Poptrie：一种用于快速和可扩展软件 IP 路由表查找的基于位 1 计数（Population Count）的压缩 Trie

摘要

物联网导致路由表爆炸。面对不断增长的 Internet 规模，需要一种廉价的 IP 路由表查找方法。我们通过一种基于多路径 trie（称为 Poptrie）的快速且可扩展的软件路由查找算法做出贡献。在我们的遍历树的方法命名后，它利用后继节点的位向量索引上的位 1 计数指令来压缩 CPU 高速缓存内的数据结构。在使用 35 个路由表（包括真正的全球 tier-1 级 ISP 的完整路由表）的所有随机和真实目的地查询的评估中，Poptrie 都胜过了最先进的技术，即 Tree BitMap，DXR 和 SAIL 。在单核和 500-800k 路由表中，Poptrie 峰值在每秒 174 百万次和超过 240 百万次查询（Mlps）之间，始终比我们运行的所有测试中的所有竞争算法快 4-578%。我们提供了全面的性能评估，特别是 CPU 周期分析。本文展示了 Poptrie 在包括 IPv6 在内的未来互联网的适用性，当预计到具有更长前缀的更大的路由表。

CCS 概念

网络→路由器；网络算法。

关键字

IP 路由表查找；最长前缀匹配；trie。

1、介绍（Introduction）

由于日常生活高度依赖互联网，因此互联网上支持高速通信的基本功能随着流量的不断增长而日益重要。其中一项关键技术是 IP 路由表查找：由于互联网核心路由器中的峰值流量大小为每秒数百 G 比特（Gbps），因此它需要非常的快速。三态内容寻址存储器（TCAM）在互联网核心路由器中执行高速 IP 路由表查询。然而，从 TCAM 中脱离出来是一种值得考虑的方法，其原因有二：首先，

TCAM 在功耗和散热方面存在问题。其次，网络功能虚拟化（Network Functions Virtualization，NFV）[6]的出现可能会使 TCAM 无法使用，因为虚拟化网络功能目前在软件中实现而不是 TCAM。因此，希望仅用通用计算机来实现软件高速 IP 路由器;即个人计算机（PC）或商用现成产品（COTS）的设备。

长期以来，IP 路由表查找一直是使用 COTS 设备进行软件 IP 转发的瓶颈[11]。这是一个具有挑战性的问题[34,31]，因为：1）路由表的大小很大并且不断增长（BGP 完整路由的数量超过了 500K），2）它需要特定的计算密集型处理步骤，称作“最长前缀匹配”，以及 3）高速通信链路需要高速处理（在最小尺寸数据包的 100G 比特以太网（GbE）上用于线速 IP 数据包转发的每秒 148.8 百万次查找（Mlps））。

最近，我们看到软件路由器性能的显着提高。有两种方法;人们期望使用诸如图形处理单元（GPU）之类的特定硬件。然而，这样的硬件继承了 TCAM 所具有的类似问题，例如热量和功耗。另一种是纯粹的软件算法方法，我们假定使用商用 CPU。这种方法的趋势是减少 IP 路由表的数据结构的内存占用量，以最大限度地发挥 CPU 缓存的优势[38,25]。尽管如此，要满足多个 100 GbE 链路的性能要求还没有实现。

我们为快速和可扩展的 IP 路由表查找提出了一种新的数据结构，称为 Poptrie。它建立在 64 元多路 trie 上，允许在树中进行少量步数的搜索。它使用一个 64 位的向量实现后代节点数组，从而使得内存占用量小，并能够快速检查后代节点。通过对位向量使用位 1 计数指令，不必要的后代节点被有效地省略，并且快速跳转到相应的后代节点变得可行。由于 Poptrie 将后代内部和叶节点放置在连续数组中 ，因此实现了更小尺寸的间接索引，从而大大减少了整个数据结构的内存占用量。 Poptrie 还支持高效的增量更新，而不会阻塞 IP 路由表查找过程。

通过本文的综合评估，我们展示了 Poptrie 对具有更长前缀的更大的路由表包括 IPv6 在内的未来互联网的友好性。与三种最先进的技术相比，Poptrie 在 35 个路由表实例的随机和真实目的地查询的所有评估中提供了卓越的性能。即使在全球一级 ISP 的核心路由器的真实路由表实例上使用单个 CPU 内核，它的运行速度为 241 Mlps（比最快替代方案 DXR [38]快 1.34 倍），并且在使用四个 CPU 内核时可以达到 914 Mlps。此外，我们分析每次查找所消耗的 CPU 周期，并显示其他查找技术与 Poptrie 之间的差异。Poptrie 的贡献之一是针对未来路由表增长的可扩展性。我们展示了 Poptrie 在具有超过 800K 路径的合成表上达到了 175 Mlps，然而 DXR 减慢到 104 Mlps，并且 SAIL [36]不起作用。

本文的其余部分安排如下。在第 2 节中，我们看到相关的过去研究。我们在第 3 节中描述了 poptrie 算法。第 4 节给出了评估，我们描述了路由表数据集，我们如何在基准测试中生成用于查询的目标地址，性能比较，CPU 周期数的分

析，用于大型路由表的可伸缩性比较，更新性能评估以及 IPv6 路由表的性能比较。在第 5 节中进行了各种讨论之后，我们在第 6 节中做出结论。

2、相关工作

TCAM 长期以来一直是路由表查找的常用技术[23,37,39]。但是，TCAM 在功耗，散热，货币成本和可扩展性方面存在问题[4,16]。Bando 等人[4]提出了一种名为 FlashTrie 的基于 FPGA 的路由表查找引擎，每个引擎可提供 200 Mlps。另一种实现快速 IP 路由表查找的方法依赖于 GPU [14]。尽管 GPU-Click [32]和 GAMT [21]等基于 GPU 的技术速度高达 500 Mlps，但它们需要耗电的 GPU，并且需要处理大批量的数据包。大数据包批量大小可能导致更高的最坏情况数据包转发延迟和抖动。

从专用硬件出发，已经有研究在 COTS 设备上实现高性能 IP 路由。Click[19]是一个模块化的软件路由器，可以实现快速数据包 I/O（用于这个时代），而 RouteBricks [10]进一步推进了该模型。Rizzo 致力于加速 Unix 系统[26]，虚拟机[28]和虚拟交换机[27]中的数据包 I/O。NFV [6]的出现使得在 COTS 设备上使用这些快速数据包 I/O 技术生产软件路由器实现时，快速 IP 路由查找更为重要。

基数树（radix tree）[29,7,18]和 Patricia trie [24,30]是最长前缀匹配的基本数据结构。一般来说，每个 IP 路由表查找需要几十次内存访问，这会导致查找性能低下。Waldvogel 等人[34]在前缀长度上使用二进制搜索减少了 IPv4 和 IPv6 路由表查找的内存访问。Gupta 等人 [13]重点关注路由表中前缀长度的分布，大多数前缀不超过/ 24。他们提出了 DIR-24-8-BASIC 数据结构，将每个/ 24 或更短的前缀提取到/ 24 前缀中，以便为这些条目提供 O（1）的查找算法。

有些研究利用布隆过滤器（bloom filters）[9]，内存流水线[5,15,20]和树中的 bitmap[11]来解决这些问题，但这些技术无法提供良好的性能或合理的管理成本。

Lulea 算法[8]被提出来减少路由表的内存占用。Srinivasan 等人[31]考虑了数据缓存并优化了数据结构以提高缓存效率。Tree BitMap [11]提供了一个多路 trie 的简洁数据结构。Tree BitMap 节点中的前缀和后代节点由两个位图和指向连续数据数组的指针表示。它使用与 Poptrie 类似的位 1 计数操作。在我们的测试中，即使在最有利的情况下，Tree BitMap 也只能达到其他现代算法性能的三分之一。4.5 节更详细地讨论了它的性能。Retvari 等人[25]提出了转发信息库（FIB）表的压缩算法，以最大限度地发挥数据缓存的优势。但是，较小的 FIB 表并不总能提供良好的查找性能。例如，它们的算法每个 IPv4 地址查找要消耗 194 个 CPU 周期，因此查找速率仅为 12.8 Mlps。这是因为他们的数据结构的深度可以增长多达 21 个。Zec 等人提出的方法[38]称为 DXR，通过利用高速缓存效率实现了高

