Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup

Hirochika Asai
The University of Tokyo
panda@hongo.wide.ad.jp

Yasuhiro Ohara
NTT Communications Corporation
yasuhiro.ohara@ntt.com

ABSTRACT

Internet of Things leads to routing table explosion. An inexpensive approach for IP routing table lookup is required against ever growing size of the Internet. We contribute by a fast and scalable software routing lookup algorithm based on a multiway trie, called *Poptrie*. Named after our approach to traversing the tree, it leverages the population count instruction on bit-vector indices for the descendant nodes to compress the data structure within the CPU cache. Poptrie outperforms the state-of-the-art technologies, Tree BitMap, DXR and SAIL, in *all* of the evaluations using random and real destination queries on 35 routing tables, including the real global tier-1 ISP's full-route routing table. Poptrie peaks between 174 and over 240 Million lookups per second (Mlps) with a single core and tables with 500–800k routes, consistently 4–578% faster than all competing algorithms in all the tests we ran. We provide the comprehensive performance evaluation, remarkably with the CPU cycle analysis. This paper shows the suitability of Poptrie in the future Internet including IPv6, where a larger route table is expected with longer prefixes.

CCS Concepts

•**Networks** → **Routers;** *Network algorithms;*

Keywords

IP routing table lookup; longest prefix match; trie

SIGCOMM '15, August 17-21, 2015, London, United Kingdom

ACM ISBN 978-1-4503-3542-3/15/08.

DOI: http://dx.doi.org/10.1145/2785956.2787474

1. INTRODUCTION

The fundamental functionality to support high speed communications on the Internet are increasing in importance with the ever-growing traffic size and due to our daily life being highly dependent on the Internet. One of the key technologies is IP routing table lookup: It needs to be extremely fast since the peak traffic size in an Internet core router is multiple hundreds of gigabits per second (Gbps). Ternary Content Addressable Memory (TCAM) performs high speed IP routing table lookup in the Internet core routers. However, departure from a TCAM is an approach worth considering for the two reasons: First, TCAM has issues in power consumption and heat. Second, the advent of Network Functions Virtualization (NFV) [6] may make the use of TCAM impossible, since the virtualized network functions are currently implemented in software without TCAMs. Therefore, it is desired to implement a software high-speed IP router only with general purpose computers; i.e., personal computers (PCs), or commercial off-the-shelf (COTS) devices.

For a long period of time, IP routing table lookup has been the bottleneck [11] in the performance of the software IP forwarding using COTS devices. It is a challenging problem [34, 31] because: 1) the size of the routing table is large and keeps growing (the number of BGP full routes is beyond 500K), 2) it requires a specific compute-intensive processing step called "longest prefix matching," and 3) high-speed communication links require high speed processing (148.8 million lookups per second (Mlps) for wire-rate IP packet forwarding on 100 Gigabit Ethernet (GbE) of minimum-size packets).

Recently, we see a significant improvement in the performance of software routers. There are two approaches; one is expecting the use of specific hardware such as graphics processing unit (GPU). Such hardware inherits, however, similar issues that TCAM has, such as heat and power consumption. The other is a pure software algorithm approach, where we assume the use of the commodity CPUs. A trend of this approach is reducing the memory footprint of the data structure of IP routing table to maximize the benefit of CPU cache [38,

25]. Still, the required performance to accommodate multiple of 100 GbE links has not been achieved.

We propose a novel data structure for fast and scalable IP routing table lookup, called *Poptrie*. It builds on the 64-ary multiway trie, allowing a low number of steps to search in the tree. It implements the descendant node array using a 64-bit vector, leading to the small memory footprint, and enabling a quick check of the descendant nodes. By the use of population count instruction on the bit vector, unnecessary descendant nodes are omitted efficiently, and a quick jump to the corresponding descendant node becomes feasible. Since Poptrie lays the descendant internal and leaf nodes in a contiguous array, the indirect index of smaller size is achieved, greatly reducing the memory footprint of the entire data structure. Poptrie also supports efficient incremental update without blocking the IP routing table lookup process.

We show, by the comprehensive evaluation in this paper, that Poptrie is promising with the friendliness to the larger routing table with longer prefixes of the future Internet, including IPv6. Compared to three state-of-the-art technologies, Poptrie gives superior performance in all evaluations for random and real destination queries on 35 instances of routing tables. It runs in 241 Mlps (1.34 times speed up from the fastest alternative, DXR [38]) even with a single CPU core on a real routing table instance of a global tier-1 ISP's core router, and it can achieve 914 Mlps with four CPU cores. Also, we analyze the consumed CPU cycles per lookup and showed the differences between other lookup technologies and Poptrie. One contribution of Poptrie is the scalability against future routing table growth. We show that Poptrie performs 175 Mlps on a synthetic table with more than 800K routes where DXR slows to 104 Mlps and SAIL [36] does not work.

