结构将在以下部分详解. 组结构的构造如下：

(1) Layer 1 的组结构存储 64 条查找表项，对应本组的 64 个转发端口数组元素，不进行任何压缩处理. 这是因为 Layer 1 只有一个树分支，而且所有的查找必须从 Layer 1 开始，我们牺牲一点存储开销来加速查找.

(2) Layer 2 和 Layer 3 的组结构由位图和查找表构成. 图 7 包含编号 177 的树分支的第 2 组的位图(*Group 2 Bitmap*，从第 0 组开始编号)、转发端口数组(*Group 2 FPA*)和组结构. 每组包含 64 个数组元素，分为 $2^{|cluster2|}=2^{|cluster3|}=8$“簇”(cluster)，每簇包含 $2^{|bit2|}=2^{|bit2|}=8$ 个元素. 8 个 Code Word 分别存储 8 簇的位图信息. 每个 Code Word 占 2 字节，

存储位图的 8 位(1 字节)和计数信息(1 字节). 8 个 Code Word 依次对应 8 簇,高 8 位(*bitset*)存储簇的位图,低 8 位(*before*)记录本组中所有前面的簇中的 1 的个数. 在图 7 的例子中,[10100001,0],[10001100,3]和[10001011,6]是前 3 个 Code Word. 第 2 个 Code Word 的"3"表示第一个 Code Word 的 *bitset*(10100001)中 1 的个数是 3. 第 3 个 Code Word 的"6"表示前两个 Code Word 的 *bitset*(10100001)和(10001100)中 1 的个数是 6. 通过 Code Word 的*before* 字段,我们能够直接计算出前面的簇中 1 的个数. 为了快速计算簇内给定位置之前的 1 的个数,我们使用 Index Table 来存储. 8 位的 *bitset* 共有 $2^8=256$ 种可能的位组合,Index Table 用来记录每一种组合的二进制表示中 1 的个数. 例如,Index Table 的第 20 项记录(00010100)中 1 的个数,所以是 2;第 38 项记录(00100110)中 1 的个数,所以是 3. 下面我们使用例子来说明如何使用 Index Table 去计算给定位置之前的 1 的个数. 为了计算"10100001"的前 5 位中 1 的个数,我们首先将比特串逻辑右移 8−5=3 位而得到新比特串"00010100"=20,Index Table的第 20 项记录着结果 2. 通过使用 Code Word 和 Index Table 结构,我们首先从 Code Word 的 *before* 字段得到之前的簇中 1 的个数,再通过Index Table 计算簇内给定位置之前的 1 的个数.

统计表明,Layer 2 和 Layer 3 的许多树分支只包含少量的前缀结点. 当前缀数目小于等于预定义的值 K(例如 $K=3$)时,树分支被认为是稀疏的,可以被特殊存储. 在图 7 的例子中,稀疏树分支结构(Sparse Chunk)存储了一个数目(*number*)、一个端口(*default*)和一组稀疏表项(*sparse entry*). 数目表示稀疏表项的个数. 端口表示树分支的默认返回端口,当没有匹配时返回. 每一个稀疏表项包含"*prefix*","*mask*","*lookup entry*"三部分. 例如,前缀(0101 ****,P4)可以表示为[*prefix*:01010000],[*mask*:11110000],[*type*:0],[*port*:*P*4]. 当进行匹配时,目的地址首先与"*mask*"做"逻辑与"操作,然后与"*prefix*"做相等比较. 由于[01101101]&[11110000]!=[01010000],所以地址[01101101]不匹配此稀疏表项,而地址[01011100]匹配,因为[01011100]&[11110000]==[01010000]. 注意,例子中的前缀的最大长度为 8,是因为树已被切割. 按照"*mask*"中 1 的个数的非递增顺序,稀疏表项被依次存储. 当查找表项被依次搜查时,首次匹配的表项就是最长前缀匹配. 当所有的稀疏表项都未匹配时,返回默认值. 当访问稀疏树分支结构时,时间开销随着访问稀疏表项的数目的不同而不同. 最坏的情况是稀疏树结构包含 K 个稀疏表项,而所有的表项都不匹配目的地址,只能返回默认值. 因此在实现中,K 必须足够小以保证查找速度.