查找率（每个 CPU 内核 100 Mlps）。 DXR 将路由表中的前缀转换为地址范围数组，并使用二分搜索来查找基于关键字地址的范围数组。然后它引入一个类似于 DIR-24-8BASIC 的查找表来优化较短前缀的查找性能。然而，他们的方法的瓶颈是对较长前缀的二分搜索。

杨等人提出了又一个快速的 IP 路由表查找算法 SAIL [36]。他们的方法通过将处理过程分成三个级别来减少查找算法中存储器访问和指令的数量。然而，SAIL 的内存占用量超过了典型的 CPU 高速缓存大小，在缓存未命中的情况下需要相对较慢的 DRAM 访问。因此，SAIL 的性能取决于流量模式的目标 IP 地址位置。此外，SAIL 不支持路由表的未来增长，因为它的结构限制在 4.8 节讨论。

3、POPTRIE

我们提出了一种称为 Poptrie 的新数据结构，用于快速 IP 路由表查找。我们假设 Poptrie 仅用于查找 FIB 索引以用于确定 IP 转发期间的下一跳；路由被保存在一个单独的路由表（RIB：路由信息库）中，例如 radix 或 Patricia trie，这样我们可以积极地压缩具有相同下一跳的路由。在本节中，我们将介绍 Poptrie 的工作原理。通常，我们可以通过删除不影响查找结果的冗余前缀来将从 RIB 摄取的路由聚合到 FIB。这种聚合在本文中称为“路由聚合”，不是我们的贡献，也适用于其他查找技术。路由聚合执行一组前缀与属于子树的相同下一跳的合并，而没有任何表示整个子树的单个前缀，除非另有说明，本文中显示的 Poptrie 的性能结果与路由聚合选项一起使用。

Poptrie 从多路径 trie（例如 $M = 2^k$ 的 M-way 或 M-ary）扩展而来。每个节点在后代数组中保存 2^k 个元素，对应于关键字 IP 地址中的 k bit 块的值。后代数组中的一个元素指向其下一级子内部节点或一个叶节点，该叶节点持有对应 FIB 条目的索引。尽管我们选择 k = 6 来实现以适应 64 位 CPU 架构的寄存器大小，但为了简洁起见，本节将说明 k = 2 的情况。图 1 中显示了 k = 2 的 2^k 多路 trie。Poptrie 的令人钦佩的性能是由于内存占用量小，因此它可以完全包含在 CPU 高速缓存中，但它利用有效的多路分支来减少搜索树所需的步骤总数。

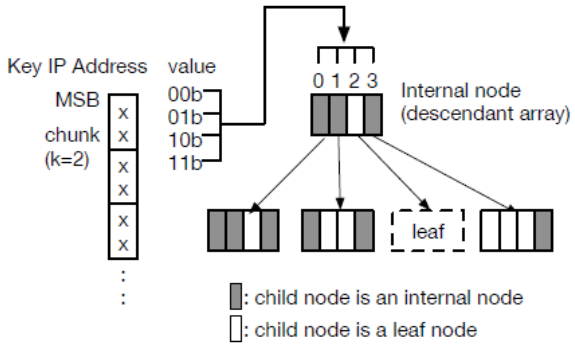


Figure 1: The 2^k-ary multiway trie ($k = 2$).

我们逐步描述了 Poptrie 的操作; 第 3.1 节中的基本机制，第 3.2 节中的查找算法，第 3.3 节中用于压缩叶大小的 leafvec 扩展以及第 3.4 节中称为直接指向的附加选项。 基本 Poptrie 中的内部节点包含向量（8 字节），base0（4 字节）和 base1（4 字节）。 因此内部节点的总大小只有 16 个字节。当我们使用 leafvec 扩展时，它另外需要 8 个字节，所以内部节点大小变为 24 个字节。在我们的实现中，内部和叶节点的连续数组由伙伴内存分配器（buddy memory allocator）[17]管理。

3.1 基本机制

首先，将多路 trie 中的后代数组改变为按位数组（即，位矢量）。vector 和 base1 共同作为后代数组。对于当前的 k 位地址块，向量是长度为 2^k 位的位向量索引。向量中的第 n 位对应于当前 k 位地址块中值为 n 的子节点。向量中的每个位都指示相应子节点的类型：如果相应子节点是内部节点，则该位设置为 1;如果相应子节点是叶节点，则设置为 0。换句话说，vector 指示相应的后代内部节点的存在，并且如果没有后代内部节点，则搜索将总是在该树的该层处指向叶节点。

必要的后代内部或叶节点被放置形成一个连续的数组。当 k = 6 时，该阵列以对应于 n = 0 的后代节点以升序直到 n = 63 开始。然而，不必要的节点（即，该节点未指向后代内部节点或叶节点）被跳过；如果节点对应于 n = 0 不是必需的，则阵列可以从 n = 1 开始，并且正确设置向量中的对应位。这样，省略了不具有分支或叶子信息的不必要的后代节点，从而允许紧凑的数据结构大小和有效使用存储器。

树中的搜索中的下一个节点可以如下获得：由于当前 k 位地址块的值 n 对应于向量中的第 n 位，因此最低有效 n + 1 位中的 1 的数量可以用作当前内部节点的后代数组中下一个节点的索引。在这里，我们可以利用 *popcnt* CPU 指令来加速搜索过程中下一个节点的计算，如 3.2 节所述。由于 vector 位向量仅提供当前

后代数组中的间接地址（即索引），因此有必要提供后代数组的起始点。 base1 是作为此节点的子节点的内部节点的连续子序列的基本索引。类似地，叶节点的间接索引通过计数向量中的 0 来获得。叶节点的起始偏移量包含在 base0 中。图 2 举例说明了使用 vector，base1 和 base0 的内部和叶节点的间接索引。

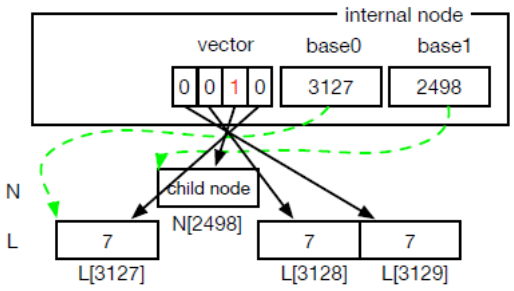


Figure 2: The *vector* in the internal node is configured so that the bit-1 indicates a descendant node, and the bit-0 indicates a leaf node.

3.2 查找算法

查找算法按照指定的 IP 地址在树中向下走，如正常的 2^k 路多路树。在深度 d 处，地址的第 d 块用作内部节点中向量的索引。设密钥地址中第 d 块的值为 n，然后按如下方式执行深度 d 处的查找：如果相应位为 1，则查找算法继续到下一个深度。通过向 base1 添加矢量的最低有效 n + 1 个比特中的 1 的数目减 1 来计算后代阵列中下一个内部节点的索引。如果对应比特为零，则查找算法结束查找并返回叶节点。叶子数组中叶节点的索引是通过将 base0 加上向量的最低有效 n + 1 个比特中的 0 的数目减 1 来计算的。

Poptrie 的关键点是使用该指令来计算 bit 串中 1 和 0 的数量。这些计数分别用作后代节点和叶节点的间接索引。在一个位串中计数 1 的过程被称为“population count”，并且执行它的指令 *popcnt* 已经在 x86 处理器的指令集中实现了。当 *popcnt* CPU 指令不可用时，一个快速替代方法可以在文献中找到[35]。

在算法 1 中显示了 k＝6 的查找算法。该算法将 poptrie 结构 t 和 IP 地址 key 作为输入，并返回最长匹配叶节点。在 t 中，有内部节点数组 N 和叶子数组 L。在第 1 行中，索引设置为 0 以访问根节点。第 2 行访问根节点的 vector。在第 4 行中，我们从偏移量 0 中获得第一个 6 位块的值。第 5-12 行是主循环，只要存在相应的后代内部节点（在第 5 行中检查）就会继续。第 7 行获取最低有效位 v + 1 位中设定位的总数，并将其存储在 bc 中。计算下一个节点的索引（第 8 行），准备下一个节点的矢量（第 9 行），并将该块移位下一个 6 位（第 10,11 行）。第 13-15 行计算相应叶节点的间接索引，并返回内容。