The rest of this paper is organized as follows. In Section 2 we see related past studies. We describe the poptrie algorithm in Section 3. Section 4 gives the evaluation, where we describe the routing table dataset, how we generate the destination address for queries in the benchmark test, performance comparison, an analysis of the number of CPU cycles, scalability comparison for large routing tables, update performance evaluation, and performance comparison for the IPv6 routing tables. After miscellaneous discussions are given in Section 5, we conclude in Section 6.

2. RELATED WORK

TCAM has been a common technology for routing table lookup for a long time [23, 37, 39]. However, TCAM has problems with its power consumption, heat, monetary cost, and scalability issues [4, 16]. Bando et al. [4] proposed an FPGA-based routing table lookup engine, called FlashTrie, that can provide 200 Mlps per engine. Another approach to achieve fast IP routing table lookup is relying on GPUs [14]. Although GPU-based technologies like GPU-Click [32] and GAMT [21] are as fast as 500 Mlps, they require power-consuming GPUs, and need to process packets in large batch sizes. The large packet batch size is likely to lead to the higher worst case packet forwarding latency, and jitters.

Departing from dedicated hardware, there have been studies to achieve high performance IP routing on COTS devices. Click [19] is a modular software router that enables fast packet I/O (for the era), and RouteBricks [10] advanced the model further. Rizzo focuses on speeding up the packet I/O in Unix systems [26], in virtual machines [28], and in virtual switches [27]. The emergence of NFV [6] makes the fast IP routing lookup more important in producing a software router implementation with these fast packet I/O technologies on COTS devices.

The radix tree [29, 7, 18] and Patricia trie [24, 30] are fundamental data structures for the longest prefix match. In general, they require some tens of memory accesses for each IP routing table lookup, which result in low lookup performance. Waldvogel et al. [34] reduced the memory access both for IPv4 and IPv6 routing table lookup using binary search on prefix length. Gupta et al. [13] focused on the distribution of the prefix length in the routing table that most of the prefixes are no longer than `/24`. They proposed the DIR-24-8-BASIC data structure that extracts each `/24` or shorter prefix into `/24` prefixes to provide the lookup algorithm with O(1) for these entries.

There are studies that utilize bloom filters [9], memory pipelining [5, 15, 20], and bitmaps in the tree [11] to solve these problems, but these technologies fail to provide either a good performance or a reasonable management cost.

The Luleå algorithm [8] was proposed to reduce the memory footprint for the routing table. Srinivasan et al. [31] had taken into account data cache and optimized the data structure to improve the cache efficiency. Tree BitMap [11] provides a succinct data structure of a multiway trie. The prefix and the descendant nodes in a Tree BitMap node are represented by two bitmaps and the pointers to a contiguous array of data. It uses the population count operation in a similar way to Poptrie. In our tests, even in the most favourable situation Tree BitMap only achieved 1/3 of the performance of other modern algorithms. Section 4.5 discusses its perfromance in more detail. Rétvári et al. [25] proposed a compression algorithm of the forwarding information base (FIB) table to maximize the advantage of the data cache. However, a smaller FIB table does not always give a good lookup performance. For example, their algorithm consumes 194 CPU cycles per IPv4 address lookup, thus the lookup rate is only 12.8 Mlps. This is because the depth of their data structure can grow as much as 21. The approach proposed by Zec et al. [38], called DXR, achieved high lookup rate (100 Mlps per CPU core) by taking advantage of cache efficiency. DXR transforms the prefixes in the routing table

into an array of address ranges, and searches the range array based on the key address using the binary search. It then introduces a lookup table similar to DIR-24-8-BASIC to optimize the lookup performance for shorter prefixes. The bottleneck of their approach, however, is the binary search for longer prefixes.

Yang et al. have proposed yet another fast IP routing table lookup algorithm called SAIL [36]. Their approach reduces the number of memory access and instructions in the lookup algorithm by splitting the procedure into three levels. Yet, the memory footprint of SAIL exceeds the typical CPU cache size, requiring relatively slow DRAM access in case of cache misses. Therefore, the performance of SAIL relies on the destination IP address locality of the traffic pattern. Moreover, SAIL does not support future growth of the routing table due to its structural limitation as discussed in Section 4.8.

3. POPTRIE

We propose a new data structure called *Poptrie* for fast IP routing table lookup. We assume that Poptrie is only used to look up a FIB index for the purpose of deciding the next hop during the IP forwarding; the routes are preserved in a separate routing table (RIB: Routing Information Base) such as radix or Patricia trie so that we can aggressively compress routes having the same next hop. In this section, we describe how Poptrie works. In general, we can aggregate the routes that are ingested from the RIB to the FIB by removing the redundant prefixes that do not influence the lookup result. This aggregation, called in this paper the "route aggregation", is not our contribution, and is applicable to other lookup technologies as well. The route aggregation performs merger of a set of prefixes with the identical next hop that belong to a subtree without any gap, into the single prefix representing the whole subtree. Unless otherwise noted, the performance results of Poptrie shown in this paper are with the route aggregation option.