4.2 查找过程

对应于[16-8-8]的切割方式,查找操作使用目的地址的前 16 比特,中间 8 比特和最后 8 比特从上到下逐层访问 Layer 1、Layer 2 和 Layer 3 的转发端口数组,如图 2 所示. 在任何 Layer 都有可能得到转发端口而返回. 否则,得到的是指向下一层的指针. 最下层的所有数组元素只可能存储端口信息.

算法 1 描述了查找的过程. 查找从 Layer 1 开始,逐层深入 Layer 2 和 Layer 3,直到找到端口信息. 如图 7 所示,假定待查找 IP 地址是[0x43898C5C]. 查找线程首先从 Layer1 的树分支开始(步骤①)直接从目的地址中提取 *group*1([0100001110]=270)和 *bit*1(([001001]=9)的值,访问 Layer 1 的树分支中第 270 组结构的第 9 个查找表项(步骤②,③),得到指向 Layer 2 的编号为 177 的树分支的指针(步骤④). 此树分支不是稀疏的,查找线程再次从目的地址的第二段中提取 *group*2="10"=2, *cluster*2="001"=1 和 *bit*2="100"=4. 查找线程由 *group*2 和 *cluster*2 定位到第"2"组结构的第"1"个 *Code Word*:[10001100,3](步骤⑤,⑥). 其中"3"表示之前簇中出现 1 的个数,而 *IndexTable*[*codeword*.*bitset*≫(8−*bit*2-1)表示簇内给定位置之前出现的 1 的个数(步骤⑦,⑧,⑨). 由于 3+*IndexTable*["10001"]=5(步骤⑩),所以查找表的第"5"项是目标表项(N4)(步骤⑪). Layer 3 的查找与 Layer 2 类似. 因为 N4 的类型是 2,查找线程得到指向稀疏树分支的指针. 此稀疏树分支包含 2 条稀疏表项,而第 2 条匹配目的地址的最后 8 位[01011100](步骤⑫),得到最终的转发端口 P4(步骤⑬).

算法 1. 路由查找算法.

输入:重叠位图结构 *ChunkList*,待查 *IP*,具体变量定义如图 7 所示

输出:下一跳端口号

```
1. cur← 0/ * Layer 1 Chunk * /
2. groupbase←ChunkList[cur].base[group1]
3. entry←groupbase[bit1]
4. IF entry stores next-hop information THEN
5.    RETURN entry.port
6. ELSE
7.    cur←entry.pointer / * Layer 2 Chunk * /
```

```
8. END IF
9. IF ChunkList[cur] is not sparse THEN
10.    groupbase←ChunkList[cur].base[group2]
11.    codeword←groupbase[cluster2]
12.    lookup←groupbase+8
13.    ix←codeword.bits≫(7−bit2)
14.    pix←codeword.before+IndexTable[ix]−1
15.    entry←lookup[pix]
16.    IF entry stores next-hop information THEN
17.       RETURN entry.port
18.    ELSE
19.       cur←entry.pointer
20.    END IF
21. ELSE
22.    search the special entries (0～K)
23. END IF
24. /* 在 Layer3 重复 Layer2 的搜索过程 */
```

4.3　更新过程

本算法的更新机制是重建和替换数据结构. 更新需要完成 3 个操作：(1) 重建改变的组结构；(2) 下载新结构到路由器线卡；(3) 修改组结构指针. 操作 1 花费时间最多，但可与查找并行工作. 操作 2 与操作 3 将旧的组结构更新为新的组结构，这个过程会中断查找，但开销较小，几乎可以忽略. 在操作 3 修改指针之前，查找在旧的组结构上进行. 在操作 3 修改指针之后，查找将在新结构上进行. 通过分割来重建数据结构，中断查找的时间和更新延迟的时间都会大大减少.