Algorithm 1 lookup($t = (N, L)$, key); the lookup procedure for the address key in the tree t (when $k = 6$). The function extract(key, off, len) extracts bits of length len, starting with the offset off, from the address key. N and L represent arrays of internal nodes and leaves, respectively. $\ll$ denotes the shift instruction of bits. Numerical literals with the UL and ULL suffixes denote 32-bit and 64-bit unsigned integers, respectively. Vector and base are the variables to hold the contents of the node's fields.

```
1: index = 0;
2: vector = t.N[index].vector;
3: offset = 0;
4: v = extract(key, offset, 6);
5: while (vector & (1ULL ≪ v)) do
6:     base = t.N[index].base1;
7:     bc = popcnt(vector & ((2ULL ≪ v) - 1));
8:     index = base + bc - 1;
9:     vector = t.N[index].vector;
10:    offset += 6;
11:    v = extract(key, offset, 6);
12: end while
13: base = t.N[index].base0;
14: bc = popcnt((~t.N[index].vector) & ((2ULL ≪ v) - 1));
15: return t.L[base + bc - 1];
```

3.3 用叶结点位向量压缩

前面小节中描述的前缀扩展会产生许多重复和冗余的叶子。在普通的 2^k 多路径 trie 中，对应于较短前缀的相同 FIB 条目可冗余地跨越到内部节点内的多个叶，最多 2^{k-1} 叶。例如，当 k＝6 时，内部节点的 64 长度的叶子阵列中可以是值 A 的下一跳和值 B 的 63 次下一跳。这样，多余的叶子（在这个例子中是 B）会消耗大量的内存。

为了避免它们，*leafvec* 被引入到 poptrie 内部节点中。leafvec 是一个位向量，指示叶阵列中相同的连续叶结点范围的起始点。这是一项关键技术，可以让 Poptrie 的内存占用量减小，并且可以减少超过 90％的叶子，如 4.3 节所述。leafvec 和 base0 共同作为定位叶节点的基础和间接索引，与上一节中介绍的 vector 和 base1 类似。只要冗余叶时隙是连续的，使用 leafvec 和 base0 的间接索引就省略冗余信息。例如，如果内部节点中的所有 64 个叶子时隙都包含相同的值，则它可以被压缩到只有一个叶子时隙，只有叶子中的最低有效位为 1。叶节点的间接索引中，当前块的值 n 的被计算为 leafvec 中最低有效位 n＋1 中的 1 的数量。这样，任何值 n 的所有间接索引落入第一个叶片槽中，从而实现有效的内存压缩。

这种机制的另一个好处在于 hole punching 发生。hole punching 是一个事件，使得较长的前缀分开为较短前缀的地址空间，因此路由表需要将地址空间作为三个不同的分区进行处理。通常，打孔可防止叶片连续，从而使上述有效叶片压缩无效。然而，在 Poptrie 中，如果存在与叶插槽对应的后代内部节点，则使叶插槽无关，从而恢复邻接。查找算法首先总是检查后代内部节点的存在性，如果有，则查找从不从较低级别追溯到当前级别。因此，具有相应后代内部节点的叶片槽被定义为不相关的，并且被设置为 0.然后，我们可以使叶片槽再次连续，忽略具有相应后代内部节点的叶片槽，如图 3 所示。

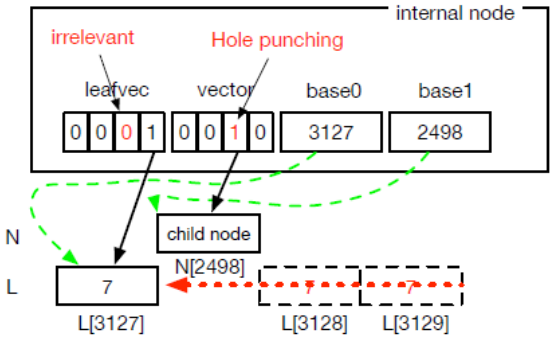


Figure 3: Merging identical leaf nodes with ignoring a hole punching using the *leafvec*

Algorithm 2 The leaf compression algorithm; the differences from the Algorithm 1. Only Line 14 is substituted.

14: bc = popcnt($t.N$[index].leafvec & ((2ULL ≪ v) - 1));

算法中的修改如算法 2 所示。只有 14 行同算法 1 不同，以便它检查新引入的 leafvec 字段以计算对应的叶索引。图 4 说明了使用 leafvec 的查找过程的一个示例；一个 8 位地址 01100111b 的搜索过程如下：（a）从地址中取前两位（01b），然后选择与该索引相对应的位（右起第二位）。（b）使用 base1 成员找到内部节点的下一个子序列的基地址。（c）它计算向量成员的最低有效两位中减 1（= 1）的个数，并找到下一个内部节点。（d）从地址中取出后两位（10b），然后选取与该索引对应的位（右起第三位）。矢量中的位为 0，因此搜索将切换为查找叶节点。（e）它从 base0（128）找到相应叶节点的基地址。 （f）它计算 leafvec 的最低有效三位中的 1 的数量减 1（= 0），最后找到叶节点。

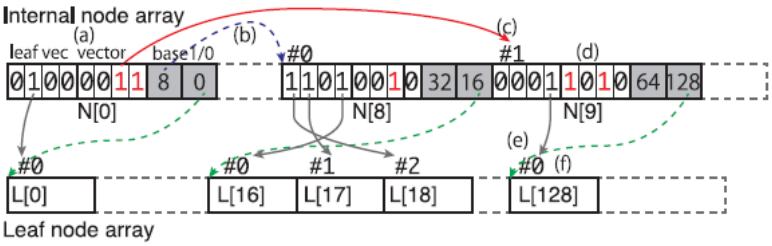
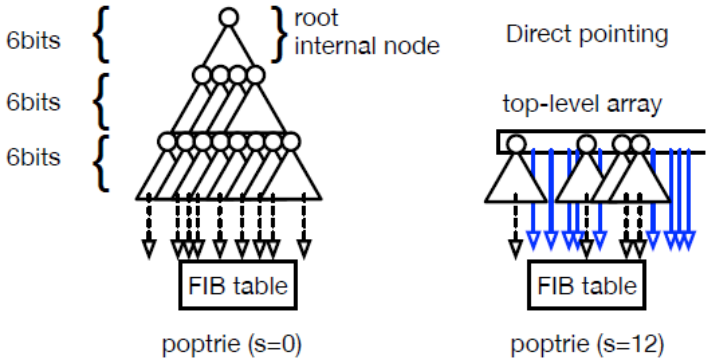


Figure 4: An example of the data structure of Poptrie where $k = 2$, and the lookup procedure for 0110b.

3.4 直接指向

尽管在内存空间上效率很高，但树结构通常不足以满足搜索速度，尤其是在我们的问题中。正如我们将在 4.1 节后面看到的，真实数据集中的大多数前缀分布在从/ 11 到/ 24 的前缀长度范围内。这意味着，对于大多数 IP 地址，任何树结构的查找算法总是需要遍历至少一些内部节点才能到达叶节点，导致一些昂贵的内存访问。如果我们使用额外的数组作为查找表，这个额外的过程可以省略，或者可以在 O（1）中完成，代价是更多的内存消耗。如今，通常采用这种优化技术;例子可以在 DIR-24-8-BASIC，DXR 和 SAIL 中看到。在 Poptrie 中，我们将这种优化称为“直接指向”（optimization\direct pointing），如图 5 所示。



(a) Without Direct Pointing. (b) Direct Pointing.

Figure 5: Direct pointing ($k = 6$, $s = 12$).