Poptrie is extended from the multiway trie (i.e., M-way or M-ary where $M = 2^k$). Each node holds 2^k elements in the descendant array, corresponding to the value of the k-bit chunk in the key IP address. An element in the descendant array points to its next-level child internal node or a leaf node holding an index to the corresponding FIB entry. Although we selected $k = 6$ for our implementation to fit the size of registers of the 64-bit CPU architecture, we illustrate the $k = 2$ case in this section for the sake of brevity. The 2^k-ary multiway trie where $k = 2$ is illustrated in Figure 1. The admirable performance of Poptrie is due to the small memory footprint so that it can be contained completely within the CPU cache, and yet it leverages the effective multiway branching to reduce the total number of steps that is necessary to search down the tree.

We describe the options in Poptrie step by step; the

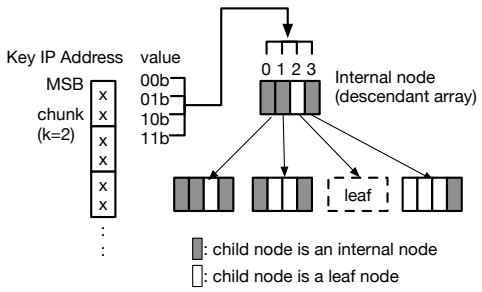


Figure 1: The 2^k-ary multiway trie ($k = 2$).

basic mechanism in Section 3.1, the lookup algorithm in Section 3.2, the *leafvec* extension to compress the leaf size in Section 3.3, and the additional options called direct pointing in Section 3.4. The internal node in the basic poptrie contains *vector* (8 bytes), *base0* (4 bytes), and *base1* (4 bytes). Hence the total size of an internal node is only 16 bytes. When we use the *leafvec* extension, it takes 8 bytes additionally, so the internal node size becomes 24 bytes. In our implementation, the contiguous arrays of internal and leaf nodes are managed by the buddy memory allocator [17].

3.1 Basic Mechanism

First, the descendant array in the multiway trie is changed to a bitwise array (i.e., a bit vector). The *vector* and the *base1* collectively serve as the descendant array. The *vector* is a bit-vector index of the length 2^k bits, for the current k-bit address chunk. The n-th bit in the *vector* corresponds to the child node with the value n in the current k-bit address chunk. Each bit in the *vector* indicates the type of the corresponding child node: the bit is set to 1 if the corresponding child is an internal node, and it is set to 0 if the corresponding child is a leaf node. In other words, the *vector* indicates the existence of the corresponding descendant internal node, and then if there is no descendant internal node, the search will always result in a leaf node at this level of the tree.

The necessary descendant internal or leaf nodes are placed so that they form a contiguous array. The array starts with the descendant node which corresponds to $n = 0$, in the ascending order up to $n = 63$ when $k = 6$. However, unnecessary nodes (i.e., descendant internal or leaf nodes that are not pointed from the node) are skipped; if the node corresponds to $n = 0$ is not necessary, the array can start with $n = 1$, with corresponding bits in *vector* properly set. This way, unnecessary descendant nodes which do not have branches or leaf information are omitted, allowing compact data structure size and efficient use of memory.

The next node in the search within the tree can be obtained as follows: Since the value n of the current k-bit address chunk corresponds to the n-th bit in the *vector*, the number of 1s in the least significant $n+1$ bits of the *vector* can be used as the index of the next node within the current internal node's descendant array. Here, we

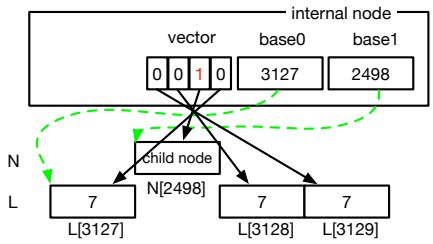


Figure 2: The *vector* in the internal node is configured so that the bit-1 indicates a descendant node, and the bit-0 indicates a leaf node.

can utilize the *popcnt* CPU instruction to accelerate the calculation of the next node in the search procedure, as described later in Section 3.2. Since *vector* bit-vector only provides the indirect address (i.e., index) within the current descendant array, it is necessary to provide the starting point of the descendant array. The *base1* is the base index to the consecutive subsequence of the internal nodes that are the children of this node. Similarly, the indirect index of the leaf node is obtained by counting the 0s in the *vector*. The starting offset for the leaf node is contained in *base0*. Figure 2 illustrates an example of the indirect index of internal and leaf nodes with *vector*, *base1*, and *base0*.