更新的种类有 3 种：插入前缀、删除前缀和修改前缀. 当采用条件优化，也就是更新满足以下三个条件时，新的组结构无需从头开始重建，只需修改旧的组结构来构造：(1) 插入或修改前缀；(2) 前缀的长度是 16 或 24；(3) 前缀对应叶子结点.

BGP 更新报告[①]显示大约一半以上的更新是插入或修改前缀且更新前缀长度是 24. 因此，采用条件优化能够大大地改善平均更新性能. 由于更新的局部性，选择合适的组划分和组结构有利于进一步改善更新的性能.

5　实验评估

5.1　实验配置

评估算法的路由表、更新包和流量分别从 Router Views Archive Project[②]、RIPE Network Coordination Center[③] 和 CAIDA Equinix-Sanjose[④] 下载. 图 8 是路由表 Oregon、Equinix 和 ISC 的前缀数目随时间的增长趋势，当前均已接近 600 K. 图 9 是更新的前缀长度分布情况，98% 以上的更新前缀分布在 15～24 之间，而长度为 24 的前缀就占大约 60%，前缀分布具有集中性. 实验的 CPU 配置是 Intel Core i7TM 6700 处理器(3.40 GHz)，它具有 8 MB 的三级缓存. 软件编译环境是 Visual C++2015. 在默认情况下，实验将转发端口数组的每 64 位划分为一组，即组的大小为 64，切割方式采用 [16-8-8]，稀疏树分支的前缀数目限制设置为 $K=3$.

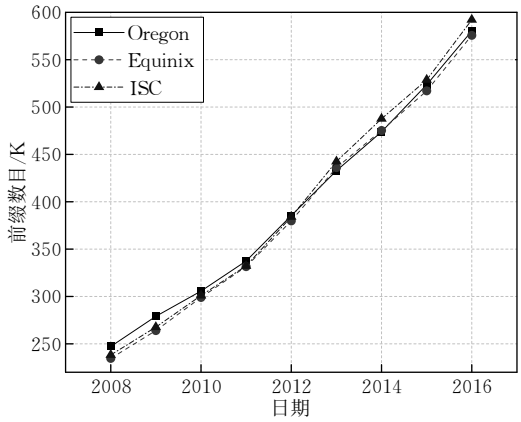

图 8　前缀数目随着时间的增长趋势

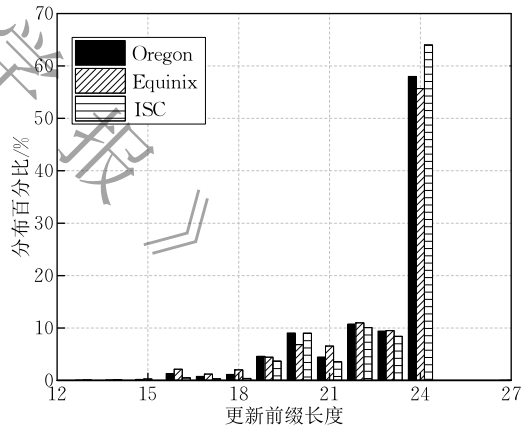

图 9　更新前缀长度的分布

5.2　存储开销

图 10 描述了 Lulea 位图和本算法位图中 1 的数目(也就是查找表的表项数目)随时间变化的趋势.

① BGP Routing Table Analysis Reports (AS65000), http://bgp.potaroo.net, 2017.2.18
② University of Oregon Route Views Archive Project, http://archive.routeviews.org, 2017.2.18
③ RIPE Network Coordination Centre (RIPE NCC), http://www.ripe.net, 2017.2.18
④ Center for Applied Internet Data Analysis (CAIDA), http:// www.caida.org/data/monitors/passive-equinix-sanjose.xml, 2017.2.18