直接指向将与最高有效 s 位相对应的节点（内部或叶节点）抽取为长度为 2^s 的数组。这里，s 变量指定应该使用多少最重要的位作为数组的索引。该索引被称为“直接索引”（direct index），并且 key 地址 n 的最高有效 s 位的值被用作直接索引。它使我们能够通过访问顶层（top-level）数组中的第 n 个元素直接跳转到相应的 FIB 条目或内部节点。在我们的实现中，最高有效位指示直接索引是指向 FIB 条目还是内部节点; 如果被设置，则剩余的位构成直接指向 FIB 条目的索

引。 否则，直接索引指向内部节点并且需要进一步的搜索。由于我们使用了 4 字节长度的直接索引，因此它最多将内存占用量增加了 $4x2^s$ 个字节。 算法 1 的修改如算法 3 所示。

Algorithm 3 The direct pointing algorithm; the differences from Algorithm 1. Line 1 and 2 in Algorithm 1 are replaced with the statements below.

```
1: index = extract(key, 0, t.s);
2: dindex = t.D[index].direct_index;
3: if (dindex & (1UL ≪ 31)) then
4:     return dindex & ((1UL ≪ 31) - 1);
5: end if
6: index = dindex;
7: offset = t.s;
```

3.5 增量更新

尽管从头开始编译 Poptrie 的时间，即从 RIB 完全重建数据结构，是短的（如稍后在表 2 中所示，少于 70 毫秒），通常希望有一种方法来快速增量的更新 FIB。Poptrie 的增量更新是通过仅替换 trie 里的更新部分来执行的。

使用写入锁定阻止对 Poptrie 的读取访问是不可接受的，因为它会阻止 IP 转发过程相当长的时间。因此，我们为 Poptrie 中的增量更新选择无锁方法。无论如何，数据结构必须始终保持一致。这里的策略是让 IP 转发过程在构建更新的 FIB 的同时继续参考当前（即较旧的）FIB。更新完成后，通过使用原子指令更改 FIB 的指针或索引，将当前的 FIB 切换到新的 FIB。由于 FIB 查找是只读过程，并且我们假定单线程更新操作，所以原子指令可以确保一致性。

Poptrie 是提供从 RIB 编译的 FIB 的数据结构。假设 RIB 由二进制基数树维护。为了支持 Poptrie 的部分更新，我们给基数节点添加了一个标记（即一个 flag），表明这个节点需要更新。与标记的基数节点相对应的 Poptrie 的内部和叶节点被替换为新节点。Poptrie 数据结构的更新过程由以下三个步骤组成。 1）更新前缀时，通过遍历子树来标记每个基数树的下一跳改变的节点。 2）Poptrie 使用来自树最低层的新的后代内部节点和叶节点构造子树，重用对应于未标记的基数节点的内部节点和叶子。当一个 Poptrie 节点只包含一个覆盖所有范围的叶子，并且它不包含任何后代节点时，通过清除上层内部节点的向量中的相应位，将该节点移除并将叶子带到上一级。程序继续进行，直到叶子不能再被带到上层。 3）受影响的子树的根需要被替换。当 root 的 vector 和 leafvec 都不改变时，我们可以用一个原子指令将 root 节点数组（base1）或叶数组（base0）替换为新构建的数组。否则，当 root 的 vector 和（或）leafvec 发生变化时，我们会替换包含 root 节

点的整个节点阵列（换句话说，就是从 root 父节点指向的节点阵列）。如图 6 所示。我们为当前数组分配一个新的节点数组并重新构建它，最后使用原子指令替换父级的 base1。请注意，我们的伙伴内存分配器实现缓解了分配连续数组中的内存碎片。

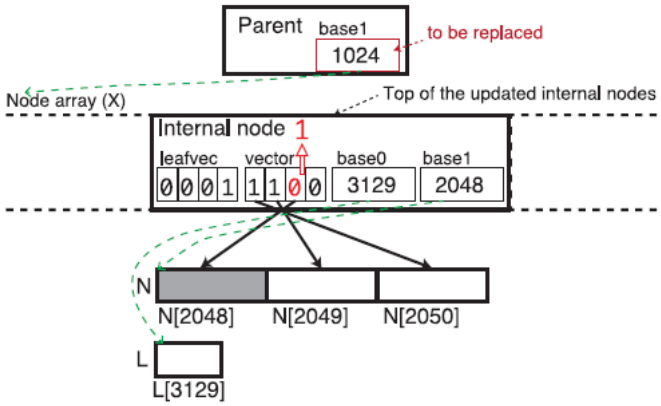


Figure 6: The lock-free update procedure to the internal node.

带有直接指向（direct pointing）的 Poptrie 的更新如下执行。如果基数树中顶部标记节点的深度等于或大于 s，则增量更新按照相同的方式进行，如上所述，不进行直接指向。如果顶部标记节点的深度小于 s，则需要替换 2^s 条目的整个顶层数组。

在完成更新过程之后，要释放未使用的存储空间（即被替换的部分）在确保没有查找过程引用它之后。

4、评估

我们使用 BGP 路由表评估所提出的查找算法的性能（在 4.1 节中描述）。我们比较 Poptrie 与二进制基数树、Tree BitMap [11]，DXR（D16R 和 D18R）[38] 和 SAIL（SAIL_L）[36]。我们自己实现了这些算法，并通过比较整个 IPv4 空间的每个地址的所有算法的所有查找结果来验证它们的正确性。为了与 Tree BitMap 和 Poptrie 进行公平比较，我们使用 popcnt 指令而不是原始 Tree BitMap 实现中使用的查找表。

我们使用配备有 Intel（R）Core i7 4770K（3.9 GHz，8 MiB 高速缓存）和 32 GB DDR31866 的计算机进行实验。 L1，L2，L3 缓存和 DRAM 访问的延迟分别为 4-5 个周期，12 个周期，36 个周期和 36 个周期加上列地址选通脉冲延迟时间（Column Address Strobe latency）。 L1，L2 和 L3 缓存的大小分别为 32 KiB，256 KiB 和 8MiB。除了第 4.6 节中的 CPU 周期分析之外，所有评估均在此计算机上

的 Ubuntu 14.04 服务器（x86_64）上执行十次。

4.1 数据集

表 1 总结了本文使用的路由表数据集。数据集名称以前缀“RV”开始，表示数据集是从 RouteViews 公共 BGP RIB 存档获得的[33]。我们通过只用一个下一跳过滤出数据集，或者路由表大小小于 500K，省略了与核心路由器的 RIB 不相似的数据集。获得 32 个数据集;数据集名称中的第二和第三项分别表示存档名称和对等号码。例如，RV-linx-p46 是 RouteViews 存档中 linx RIB 快照的第 46 个（从零开始编号）peer。

Table 1: RIB Datasets; the name, number of prefixes, and number of distinct next hops.

Name	# of prefixes	# of nhops	Name	# of prefixes	# of nhops	Name	# of prefixes	# of nhops
RV-linx-p46 †	518,231	308	RV-saopaulo-p12 ‡	516,536	510	RV-singapore-p3 †	518,620	136
RV-linx-p50 †	512,476	410	RV-saopaulo-p13 ‡	517,914	504	RV-singapore-p5 †	516,557	129
RV-linx-p52 †	514,590	419	RV-saopaulo-p16 †	521,405	528	RV-sydney-p0 †	520,580	122
RV-linx-p57 †	514,070	142	RV-saopaulo-p18 ‡	521,874	522	RV-sydney-p1 †	515,809	125
RV-linx-p60 †	508,700	70	RV-saopaulo-p2 ‡	523,092	530	RV-sydney-p3 †	517,511	115
RV-linx-p61 †	512,476	149	RV-saopaulo-p20 ‡	523,574	470	RV-sydney-p4 †	519,246	86
RV-nwax-p1 †	519,224	60	RV-saopaulo-p23 ‡	523,013	517	RV-sydney-p9 †	523,400	127
RV-nwax-p2 †	514,627	46	RV-saopaulo-p25 ‡	532,637	523	RV-telxatl-p3 ‡	511,161	56
RV-nwax-p5 †	519,195	49	RV-saopaulo-p26 ‡	516,408	479	RV-telxatl-p6 ‡	519,537	42
RV-paixisc-p12 †	519,142	68	RV-saopaulo-p8 ‡	522,296	477	RV-telxatl-p7 ‡	513,339	49
RV-paixisc-p14 †	524,168	49	RV-saopaulo-p9 ‡	515,639	507			
REAL-Tier1-A *	531,489	13	SYN1-Tier1-A	764,847	45	SYN2-Tier1-A	885,645	87
REAL-Tier1-B *	524,170	9	SYN1-Tier1-B	756,406	19	SYN2-Tier1-B	876,944	33
REAL-RENET ◇	516,100	32						

† Snapshot of 2014-12-17 00:00 UTC, ‡ Snapshot of 2014-12-16 23:00 UTC, ∗ Obtained on Jan. 9, 2015, ◇ Obtained on Jan. 3, 2015.