3.2 Lookup Algorithm

The lookup algorithm steps down the tree according to the specified IP address like the normal 2^k-ary multiway trie. At the depth of d, the d-th chunk of the address is used as the index of the *vector* in the internal node. Let the value of the d-th chunk in the key address be n, and then the lookup at the depth of d is executed as follows: If the corresponding bit is one, then the lookup algorithm continues to the next depth. The index of the next internal node in the descendant array is computed by adding to the *base1* the number of 1s in the least significant $n+1$ bits of the *vector* minus 1. If the corresponding bit is zero, then the lookup algorithm finishes the lookup with returning the leaf. The index of the leaf node in the leaf array is computed by adding to the *base0* the number of 0s in the least significant $n+1$ bits of the *vector* minus 1.

The key point in Poptrie is the use of the instruction to count the number of 1s and 0s in a bit string. Those counts are used as the indirect index of the descendant node and the leaf node, respectively. The procedure of counting the number of 1s in a bit string is called "population count", and an instruction executing it, *popcnt*, has been implemented in the x86 processor's instruction set. When the *popcnt* CPU instruction is not available, a fast alternative can be found in the literature [35].

The lookup algorithm with $k = 6$ is shown in Algorithm 1. The algorithm takes the poptrie structure t and the IP address *key* as input, and returns the longest matching leaf. In t, there are the internal node array N, and the leaf array L. In Line 1 the index is set to 0 to access the root node. Line 2 accesses to the *vector* of the root node. In Line 4, we obtain the value of the first 6-bit chunk from the offset 0. Lines 5–12 are the main loop that continues as long as there is a corresponding descendant internal node (checked in Line 5). Line 7 gets the population count of set bits in the least significant $v+1$ bits and store it in **bc**. The next node's index is calculated (Line 8), the next node's *vector* is prepared (Line 9), and the chunk is shifted by 6 bits for the next round (Line 10, 11). Lines 13–15 calculate the indirect index of the corresponding leaf, and returns the content.

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 Compression with the Leaf Bit-Vector

The prefix expansion described in the previous subsection yields many duplicate and redundant leaves. In the ordinary 2^k-ary multiway trie, an identical FIB entry corresponding to a shorter prefix may redundantly span to multiple leaves within an internal node, up to $2^k - 1$ leaves. For example when $k = 6$, an internal node can have a next hop of value A and 63 next hops of value B, in its 64-length leaf array. In this way, the redundant leaves (B in this example) can consume significant memory.

In order to avoid them, *leafvec* is introduced in the poptrie internal node. *Leafvec* is a bit vector that indicates the starting points of the identical, contiguous leaf ranges within the leaf array. It is a key technique to keep the memory footprint of Poptrie small, and reduces more than 90% of leaves as we will see in Section 4.3. The *leafvec* and *base0* collectively serve as the base and

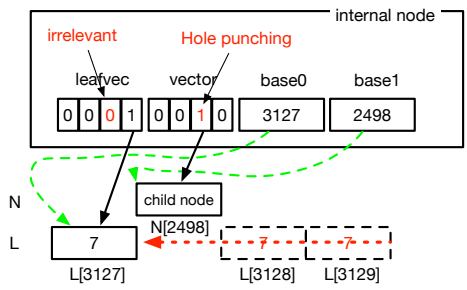


Figure 3: Merging identical leaf nodes with ignoring a hole punching using the *leafvec*

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 $\ll v$) - 1));

the indirect index to locate the leaf node, which is similar to the *vector* and *base1* described in the previous section. The indirect index using the *leafvec* and *base0* omits the redundant information as long as the redundant leaf slots are contiguous. For example, if all the 64 leaf slots in an internal node contains the same value, then it can be compressed to just one leaf slot with only the least-significant bit in the *leafvec* being 1. The indirect index for the leaf that corresponds to the value n for the current chunk is calculated as the number of 1s in the least-significant $n+1$ bits in the *leafvec*. This way, all the indirect indices for any value n fall into the first leaf slot, making the efficient memory compression.

Another benefit of this mechanism lies when *hole punching* occurs. Hole punching is an event such that a longer prefix divides a shorter prefix's address space, necessitating the routing table to deal with the address space as three distinct divisions[1]. Generally hole punching prevents the leaves from being contiguous, disabling the aforementioned efficient leaf compression. However, in Poptrie, the contiguity is recovered by making the leaf slot irrelevant if there is a descendant internal node that corresponds to the leaf slot. The lookup algorithm checks always the existence of the descendant internal node first, and if there is one, the lookup never tracks back from the lower level to the current level. Hence the leaf slot with a corresponding descendant internal node is defined as irrelevant, and is set to 0. Then, we may make the leaf slot contiguous again, ignoring those leaf slots with corresponding descendant internal node, as shown in Figure 3.