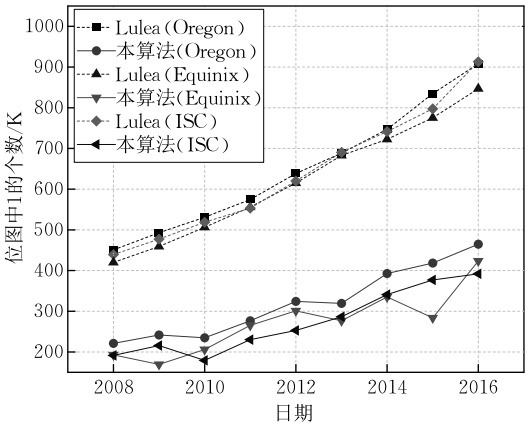

图 10 位图中 1 的个数的比较

随着路由表的增大,两种算法的位图中 1 的数目都线性增长.同 Lulea 位图相比,重叠式位图包含更少的 1,即查找表包含更少的表项.平均情况下,重叠式位图中 1 的数目比 Lulea 少 51%～57%,这表明本算法能够显著减少查找表表项数目,更加有效地压缩存储.本算法的存储开销与树分支的数目正相关.图 11 是树分支数目随时间变化的趋势,可以看到,总的树分支数目逐年增加,但稀疏树分支增长不大,并且大约 1/3 的树分支属于稀疏树分支.

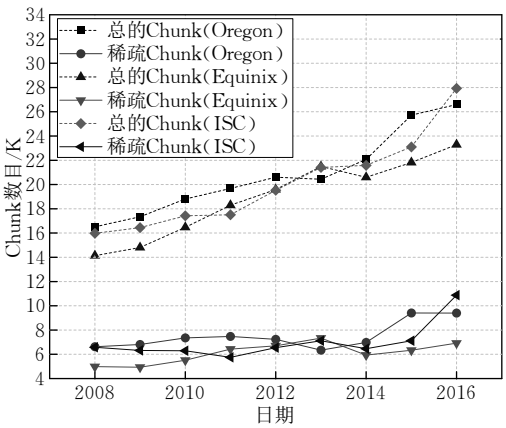

图 11 树分支数目的增长趋势

尽管本算法相比 Lulea 算法能够减少大量的查找表表项,但是位图分割又会产生大量的组指针.为了评估最终的存储开销,我们逐年统计了两种算法所有的存储开销,如图 12 所示.实验结果表明,本算法在能够快速处理路由更新的基础上,平均情况还能比 Lulea 算法减少 26%的存储空间.在图 12 中,随着路由表表项数目增多,Lulea 算法和本算法的存储空间都呈现上升趋势,但本算法增长速度缓慢.而且本算法增长平稳,Lulea 算法在不同时间存储开销波动很大,这也说明本算法的存储开销对具体前缀数目和前缀分布不敏感,具有很强的稳定性.对于路由表 Oregon,Trie 树结构的存储开销在 2008～2016 年间从 8.01 MB 增长到 17.17 MB,其存储开销大约是本算法的 6～8 倍.路由表 Equinix 和 ISC 也表现出类似的性能.对于 2016 的路由表Oregon,本算法每条前缀平均占用 3.94 个字节,平均占用 0.046 树分支,意味着平均每个树分支包含约 22 个前缀结点.尽管路由表的前缀数目随着时间日益增加,但是本算法的存储开销表现出缓慢的增长趋势,这是因为平均每条前缀所占用的存储空间逐渐减少,从 2008 年的 5.06 字节下降到 2016 年 3.94 字节.