名称以“REAL”前缀开头的数据集是从运行中的 ISP 路由器获得的真实路由表。真实的和 RouteViews RIB 数据集之间的关键区别在于真实的包含通过内部网关协议（IGP）交换的路由。这些更长的前缀会使查找技术向下搜索树的更深层次，我们将在后面看到。REAL-Tier1-A 是从全球一级 ISP 中的真实核心骨干路由器获得的路由表。REAL-Tier1-B 从相同 ISP 的国家骨干路由器获得。REAL-RENET 是从 WIDE 项目的研究和教育网络中的路由器获得的[1]。

为了测试我们的技术对未来路由表增长的可扩展性，我们通过扩展 REAL-Tier1-A 和 REAL-Tier1-B 创建了两种类型的包含超过 700K 条目的合成路由表。第一种名称以'SYN1'开始的类型由以下过程创建：不超过/ 24 和/ 16 的每个前缀分别被分成两个和四个前缀。第二种类型的名称以“SYN2”开头，由以下过程创建：不超过/ 24，/ 20 和/ 16 的每个前缀分别被分成两个，四个和八个前缀。系统地为每个分割前缀分配不同的下一跳；第 i 个分割前缀具有下一跳 n + i，其中 n 是原始下一跳。请注意，下一跳 n + i 与原始数据集中的任何现有下一跳不重叠。这些合成数据集是更具挑战性的环境，因为它们具有更多的路由条目，并且由于从分配策略派生出不同的下一跳值，所以叶节点的聚合变得更加困难。

值得注意的是，一般来说，最长前缀匹配中检查的比特数大于单个路由的前缀长度。这是由于地址空间中前缀内的其他更长的前缀。我们称决定最长匹配前

缀“二进制基数深度”（binary radix depth）所需的比特数（相当于在二进制基数树中搜索的深度）。二进制基数深度可能比前缀长度更深，如图 7 所示。我们看到很多情况下需要进行更深入的搜索来决定较短的最长匹配前缀。例如，在许多情况下，需要搜索到第 24 级以确定匹配的前缀仅为/ 8。这会影响后面章节中显示的查找技术的性能。

4.2 流量模式

我们考虑以下流量模式来进行查找性能评估：**随机**，**顺序**，**重复**和**实时跟踪**。前三个是合成的，最后一个是真实的交通。

对于**随机**流量模式，使用 xorshift [22]生成 2^{32} 个随机 IP 地址。如果我们预先在内存中准备一个随机数组，数据结构可能会从缓存中推出。为了最小化这种缓存污染，每个随机数在使用 xorshift（仅分配四个 32 位变量）的查找例程之前生成。

测得的随机数发生器的平均开销是每次 1.22 纳秒。请注意，我们并未从结果中排除此开销。对于**顺序**，从 0.0.0.0 到 255.255.255.255 的 2^32 个地址被顺序查询。顺序表示具有空间和时间局部性的流量。由于不存在随机数生成技术，技术往往会显示更好的顺序性能，并且在搜索树的相同部分时，高速缓存命中的可能性更高。**重复**类似于随机，但每个随机数地址重复 16 次（总计 $16x2^{32}$ 查找）。它代表了具有高时间局部性的交通。

实时跟踪是一项真实的互联网流量跟踪[2]，于 2014 年 12 月 16 日截获，时间为 15 分钟。跟踪是在生成 REAL-RENET RIB 数据集的相同 AS 边界路由器的中转链路上捕获的。我们排除了使用大量实验性 ICMP 数据包探测整个 IPv4 地址空间的 IP 地址。这些数据包占了跟踪中 IPv4 数据包总数的 24％。此跟踪中（过滤后）的 IPv4 数据包数为 97,126,495，其中包含 644,790 个不同的目标 IPv4 地址。在评估中，我们先将所有目标 IP 地址加载到内存中的数组中，然后依次发布查询查询。

4.3 Poptrie 中扩展的效果

我们首先评估扩展和 Poptrie 的设计选项的有效性。它们被标记为“basic”（3.1 节），“leafvec”（3.3 节）和“s”（3.4 节中描述的直接指向的参数）。使用 REAL-Tier1-A，我们测量了内部节点（标记为“# of inodes”），叶节点数（“# of leaves”），内存占用量，从二进制基树构造 Poptrie 的编译时间以及平均查找性能。结果总结在表中 2。请注意，没有直接指向的 Poptrie 表示为 s = 0。

Table 2: The compilation time, the number of nodes, the memory footprint, and the lookup rate for random with direct pointing ($s = 0, 16, 18$).

Name and options	s	# of inodes	# of leaves	Mem. [MiB]	Compilation (std.) [ms]	Rate (std.) [Mlps]
Radix	–	–	–	30.48	–	8.82 (0.05)
Poptrie (basic) without route aggregation	0	64,009	4,032,568	8.67	31.07 (0.45)	87.71 (1.65)
	16	172,101	10,862,901	23.60	64.18 (0.33)	130.72 (1.72)
	18	61,282	3,911,422	9.40	36.06 (1.14)	170.69 (2.92)
Poptrie (*leafvec*) without route aggregation	0	64,009	280,673	2.00	32.60 (1.25)	89.15 (1.59)
	16	172,101	347,449	4.85	62.97 (0.20)	154.33 (1.53)
	18	61,282	265,320	2.91	33.37 (0.25)	191.95 (1.67)
Poptrie	0	43,191	263,381	1.49	32.84 (0.29)	96.27 (1.84)
	16	86,171	274,145	2.75	65.91 (0.35)	198.28 (5.29)
	18	40,760	245,034	2.40	33.24 (0.24)	240.52 (5.47)

使用 leafvec 的叶片压缩减少了 69-79％的内存占用（对于 s = 0，16，18），并且添加路由聚合选项可将内存占用减少 74-88％（与 Poptrie（basic）相比）。它们有效地减少了内存占用，从而提高了查找速度。s＝18 与 s＝0 的情况相比，在不增加编译时间的情况下，s＝18 时的查找率加倍，内存增长不到 1 MiB。此后，我们用 Poptriex 用 s＝x 来表示 Poptrie_x。

在本文中，我们选择 16 或 18 作为 s 值进行公平比较，因为 SAIL 和 DXR 扩展了这些比特长度。因为/ 24 前缀是 BGP 路由域中最长也是最大数量的前缀，所以 s＝18 的另一个原因是使 poptrie 节点正确对齐，前缀长度为/ 24。 s＝18 允许 Poptrie 为/ 24 前缀只需要一个内部节点遍历（对应于前进 6 位），而内存占用增加是可接受的。

4.4 多核的预期

现代 CPU 通常在单个 CPU 中实现多个内核。 因此，使用多个 CPU 内核评估性能仍然值得一提，以表明我们可以从单个 CPU 获得多少聚合性能。请注意，将数据包分发到多个 CPU 内核的调度体系结构超出了本文的范围。

图 8 显示了 Poptrie_18 对 REALTier1-A 和 REAL-Tier1-B 的线程数的聚合查找率。 显然，Poptrie 的数据结构可以在线程间共享，因此多线程不会增加内存占用。 此外，共享缓存具有足够的带宽来并行执行查找算法。 因此，Poptrie 的查找率可以线性扩展至 CPU 内核的数量。

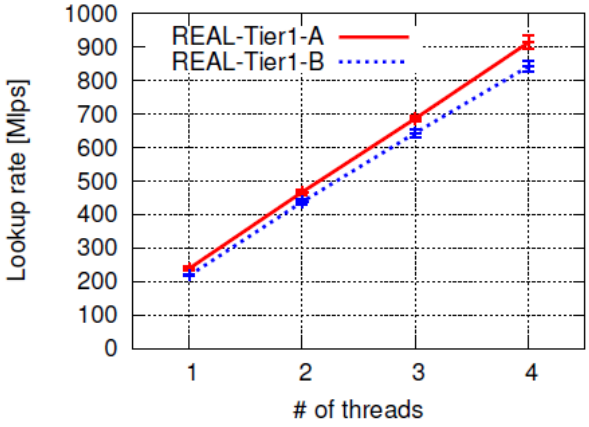


Figure 8: The aggregated lookup rate by the number of threads on the four core CPU.