The modification in the algorithm is shown in Algorithm 2. Only the Line 14 is changed from Algorithm 1 so that it checks the newly introduced *leafvec* field to compute the corresponding leaf index. An example of the lookup procedure using *leafvec* is illustrated in Figure 4; an 8-bit address `01100111b` is searched as follows: (a) It takes the first two bits (`01b`) from the address, and then picks the bit corresponding to this index (the second bit from the right). (b) It finds the base address of the next subsequence of internal nodes using the *base1* member. (c) It counts the number of 1s in the least significant two bits of the *vector* member minus 1 (= 1), and finds the next internal node. (d) It takes the second two bits from the address (`10b`), and then picks the bit corresponding to this index (the third bit from the right). The bit is 0 in *vector*, so the search switches to find a leaf. (e) It finds the base address for the corresponding leaf nodes from *base0* (128). (f) It counts the number of 1s in the least significant three bits of the *leafvec* minus 1 (= 0), and finally finds the leaf node.

[1]Refer to Freedman et al. [12] for further definition of hole punching.

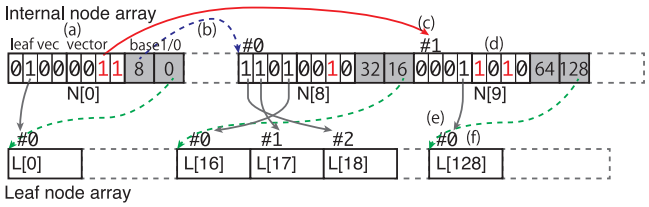
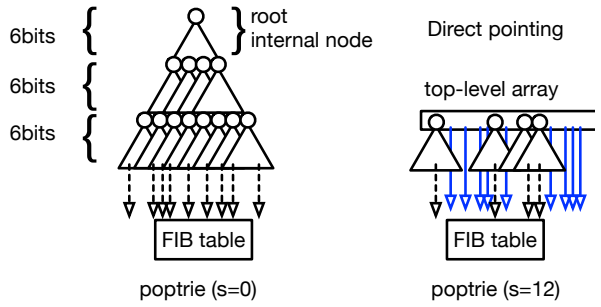


Figure 4: An example of the data structure of Poptrie where $k = 2$, and the lookup procedure for `0110b`.

3.4 Direct Pointing

Although efficient in memory space, the tree structure in general is not sufficient in search speed, especially in our problem. As we will see later in Section 4.1, most prefixes in the real datasets are distributed in the range of prefix length from `/11` through `/24`. This means that, for most IP addresses, the lookup algorithm of any tree structure will *always* need to traverse at least some internal nodes to reach to a leaf node, incurring some expensive memory accesses. This extra process can be omitted and alternatively be finished in $O(1)$ if we employ an additional array as a lookup table, at the expense of more memory consumption. These days, it is common to conduct such an optimization technique; examples can be seen in DIR-24-8-BASIC, DXR and SAIL. In Poptrie, we call this optimization "direct pointing". It is illustrated in Figure 5.

Direct pointing extracts the nodes (either internal or leaf) that corresponds to the most significant s bits to an array of length 2^s. Here, the s variable specifies how many most significant bits should be used as the index of the array. The index is called "direct index", and the value of the most significant s bits of the key address, n, is used as the direct index. It enables us to jump directly to the corresponding FIB entry or internal node, by accessing the n-th element in the top-level array. In our implementation, the most significant bit indicates whether the direct index points to a FIB entry or an internal node; if it is set, the remaining bits constitute