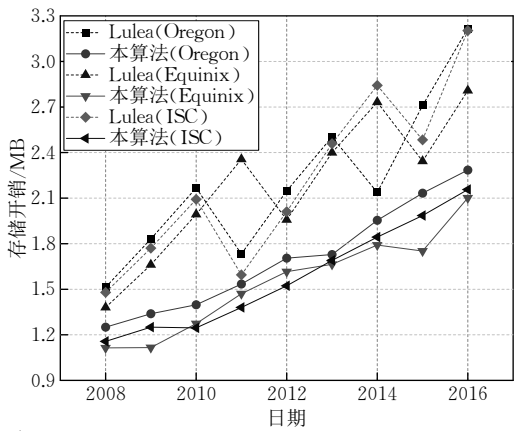

图 12 存储开销比较

5.3 查找性能

查找性能是衡量路由算法性能的关键指标.对于软件查找算法,因其查找性能普遍低于硬件查找算法,所以查找性能一直是影响软件查找算法的瓶颈.我们采用随机生成的流量和实际网络流量(来自 CAIDA)作为输入,分析比较 Trie 树结构、Lulea 算法和本算法的查找性能,结果如表 3 所示.

表 3 三种算法查找速度比较 (单位:M/s)

路由表	随机流量			CAIDA 流量		
	Trie	Lulea	本算法	Trie	Lulea	本算法
Equinix	7.078	25.297	68.647	22.274	44.742	111.057
ISC	6.889	24.361	65.068	21.231	44.146	111.556
Oregon	6.812	24.879	67.510	21.874	44.409	111.621
平均	6.927	24.846	67.075	21.793	44.432	111.411

从表 3 可以看出,不论是何种输入流量,本算法都具有最高的查找速度,随机流量下平均查找速度为 67.0 M/s,实际 CAIDA 流量下平均查找速度为 111.4 M/s.因为 Trie 树结构访存次数太多,所以其查找性能最差,随机流量情况下查找速度仅为本算

法的 1/10. 尽管 Lulea 算法与本算法的访存次数基本相同，但 Lulea 存储开销大，Cache 命中率低，两种情况下查找速度均不到本算法的一半. 另外，当采用实际的网络流量，三种算法的查找性能都有了很大的提高，这是因为实际流量往往具有明显的局部性，Cache 命中率会大大提高.

5.4　更新性能

更新性能评估有两个度量：一个是“中断时间”，它是更新引发的查找中断的时间. 另一个是“处理时间”，它指更新算法本身消耗的时间. 前者对服务质量影响较大，因为长的中断时间导致长的分组缓冲排队甚至丢失分组. 后者与更新的实时性有直接关系，长的处理时间导致大的更新延迟. 在本算法中，更新开销与修改的组结构的数目 N 和修改的稀疏树分支的数目 M 有关. 处理时间是重建 N 组结构和 M 稀疏树分支的时间. 而中断时间是将新数据下载到线卡，然后修改 $N+M$ 指针所需要的时间. 由于大部分的更新发生在 Layer 2 和 Layer 3 上，所以 N 和 M 都很小. 下面将分别讨论处理时间和中断时间.

处理时间：图 13 是每一条更新的处理时间. 路由表 Oregon 的每条更新平均需要 1.7 K CPU cycles，路由表 Equinix 的每条更新平均需要 2.5 K CPU cycles，而路由表 ISC 的每条更新则平均需要 1.6 K CPU cycles. 组结构的重建（处理时间）需要 750～300 K CPU cycles 不等，依赖于更新所在组的前缀分布. 稀疏树分支的重建比组结构的重建花更少的时间. 更新如果修改多个组结构或稀疏树分支，可能会导致图 13 中的一个突起.

表 4 列出了本算法与 Lulea 算法和 Trie 树结构平均处理每条更新所用的时间. Trie 树结构因为不需要叶推送、遍历和构造位图等操作，具有最高的更新速度，平均处理每条更新只需要 0.27 μs. 标准 Lulea 算法由于没有采用位图分割，所以每次更新都需要重构整个查找结构，处理每条更新耗时接近 0.1 s，更新速度远远低于其它两种算法. 本算法采用多种更新优化技术，显著提高了更新速度，平均每条更新仅用时 0.57 μs.

表 5 列出更新经过优化的平均时间开销. “未优化”指对所有更新同样处理，不存储转发端口数组，而在更新的时候重建. 当更新满足优化条件时，处理时间可以降到几百 CPU cycles. 由于很多更新满足条件优化，所以条件优化总体能提升大约 80% 的平均性能. 在实际应用中，通用优化能够大大减少数组端口的重新计算，进一步提升更新的性能. 相比之下，条件优化不需要额外的存储开销，通用优化则不然.