4.5 与其他算法比较

在本节中，我们比较了 Poptrie 与 Tree BitMap [11]，SAIL [36]和 DXR [38]在三种综合流量模式下的性能；**随机**，**顺序**和**重复**，以展示 Poptrie 的优势。

我们首先比较**随机**流量模式的性能。图 9 显示了来自 RouteViews 和真实数据集的所有 35 个路由表实例的 Radix，Tree BitMap，SAIL，D16R，Poptrie_{16}，D18R 和 Poptrie_{18} 的查找性能。使用误差条显示 10 次实验的平均查找率和标准差（均方差）。对于所有这 35 个 RIB 数据集，Popperm 的平均查找率都优于所有其他查找算法；Poptrie_{18} 分别比 Radix，Tree BitMap，SAIL 和 D18R 快 24.5-46.1，3.52-6.78，1.37-2.62 和 1.04-1.34 倍。有五个 RIB 数据集，其中 Poptrie_{16} 优于 Poptrie_{18}（例如 RV-saopaulo-p2）。我们调查了，对于这些 RIB 数据集，路由聚合减少了大量比/ 16 更具体的前缀，因此 Poptrie_{16} 通过显着减少内存占用量实现了比 Poptrie_{18} 更好的性能。

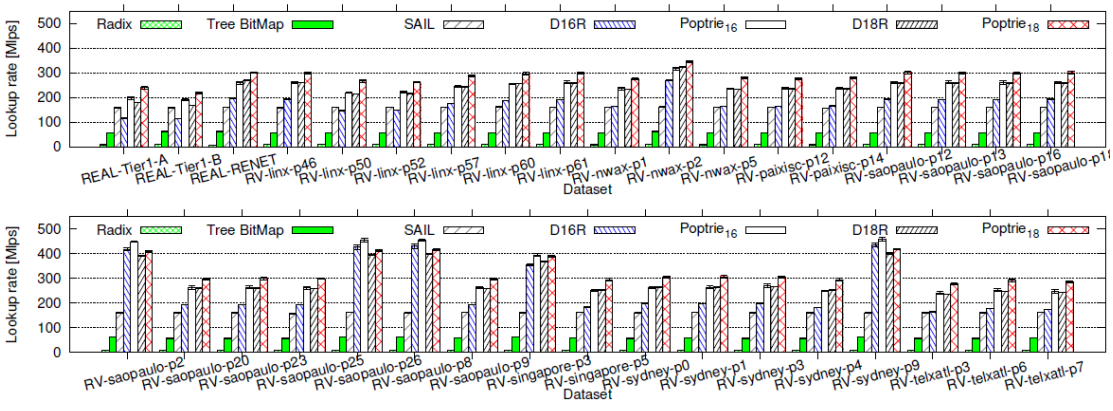


Figure 9: The average lookup rate for random IP addresses on RouteViews' tables.

在这里，我们仔细查看 REALTier1-A 和 REAL-Tier1-B 的结果，其结果是

Poptrie$_{18}$ 的最差和次最差的查找率。表 3 总结了这两个数据集的每种算法的内存占用和查找率。对于 REAL-Tier1-A 和 REAL-Tier1-B，Poptrie$_{18}$ 分别比 SAIL 快 1.52 和 1.37 倍。同样，对于 REAL-Tier1A 和 REAL-Tier1-B，Poptrie$_{18}$ 分别比 D18R 快 1.34 和 1.30。内存占用不是性能结果的唯一因素；即使 Poptrie$_{16}$ 和 Poptrie$_{18}$ 的内存占用量大于 DXR（D16R 和 D18R），它们仍然在 L3 缓存的大小之内，因此它们仍然可以胜过 DXR。相比之下，由于 SAIL 的内存占用量超过了 L3 缓存大小，因此会导致缓存未命中并导致相对较慢的 DRAM 访问。所以，SAIL 表现出比 Poptrie 慢的表现。表 3 还介绍了 Tree BitMap 的性能。使用 popcnt 指令而不是查找表可以使 Tree BitMap 增加多路树的顺序，以减少从原始 16 元到 64 元的查找深度。但是，如表 3 所示，即使 64 位 Tree BitMap 也无法实现良好的性能。我们怀疑这是因为在一个 Tree BitMap 节点中寻找一个匹配前缀需要在一个 2^k 多路树上运行 O(K)，而 Poptrie 在一个节点内找一个叶节点只需要运行 O(1)。

Table 3: The memory footprint and lookup rate for random of each algorithm.

	REAL-Tier1-A		REAL-Tier1-B	
Algorithm	Mem. [MiB]	Rate [Mlps]	Mem. [MiB]	Rate [Mlps]
Radix	30.48	8.82	29.34	8.92
Tree BitMap	2.62	56.24	2.54	62.13
Tree BitMap (64-ary)	3.10	61.61	2.89	68.82
SAIL	44.24	158.22	42.62	159.39
D16R	1.16	116.63	0.93	114.30
D18R	1.91	179.92	1.71	168.80
Poptrie$_0$	1.49	96.27	1.32	92.99
Poptrie$_{16}$	2.75	198.28	1.87	191.83
Poptrie$_{18}$	2.40	240.52	2.25	218.97

高局部性流量模式的查找性能也进行了比较。对于**顺序**来说，所有算法都有效地利用了 CPU 缓存，用于流量模式的高局部性。对于 Poptrie 表现较差的 REAL-Tier1-B， SAIL，D16R，D18R，Poptrie$_{16}$ 和 Poptrie$_{18}$ 对于顺序的平均查找率分别为 1264，628，911，955 和 1122 Mlps。 REALTier1-B 上 SAIL，D16R，D18R，Poptrie$_{16}$ 和 Poptrie$_{18}$ 对于重复的平均查找率分别为 492，382，454，470 和 480 Mlps。由于 SAIL 使用内存访问而不是执行某些指令来搜索树，所以当缓存命中率很高时，SAIL 可以达到高性能。Poptrie$_{16}$ 和 Poptrie$_{18}$ 比 SAIL 慢，因为我们的算法需要按位指令来查找相应 FIB 条目的索引，而 SAIL 直接访问连续数组。此外，即使在 CPU 缓存的有效利用中 DXR 也比 Poptrie 慢，因为 DXR 通过需要更多步骤的二进制搜索来搜索对应的 FIB 条目。

根据以上比较结果，Poptrie$_{18}$ 实现了高查找率，独立于流量模式和数据集，而 SAIL 的性能取决于流量模式的局部性。此外，对于流量模式和数据集的任何组合，Poptrie 都优于 DXR。

4.6 每次查找的 CPU 周期

为了进行详细分析，我们调查了每次查表所花费的 CPU 周期数。我们使用 REAL-Tier1-A 和 REAL-Tier1-B 作为数据集。我们使用我们正在开发的单任务操作系统（OS）来测量每个查找 CPU 周期。单任务操作系统使我们能够通过消除上下文切换的干扰和其他任务导致的缓存污染来精确测量 CPU 周期。由于缓存行为不能由 OS 控制，因此我们在大量查找中统计地分析 CPU 周期的分布情况。CPU 周期由 CPU 的性能监视计数器（PMC）监视[3]。读取 PMC 的开销一直是 83 个周期，不包括在结果中。对于每种算法，CPU 周期测量是针对 2^{24} 个随机 IP 地址查找进行的。请注意，对于随机数生成器，我们使用相同的种子来精确比较不同的算法。

REALTier1-A 数据集每次查找的 CPU 周期的 CDF（累积分布函数）如图 10 所示。我们看到当 CPU 周期数小于 120 时，D16R 和 Poptrie_{16} 以及 D18R 和 Poptrie_{18} 具有几乎相同的分布，除了 D16R 和 Poptrie_{16} 之间在 22 个周期处的小差异，这将在后面讨论。这是因为在查找较短的前缀时，DXR 和 Poptrie 的数据结构和查找算法几乎相同。D16R 和 Poptrie_{16} 的 CPU 周期数在 21-22 个周期比 D18R 和 Poptrie_{18} 周期的梯度更大。原因是 D16R 和 Poptrie_{16} 的第一次内存访问的内存占用空间分布在 256 KiB（4×2^{16}）的范围内，它可以完全包含在 L2 高速缓存大小中，而 D18R 和 Poptrie_{18} 用于第一个内存访问的内存占用空间分布在 1 MiB（4×2^{18}）的范围内，超出 L2 高速缓存大小并导致访问时间较慢。 SAIL 的 CPU 周期比 D16R 和 Poptrie16 的周期在 21-22 个周期显示出更陡的梯度。这是因为 SAIL 的顶层部分是 128 KiB（2×2^{16}），这是 L2 高速缓存大小的一半。因此，它可以利用 L2 缓存效率。但是，SAIL 的尾部分布扩大到较大的 CPU 周期，因为整个 SAIL 的内存占用量大于 L3 高速缓存大小，并导致 L3 高速缓存未命中。请注意，在 REAL-Tier1-B 数据集的每次查找的 CPU 周期的 CDF 中观察到类似的特征。

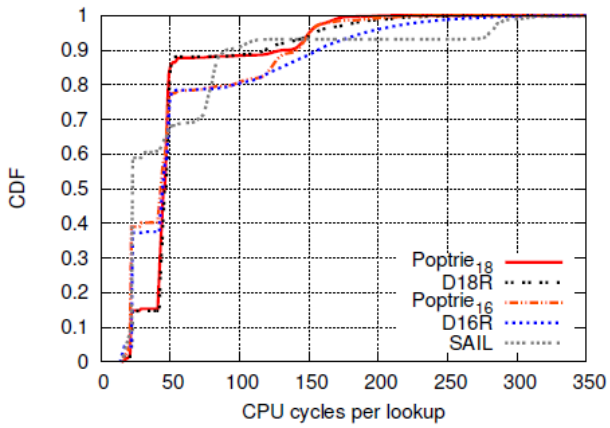


Figure 10: CDF of CPU cycles per lookup.