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

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

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;
```

an index that points to a FIB entry directly. Otherwise, the direct index points to the internal node and further search is necessary. Since we used the direct index of 4 bytes length, it increases the memory footprint by 4×2^s bytes at maximum. The modifications to Algorithm 1 are shown in Algorithm 3.

3.5 Incremental Update

Although compilation time of Poptrie from scratch, i.e., rebuilding the data structure entirely from the RIB, is short (less than 70 milliseconds as shown later in Table 2), it is generally desired to have a way to quickly update the FIB incrementally. The incremental update of Poptrie is performed by replacing only the updated part of the trie.

Blocking the read access to Poptrie using write lock is not acceptable because it blocks IP forwarding process for a considerable amount of time. Hence, we opt for a lock-free approach for the incremental update in Poptrie. In either way, the data structure must be kept consistent all the time. The strategy here is to let the IP forwarding process keep referring to the current (i.e., older) FIB while the construction of the updated FIB is ongoing. When the update is finished, the current FIB is switched to the new one, by changing the pointer or the index of the FIB using an atomic instruction. Since the FIB lookup is read-only procedure and we assume the single-threaded update operation, the atomic instruction can ensure the consistency.

Poptrie is a data structure to provide the FIB compiled from a RIB. Suppose the RIB is maintained by

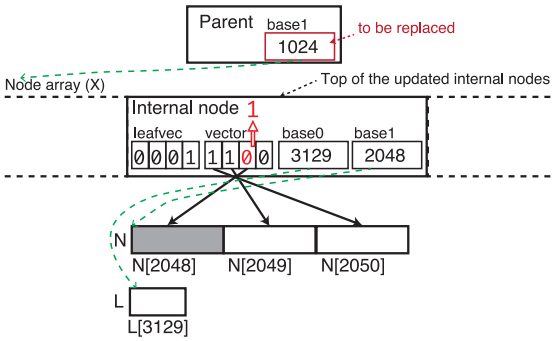


Figure 6: The lock-free update procedure to the internal node.

a binary radix tree. To support the partial update of Poptrie, we add a mark (i.e., a flag) to the radix node indicating that this node needs updating. The internal and leaf nodes of Poptrie corresponding to the marked radix nodes are replaced with new ones. The update procedure of the data structure of Poptrie consists of the following three steps. 1) When a prefix is updated, each radix tree's node of which next hop changes is marked by traversing the subtree. 2) Poptrie constructs the subtree with new descendant internal nodes and leaf nodes from the lowest level of the tree, reusing the internal nodes and leaves corresponding to the non-marked radix nodes. When a Poptrie node consists of only one leaf covering all the range, and it does not hold any descendant node, the node is removed and the leaf is brought to the upper level by clearing the corresponding bit in *vector* of the upper level internal node. The procedure continues until the leaves cannot be brought to the upper level any more. 3) The root of the affected subtree needs to be replaced. When neither of the root's *vector* nor *leafvec* change, then we can replace the root's node array (*base1*) or leaf array (*base0*) to a newly constructed array, with an atomic instruction. Otherwise, when the root's *vector* and/or *leafvec* change, we replace the entire node array that includes the root node (in other words, the node array that is pointed from the parent of the root). This is illustrated in Figure 6. We allocate a new node array for the current array and rebuild it, and finally *base1* of the parent is replaced with an atomic instruction. Note that our buddy memory allocator implementation mitigates the memory fragmentation in allocating a contiguous array.

The update of Poptrie with direct pointing is performed as follows. If the depth of the top marked node in the radix tree equals or is greater than s, the incremental update follows the same way without direct pointing, as described above. If the depth of the top marked node is less than s, the entire top-level array of 2^s entries needs to be replaced.

After the completion of the update procedure, the unused memory space, i.e., the replaced part, is freed after ensuring no lookup procedure is referring to it.

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.

4. EVALUATION

We evaluate the performance of the proposed lookup algorithm[2] using BGP routing tables (described in Section 4.1). We compare Poptrie with the binary radix tree, Tree BitMap [11], DXR (D16R and D18R) [38] and SAIL (SAIL_L) [36]. We implemented these algorithms ourselves, and validated their correctness by comparing all lookup results of all algorithms for each address of the whole IPv4 space. For fair comparison with Tree BitMap and Poptrie, we use the *popcnt* instruction rather than the lookup table used in the original Tree BitMap implementation.

We use a computer equipped with an Intel(R) Core i7 4770K (3.9 GHz, 8 MiB cache) and 32 GB DDR3-1866 for the experiments. The latencies[3] of L1, L2, L3 cache, and DRAM access are 4-5 cycles, 12 cycles, 36 cycles, and 36 cycles plus Column Address Strobe latency, respectively. The size of L1, L2, and L3 cache are 32 KiB, 256 KiB, and 8MiB, respectively. All evaluations are conducted ten times on the Ubuntu 14.04 server (x86_64) on this computer, except for the CPU cycle analysis in Section 4.6.

4.1 Datasets