表 4　三种算法平均处理每条更新时间　（单位：μs）

路由表	Trie	Lulea	本算法
Equinix	0.333	88173.850	0.747
ISC	0.231	101364.409	0.461
Oregon	0.253	98159.351	0.512

表 5　更新优化性能（CPU 3.4 GHz，组的大小默认为 64）

	未优化/M	条件优化/M	条件＋通用优化/M
平均性能（更新/s）	1.011	1.820	2.412

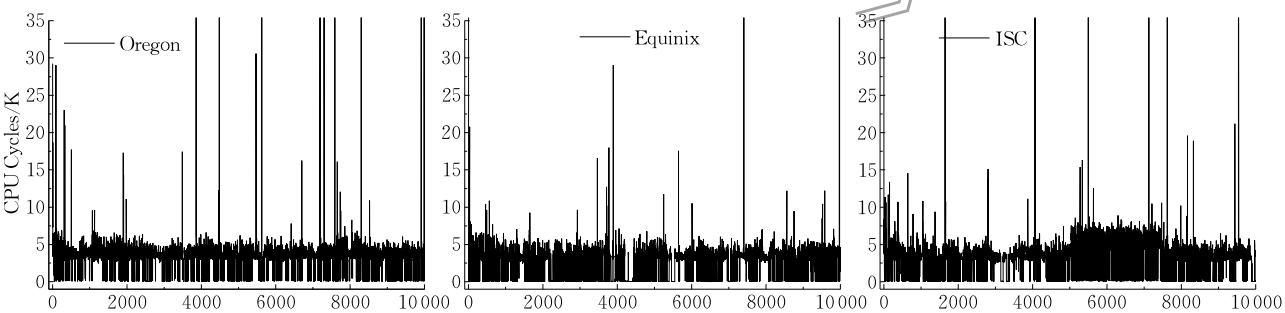

图 13　逐条更新时每条更新前缀的处理时间

中断时间：图 6 是修改指针的数目（$N+M$）的分布. 超过 74% 的更新前缀只修改一个指针，也就是说大部分更新只修改一个组结构或一个稀疏树分支，那意味着 $N=1$ 和 $M=0$，或者 $N=0$ 和 $M=1$. 一般来讲，稀疏树分支的数据结构至多占用 16 字节，而组结构占用约 60～150 字节. 因此，下载数据结构和修改指针的中断时间是很短的.

表 6　更新前缀修改指针数分布

修改指针数（$N+M$）	1	2	大于 2
Oregon 路由表	76.48	0.02	23.50
Equinix 路由表	82.87	0.12	17.01
ISC 路由表	74.16	0.08	25.76

上述实验将位图的每 64 位划分成一组. 实际上，划分有多种方式. 组的大小影响存储和更新. 分

组越小越利于更新,但导致越大的存储开销.17 是当采用不同的组大小(32,64,128)时每条前缀的平均存储开销和每条更新的平均处理时间.权衡存储和更新的性能,我们将组的大小设为 64.

表 7 组的大小对存储和更新的影响(条件优化)