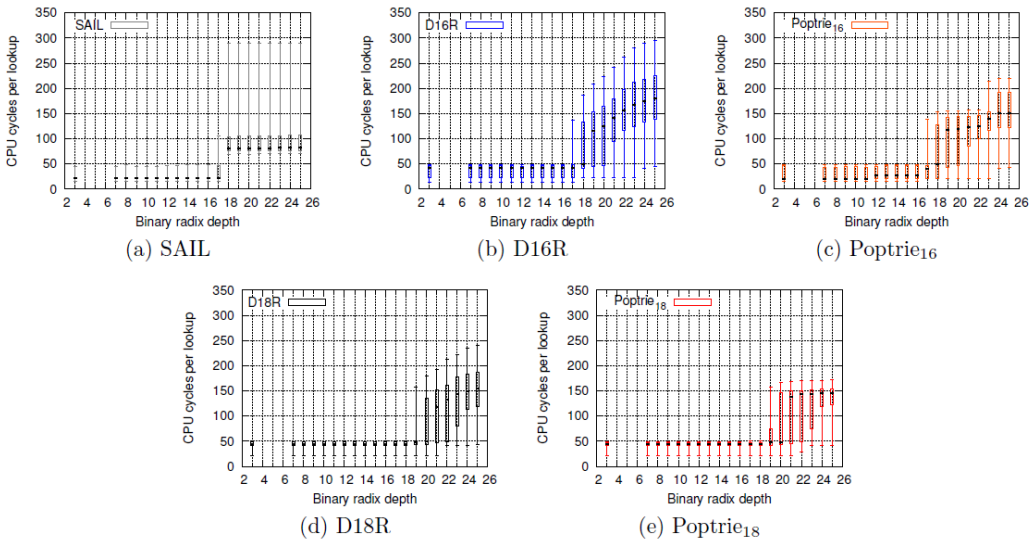
表 4 总结了 REAL-Tier1-A 和 REAL-Tier1B 上每种算法的每个查找 CPU 周期的平均值，第 50（中值），第 75，第 95 和第 99 百分点。第 n 个百分点值意味着百分之 n 的查找不会消耗比此值更多的 CPU 周期。第 95 和第 99 百分点的比较非常重要，因为它们表示除了特殊情况外查找性能的最坏情况保证。对于 REAL-Tier1-B，SAIL 在第 99 百分点消耗比 $Poptrie_{18}$ 多 124 个周期（74.7%）。这可以归因于缓存未命中和较慢的 DRAM 访问。第 99 百分点的 D18R 比 SAIL 好，但比 $Poptrie_{18}$ 大 21 个周期（12.7%）。这种差异归因于 DXR 中的二进制搜索阶段。

Table 4: The per-lookup CPU cycles by random traffic on REAL-Tier1-A and REAL-Tier1-B.

Dataset	Algorithm	Mean	50th	75th	95th	99th
REAL -Tier1-A	SAIL	57.43	22	76	279	299
	D16R	60.92	44	49	189	255
	D18R	54.84	46	48	154	207
	$Poptrie_{16}$	54.58	43	48	150	192
	$Poptrie_{18}$	53.59	46	48	150	169
REAL -Tier1-B	SAIL	56.34	22	75	279	290
	D16R	61.86	44	50	182	277
	D18R	56.88	47	49	154	187
	$Poptrie_{16}$	55.53	43	48	141	167
	$Poptrie_{18}$	55.82	46	48	150	166

虽然在 SAIL，DXR 和 Poptrie 中第一阶段（即 Poptrie 中的直接指向）是类似的，但当二进制基数深度大于 16 或 18 时，它们的行为应该不同，因为每个算法的第二阶段是不同的。因此，我们调查每个二进制基数深度的情况。图 11 显示了每个二进制基数深度的 CPU 周期分布。每烛台的烛芯代表第 5/95 百分点，身体代表第一四分位数（第 25%）和第三四分位数（第 75%），内部条形代表中位数值。这个数字证明了在更大的二进制基数深度上的显着差异；例如，对于任何二进制基数深度，$Poptrie_{18}$ 的第 95 个百分点不超过 172 个周期，而 SAIL 和 DXR 的在二进制基数深度 24 和 25 会超过 234 个周期。图 11 还表明，在一些二

进制基数深度上 DXR 实现比 Poptrie 更好的性能，当你查看 D18R 和 $Poptrie_{18}$ 在深度 21 的中值时。这是因为 DXR 的二进制搜索阶段可以在范围表中的前缀数量很小的情况下以较小的步数找到结果。总体而言，Poptrie 在各种情况下成功地保持了较低的 CPU 周期数，从而获得了卓越的性能。我们发现数据集 REAL-Tier1-B 也有类似的趋势。



(a) SAIL (b) D16R (c) $Poptrie_{16}$

(d) D18R (e) $Poptrie_{18}$

Figure 11: The quartiles and 5th/95th percentiles of per-lookup CPU cycles for each binary radix depth on REAL-Tier1-A.

当二进制基数深度小于 16 时，所有算法都保持 CPU 周期一直很小，小于 50。有趣的是，D16R 的中位数大于其他位数。如图 10 所示，我们也看到了 D16R 和 $Poptrie_{16}$ 在 22 个周期之间的分布差别很小。我们怀疑这可以归因于 DXR 的行为；对其范围表的二进制搜索会多次访问内存，以至于更小的二进制基数深度的数据结构难以保留在 L2 高速缓存中。

4.7 用真实的互联网流量跟踪进行性能评估

图 12 显示了 REAL-RENET 上实时跟踪的平均查找率。$Poptrie_{18}$ 分别比 Tree BitMap，D18R 和 SAIL 快 3.02，1.61 和 1.22 倍。此外，我们还确认 $Poptrie_{18}$ 比所有其他 RIB 数据集上的实时跟踪都优于 Tree BitMap，DXR（D16R 和 D18R）和 SAIL，尽管实时跟踪应该与其他 RIB 数据集上的真实流量不同。

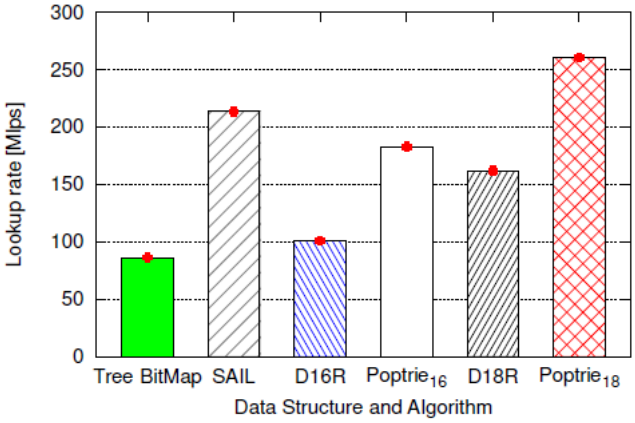


Figure 12: The average lookup rate for real-trace on REAL-RENET.