Table 1 summarizes the routing table datasets we use in this paper. The dataset name starts with the prefix 'RV' indicates the dataset is obtained from the RouteViews public BGP RIB archives [33]. We omit datasets that do not resemble a core router's RIB by filtering out the datasets with only one next hop, or with routing table size less than 500K. 32 datasets are obtained; the second and the third segments in the dataset name represents the archive name and the peer number, respectively. For example, RV-linx-p46 is the 46th[4] peer in the linx RIB snapshot in the RouteViews archives.

The datasets whose name starts with the prefix 'REAL' are the real routing tables that are obtained from ISP's routers in operation. The key difference between the real and the RouteViews RIB datasets is that the real ones contain routes exchanged via Interior Gateway Protocols (IGPs). These longer prefixes cause the lookup technology to search down to a deeper level of the tree, as we will see later. REAL-Tier1-A is the routing table obtained from a real core backbone router in a global tier-1 ISP. REAL-Tier1-B is obtained from a national backbone router in the same ISP. REAL-RENET is obtained from a router in a research and educational network of WIDE Project [1].

To test the scalability of our technology against future routing table growth, we created two types of synthetic routing tables containing more than 700K entries, by extending REAL-Tier1-A and REAL-Tier1-B. The first type, whose name starts with 'SYN1', is created by the following procedure: Each prefix that is no longer than `/24` and `/16` is split into two and four prefixes, respectively. The second type, whose name starts with 'SYN2', is created by the following procedure: Each prefix that is no longer than `/24`, `/20`, and `/16` is split into two, four, and eight prefixes, respectively. Each split prefix is assigned a different next hop systematically; the i-th split prefix has the next hop $n + i$ where n is the original next hop. Note that the next hop $n + i$ did not overlap any existing next hops in the original datasets. These synthetic datasets are more challenging environments because they have a larger number of route entries, and the aggregation of leaf nodes becomes more difficult due to the distinct next hop values derived from the assignment policies.

It is worth noting that in general, the number of bits checked in the longest prefix matching is larger than the prefix length of individual routes. This is due to the other longer prefixes within the prefix in the address space. We call the number of bits (equivalently, the depth to search in the binary radix tree) that is neces-

[2]The reference implementation of Poptrie is available at https://github.com/pixos/poptrie.
[3]reported at http://www.7-cpu.com/cpu/Haswell.
[4]with zero-based numbering.

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)	0	64,009	4,032,568	8.67	31.07 (0.45)	87.71 (1.65)
without route aggregation	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*)	0	64,009	280,673	2.00	32.60 (1.25)	89.15 (1.59)
without route aggregation	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)

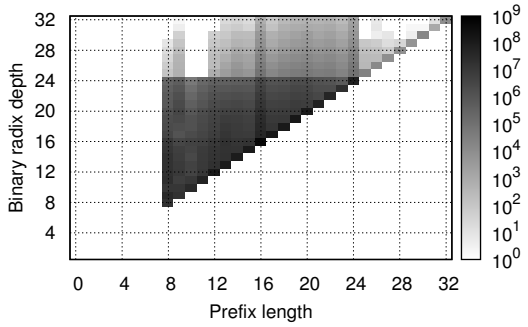


Figure 7: The heat map of the binary radix depth for all 2^{32} IPv4 addresses on REAL-Tier1-A. x-axis is the matched prefix's prefix length. y-axis is the length of checked bits.

sary to decide the resulting longest matching prefix "*binary radix depth*". The binary radix depth is possibly deeper than the prefix length as shown in Figure 7. We see a number of cases where deeper search is required to decide the shorter longest matching prefix. For example, there are many cases where it is necessary to search down to the 24th level to decide that the matching prefix is only `/8`. This influences the performance of lookup technologies shown in the later sections.

4.2 Traffic Patterns

We take the following traffic patterns into consideration for the lookup performance evaluation: **random**, **sequential**, **repeated**, and **real-trace**. The first three are synthetic ones, and the last one is real traffic.

For **random** traffic pattern, 2^{32} random IP addresses are generated using *xorshift* [22]. If we prepare an array of random numbers on memory in advance, the data structure may be pushed out from the cache. In order to minimize this cache pollution, each random number is generated just before the lookup routine using the xorshift, which allocates only four 32-bit variables. The measured average overhead of the random number generator was 1.22 nanoseconds per generation. Note that we did *not* exclude this overhead from the results. For **sequential**, 2^{32} addresses from `0.0.0.0` to `255.255.255.255` are queried sequentially. **Sequential** represents the traffic with spatial and temporal locality. Technologies tend to show better performance for **sequential** because of the absence of random number generation, and the higher possibility of cache hit in searching down the same part of the tree. **Repeated** is similar to **random** except that each random number address is repeated 16 times (total 16×2^{32} lookups). It represents the traffic with high temporal locality.

Real-trace is a real Internet traffic trace [2], captured on December 16, 2014, for 15 minutes. The trace was captured on a transit link of the same AS border router that produced the REAL-RENET RIB dataset. We excluded an IP address that probes the entire IPv4 address space with a large amount of experimental ICMP packets[5]. The packets accounted for 24.4% of the total IPv4 packets in the trace. The number of IPv4 packets in this trace (after the filtering) is 97,126,495 with 644,790 distinct destination IPv4 addresses. In the evaluation, we load all the destination IP addresses of **real-trace** into an array in memory in advance, and issue the lookup queries one by one in sequence.