组的大小	平均每条前缀存储开销/(字节/条)	更新速度/(更新/s)
32	4.09	2.157
64	3.94	1.820
128	3.87	1.333

软件路由算法的查找性能在很大程度上依赖于 Cache 的命中率.在实际网络中,查找的过程伴随着更新.如果查找无中断进行,流量局部性有利于 Cache 命中率提升,从而提高查找吞吐量.交替地处理更新与查找会影响 Cache 行为,从而降低整体性能.图 14 是查找速度随着更新速度的变化趋势(只进行条件优化).图中采用三种查找与更新的混合方式:(1)'均衡'指每隔一定量的查找插入一条更新,更新均匀地混合到查找中;(2)'随机'指将更新随机地混合到查找中;(3)'突发'指每隔一定量的查找混合一段时间(1s)的更新.在初始情况下,随着更新速度的提高,查找性能快速下降.而当更新更加频繁时,查找速度几乎线性递减.在无更新的情况下,本算法能够完成约 100 M/s 的查找速度;当只处理更新时,本算法能够达到接近 2 M/s 的更新速度.真实的网络在突发时每秒只会有几十 K 的更新,远远达不到 1 M/s.在支持 100 K/s 更新的前提下,本算法还能够达到 90 M/s 以上的查找速度,是普通树查找速度的 4 倍多.相比之下,Lulea 算法仅能达到一半的查找速度,而且它不支持增量更新.综合存储、查找和更新性能,本算法性能优异,能够更好地满足实际网络的性能要求.

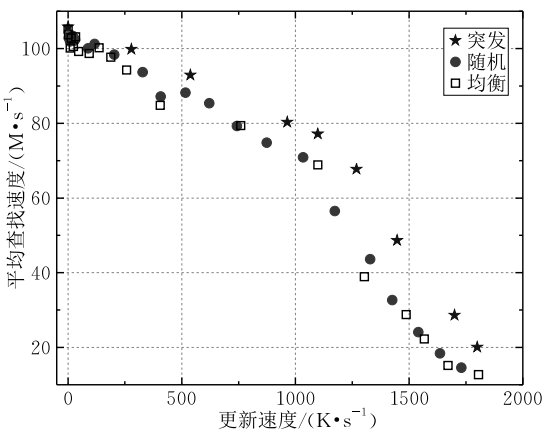

图 14 查找随更新增长的变化趋势

6 结束语

本文提出了一种基于重叠位图压缩的软件路由查找算法.该算法具有位图算法存储高效的特点,同时采用重建和替换的机制克服位图增量更新的困难.位图压缩使得存储开销随着路由表增大而增长缓慢,水平切割使得查找深度有限,且不受路由表增大的影响.对比于 Lulea 算法,该算法采用层次遍历减少位图中 1 的个数,既减小存储又有利于增量更新.另外,采用位图分割和更新特性优化,减少更新时间.本算法能够将 600 K 的路由表压缩存储为 2.3 MB 的高效数据结构,在一秒钟内同时处理 10～100 K 更新和 90～100 M 查找.实验表明该算法具有良好的存储、查找和更新性能,而且可扩展性好.

致 谢 感谢同课题组的米志安同学,他在该研究中做了前期的基础性工作!

参 考 文 献

[1] McAuley A, Francis P. Fast routing table lookups using CAMs//Proceedings of the IEEE International Conference on Computer Communications. San Francisco, USA, 1993: 1382-1391

[2] Zane F, et al. CoolCAMs: Power-efficient TCAMs for forwarding engines//Proceedings of the IEEE International Conference on Computer Communications. San Francisco, USA, 2003: 42-52

[3] Zheng K, Hu C, Lu H, Liu B. A TCAM-based distributed parallel IP lookup scheme and performance analysis. IEEE/ACM Transactions on Networking, 2006, 14(4): 863-875

[4] Le H, Ganegedara T, et al. Memory-efficient and scalable virtual routers using FPGA//Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays. Monterey, USA, 2011: 257-266

[5] Han S, Jang K, et al. PacketShader: A GPU-accelerated software router//Proceedings of the ACM Special Interest Group on Data Communication. New Delhi, India, 2010: 195-206

[6] Nilsson S, Karlsson G. IP-address lookup using LC-tries. IEEE Journal on Selected Areas in Communications, 1999, 17(6): 1083-1092

[7] Lampson B, Srinivasan V, Varghese G. IP lookups using multiway and multicolumn search. IEEE/ACM Transactions on Networking, 1999, 7(3): 324-334

[8] Dharmapurikar S, Krishnamurthy P, et al. Longest prefix matching using bloom filters. IEEE/ACM Transactions on Networking, 2006, 14(2): 397-409