与随机数据相比，Poptrie 和 DXR 在实时跟踪中的查找率降低了。这是因为在实时跟踪中，大量的数据包通常去往比 BGP 路由更具体的 IGP 路由。32. 5％的 REAL-RENET 上实时跟踪数据包的二进制基数深度大于 18，而对于整个 IPv4 地址空间，只有 22.1%的二进制基数深度超过 18 个。这些地址无法在 Poptrie18 和 D18R 算法的第一阶段查看。此外，21.8％的实时跟踪数据包的二进制基数深度大于 24，而整个 IPv4 地址空间中只有 1.66％的二进制基数深度大于 24。

对于实时跟踪，SAIL 在查找率方面表现比随机表现更好。这是因为由于目的地 IP 地址的局部性，即具有相同目的地 IP 地址的分组序列，SAIL 可以利用 CPU 高速缓存的优势。

4.8 可扩展性

我们测量了合成 RIB（即那些带有“SYN”前缀）的性能，以评估未来路由表增长的可扩展性。由于其结构限制，SAIL 无法编译 SYN2-Tier1-A 和 SYN2-Tier1-B；SAIL 中的 $C_{16}[i]$ 被编码在 BCN [i]的 15 位中，但是这些数据集超过了 2^{15}。 DXR 也超过了最多支持到 2^{19} 的范围数量的结构性限制。但是，我们可以通过合并地址范围索引的“short”格式标志的一位来扩展到 2^{20}。因此，我们修改了 DXR 并进行了评估。第 5 节讨论了 Poptrie 的结构可伸缩性。

表 5 总结了合成 RIB 上随机流量模式的每种算法的平均查找率。$Poptrie_{18}$ 优于 SAIL 和 D18R，$Poptrie_{18}$ 的查找率超过 100 GbE 线速（即 148.8 Mlps），对于这些 RIB ，而对于 SYN2-Tier1-A，DXR 减速至 102.59 Mlps。因此，Poptrie 的查找性能对于路由表增长是可以扩展的。

Table 5: The lookup rates of each algorithm in Mlps for random traffic on synthetic large RIBs. The number of routes are parenthesized.

Algorithm	SYN1 -Tier1-A (764,847)	SYN1 -Tier1-B (756,406)	SYN2 -Tier1-A (885,645)	SYN2 -Tier1-B (876,944)
SAIL	102.86	99.98	N/A	N/A
D18R†	115.45	117.48	102.59	104.22
Poptrie_{18}	188.02	187.69	174.42	175.04

† modified

4.9 更新性能

我们还评估更新 Poptrie_{18} 数据结构的性能。首先对 RIB 维护的基数树执行更新，然后替换 Poptrie 中的部分树，如 3.5 节所述。我们使用 RV-linx-p52 的四个 15 分钟更新档案文件（即总共一小时）来评估更新性能。该数据集包含 7,424 条消息中的 23,446 条路由更新（公布 18,141 条，撤销 5,305 条）。顶层数组在直接指向，叶节点和内部节点每次更新时的平均替换数分别为 0.041，6.05（12.1 字节）和 0.48（11.52 字节）。这意味着一次更新只会替换数据结构中的少量对象。我们还测量了完成更新的时间；对于所有 23,446 次更新需要 58.90 毫秒，即每次更新只有 2.51 微秒。

作为更新性能评估的另一个输入数据，我们测量了路由表中完整路由的插入时间。请注意，条目的顺序是随机的，以消除对更新位置的任何假设。REAL-Tier1-A 和 REAL-Tier1-B 的平均插入时间分别为 2.71 和 2.40 秒，因此这些数据集每个前缀的平均插入时间分别为 5.10 和 4.57 微秒。所有这些结果都表明更新算法的复杂性实际上是可以接受的。

4.10 对 IPv6 的适用性

Poptrie 的一个优点是它足够普遍适用于 IPv6。为了评估，我们使用与 REAL-Tier1-A 相同路由器的 IPv6 路由表。由于 IPv6 路由表中的前缀数量并不大，因此对 IPv6 路由表的评估目前不如 IPv4 更让人感兴趣；数据集中只有 20,440 个前缀可用。表 6 总结了该数据集在 2000::/ 8 内的 2^{32} 个随机地址的性能。具有直接指向（s = 16,18）的 Poptrie 的查找率比 100 GbE 线速（即 148.8 Mlps）高。请注意，这个实验包含生成四个 xorshift 32 位随机数的不可忽略的开销，以生成一个 128 位随机地址。虽然最初引入了直接指向来优化 IPv4 查找性能，但即使在 IPv6 的情况下，直接指向（s = 16,18）的 Poptrie 通过利用 CPU 高速缓存而同时减少查找深度，获得了比没有直接指向（s = 0）的性能更高的性能。

Table 6: Poptrie's size, compilation time, and the performance on 2^{32} random lookups on the IPv6 routing table.

s	# of inodes	# of leaves	Mem. [KiB]	Compilation (std.) [ms]	Rate (std.) [Mlps]
0	14,925	32,586	414	7.22 (0.00)	138.51 (0.08)
16	16,554	33,047	709	4.77 (0.00)	209.84 (0.12)
18	14,910	32,569	1437	4.73 (0.00)	211.32 (0.09)

为了进行比较，我们扩展了 DXR 以支持 IPv6，方法是禁用“short”格式并将大小扩大一位，以允许每块大小为 2^{13} 个条目。我们无法将性能与 SAIL 进行比较，因为它不支持比/ 64 更具体的路由。对于 IPv6 数据集的随机流量，D16R 和 D18R 的平均查找率分别为 163.07 和 169.91 Mlps；Poptrie_{18} 比 D18R 快 1.24 倍。

我们还使用由 RouteViews 存档的当地时间 2014-12-25 00:00 的 13 个公共 RIB 来评估查找率，其包含超过 20K 个前缀和多个不同的下一跳。Poptrie_{16} 和 Poptrie_{18} 最差的平均查找率分别为 209.98 和 211.32 Mlps，仍然超过了 100 GbE 线速。总之，Poptrie 对于 IPv6 路由表也实现了高查找速率。

5 讨论

结构可扩展性：路由和下一跳的容量对于真实世界的部署非常重要。在 Poptrie 的实现中，叶节点的大小为 16 位，因此 FIB 条目的数量限制为 2^{16}，尽管我们可以简单地扩展大小而不是以更大的内存占用为代价。尽管由于空间限制我们无法提供详细信息，但我们估计了对内部节点，叶节点和下一跳的数量的限制，以及 Poptrie 可以分别支持 IPv4 和 IPv6 的一亿和七百万条路由的项目。这与我们在合成 RIB 评估中已经达到其限制的 DXR 和 SAIL 形成鲜明对比。

使用不同代 CPU 架构进行评估：我们确认 Poptrie 未针对本文中使用的一款 CPU 模型进行优化，进行性能评估使用的是另一代 CPU 架构（带有 8 MiB 高速缓存的 Intel（R）Xeon X3430 2.40 GHz）。对于来自 RouteViews 和真实网络的所有 35 个路由表中的随机流量模式，Popper 优于 SAIL 和 DXR;例如在 REAL-Tier1-A 上，Poptrie_{18} 分别比 D18R 和 SAIL 快 1.27 和 1.17 倍。

6、结论

我们从 64 路多路 trie 扩展提出了 Poptrie，用于在通用计算机上快速和可扩展的 IP 路由表查找。 Poptrie 利用位 1 计数指令为后代节点提供间接索引，以保

持 CPU 缓存内的小内存占用。在本文中，我们证明了 Poptrie 在核心路由器的 3 个专用路由表和 32 个 RouteViews 的公共 BGP 路由表上的随机和实际流量实验中胜过了现有的算法。 Poptrie 在小内存中存储了 1 级 ISP 路由表的 FIB，并且仅使用一个 CPU 内核就可以实现 241 Mlps 的随机流量查找性能。它适用于并行处理，并且对于随机流量使用四个 CPU 核心的速度可达 914 Mlps。CPU 周期分析揭示了每个 IP 路由表查找算法的特点，并展示了 Poptrie 对更长前缀的优势；$Poptrie_{18}$ 在较差的情况下需要显着较少的 CPU 周期，即第 95 个百分点，对于较长的前缀比其他情况。我们对未来预见的更大路由表的评估也证明了 Poptrie 在结构可伸缩性和查找性能方面的优势。 Poptrie 高效且可扩展，因此可以预期被用于需要最长前缀匹配的各种应用程序，例如 NFV 中的软件防火墙等。

致谢

我们感谢 Masafumi Oe 和 Rodney Van Meter 的持续和慷慨的支持。我们还要感谢匿名审稿人和我们的指导者 Luigi Rizzo 在我们的论文中提出的宝贵意见。

参考文献

略。