[9] Degermark M, Brodnik A, Carlsson S, et al. Small forwarding tables for fast routing lookups//Proceedings of the ACM Special Interest Group on Data Communication. Cannes, France, 1997: 3-14

[10] Eatherton W, Varghese G, et al. Tree bitmap: Hardware/software IP lookups with incremental updates. ACM Special Interest Group on Data Communication Computer Communications Review, 2004, 34(2): 97-122

[11] Rétvári G, Tapolcai J, Korösi A, et al. Compressing IP forwarding tables: Towards entropy bounds and beyond//Proceedings of the ACM Special Interest Group on Data Communication. Hong Kong, China, 2013: 111-122

[12] Yang T, Xie G, Li Y B, et al. Guarantee IP lookup performance with FIB explosion//Proceedings of the ACM Special Interest Group on Data Communication. Chicago, USA, 2014: 39-50

[13] Asai H, Ohara Y. Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup//Proceedings of the ACM Special Interest Group on Data Communication. London, UK, 2015: 57-70

[14] Mi Z, Yang T, et al. LOOP: Layer-based overlay and optimized polymerization for multiple virtual tables//Proceedings of the IEEE International Conference on Network Protocols. Göttingen, Germany, 2013: 1-10

[15] Yang T, Mi Z, et al. An ultra-fast universal incremental update algorithm for trie-based routing lookup//Proceedings of the IEEE International Conference on Network Protocols. Austin, USA, 2012: 1-10

[16] Srinivasan V, et al. Fast address lookups using controlled prefix expansion. ACM Transaction on Computer Systems, 1999, 17(1): 1-40

LIU Bin, born in 1964, Ph. D., professor, Ph. D. supervisor. His research interests include high-performance switches/routers, network processors, named data networking and software-defined networking.

ZHANG Chu-Wen, born in 1992, Ph. D. candidate. His research interests include high-performance switches/routers, named data networking and vehicle networks.

Background

Routing lookup has been a research hotspot since the birth of TCP/IP, and it is growing the second spring due to the rise of SDN. This paper also focuses on this old but still thriving problem. Currently, with the decades of research and technology development, there have been thousands of papers and patents on accelerating lookup speed, decreasing storage overhead or improve update performance. In general, we can divide routing lookup algorithm into hardware-based algorithm and software-based algorithm. The former one, such as algorithm based on TCAM, FPGA and GPU, is mainly deployed in backbone network, which is strict to lookup speed but does not care about cost too much. While, this paper is focus on the software-based algorithm that is sensitive to cost, including the traditional trie structure, the bitmap lookup algorithm with high compress ratio and some algorithms based on bloom. Although researchers have done many meaningful works, no software algorithm that can have balanced performance considering the lookup speed, storage overhead, update performance and flexibility under SDN. Therefore, we are devoted to designing an algorithm satisfying the above requirements.

Based on trie structure, Lulea and LOOP algorithm, we propose a new software-based bitmap routing lookup algorithm that adopt overly bitmap to increase the compress ratio. We avert the traditional Leaf Pushing and make use of level traverse to get the bitmap that has the nature of overlapping. In addition, we take advantage of bitmap segmentation and multiple optimization methods inspired by update locality, which enhance the update speed effectively.

Experimental results show that our scheme can save 20%—30% storage overhead than the Lulea algorithm which is recognized as the highest compression algorithm. It also have good scalability and flexibility because the storage overhead grows very slowly with prefix number increasing. Furthermore, it can compress a routing table of 600 K prefixes to 2.3 MB and achieve a fast lookup up to 90—100 MSPS (million searches per second) under 10—100 K updates per second.

This work was supported in part by the National Natural Science Foundation of China under Grant Nos. 61432009, 61373143, 61602271, which aims to address the modeling, architecture and operating principles on SDN. Specially, high-speed flow table lookup and update with small memory footprint is one of the key research directions in our foundation proposal. Although, our scheme is under the background of traditional networks, it can also be extended to SDN and improving the flow table lookup and update performance due to its scalability and flexibility.