4.3 Effect of Extensions in Poptrie

We first evaluate the effectiveness of the extensions and the design options of Poptrie. They are labeled "basic" (Section 3.1), "*leafvec*" (Section 3.3), and "s" (the parameter for direct pointing, described in Section 3.4). Using REAL-Tier1-A, we measured the number of internal nodes (labeled "# of inodes"), the number of leaf nodes ("# of leaves"), the memory footprint, the compilation time to construct Poptrie from the binary radix tree, and the average lookup performance. The results are summarized in Table 2. Note that Poptrie without direct pointing is represented as $s = 0$.

The leaf compression by the *leafvec* reduced the memory footprint by 69–79% for $s = 0, 16, 18$, and addition of the route aggregation option reduces the memory footprint by 74–88% compared to Poptrie (basic). They

[5]USC ANT project: http://www.isi.edu/ant/address/

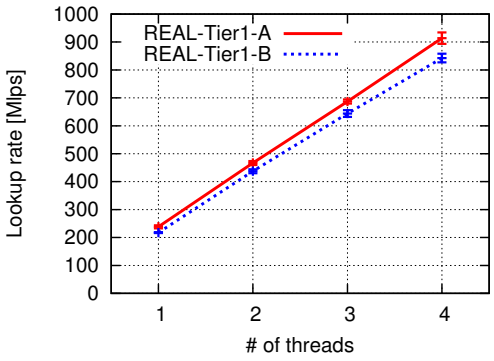


Figure 8: The aggregated lookup rate by the number of threads on the four core CPU.

effectively reduce the memory footprint, leading to the increased lookup rate. Poptrie with $s = 18$ doubles the lookup rate with less than 1 MiB memory increase compared to $s = 0$, without increasing the compilation time. Hereafter, we denote Poptrie with $s = x$ by Poptrie$_x$.

In this paper, we choose 16 or 18 as the value of s to conduct a fair comparison, since SAIL and DXR expand in those bit lengths. Another reason for $s = 18$ is to make the poptrie nodes aligned properly with the prefix length of `/24`, since the `/24` prefixes are the longest and also the largest number of prefixes in the BGP routing domain. $s = 18$ allows Poptrie to require only one internal node traversal (which corresponds to proceeding by 6 bits) for the `/24` prefixes, with an acceptable memory footprint increase.

4.4 Expectation in Multi-Core

Modern CPUs commonly implement multiple cores in a single CPU. Therefore, it is still worth evaluating the performance using multiple CPU cores, to indicate how much aggregated performance we can get from a single CPU. Note that the scheduling architecture to distribute packets into multiple CPU cores is beyond the scope of this paper.

Figure 8 presents the aggregated lookup rate of Poptrie$_{18}$ against the number of threads for REAL-Tier1-A and REAL-Tier1-B. Obviously the data structure of Poptrie can be shared among threads, and hence multithreading does not increase memory footprint. Furthermore, the shared cache has enough bandwidth to execute the lookup algorithm in parallel. Therefore, the lookup rate of Poptrie can be linearly scaled up to the number of CPU cores.

4.5 Comparison with Other Algorithms

In this section, we compare the performance of Poptrie with Tree BitMap [11], SAIL [36] and DXR [38], for three synthetic traffic patterns; **random**, **sequential**, and **repeated**, to demonstrate the advantage of Poptrie.

We first compare the performance for the **random** traffic pattern. Figure 9 shows the lookup performance of Radix, Tree BitMap, SAIL, D16R, Poptrie$_{16}$, D18R, and Poptrie$_{18}$ for all of the 35 routing table instances from the RouteViews and the real datasets. The average lookup rate and the standard deviation of ten experiments are shown using the error bars. Poptrie outperforms *all* other lookup algorithms in the average lookup rate for *all* of these 35 RIB datasets; Poptrie$_{18}$ is 24.5–46.1, 3.52–6.78, 1.37–2.62, and 1.04–1.34 times faster than Radix, Tree BitMap, SAIL, and D18R, respectively. There are five RIB datasets where Poptrie$_{16}$ outperforms Poptrie$_{18}$ (e.g., RV-saopaulo-p2). We investigated that, for these RIB datasets, the route aggregation reduced a large number of prefixes that are more specific than `/16`, and consequently, Poptrie$_{16}$ achieved better performance than Poptrie$_{18}$ through the significant reduction of the memory footprint.

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

Here, we take a close look at the results for REAL-Tier1-A and REAL-Tier1-B, which yielded the worst and second worst lookup rate of Poptrie$_{18}$. Table 3 summarizes the memory footprint and the lookup rate of each algorithm for these two datasets. Poptrie$_{18}$ is 1.52 and 1.37 times faster than SAIL for REAL-Tier1-A and REAL-Tier1-B, respectively. Similarly, Poptrie$_{18}$ is 1.34 and 1.30 times faster than D18R for REAL-Tier1-A and REAL-Tier1-B, respectively. The memory footprint is not the only factor for the performance results; even though the memory footprint of Poptrie$_{16}$ and Poptrie$_{18}$ is larger than DXR (D16R and D18R), they are still within the size of the L3 cache, and therefore, they can still outperform DXR. In contrast, because the memory footprint of SAIL exceeds the L3 cache size, it leads to the cache misses and to the relatively slow DRAM access. Consequently, SAIL exhibited slower performance than Poptrie. Table 3 also presents the performance of Tree BitMap. The use of the *popcnt* instruction instead of a lookup table enables Tree BitMap to increase the order of a multiway trie to reduce the lookup depth, from original 16-ary to 64-ary. However, as shown in Table 3, even the 64-ary Tree BitMap cannot achieve good performance. We suspect that this is because searching a matching prefix within a Tree

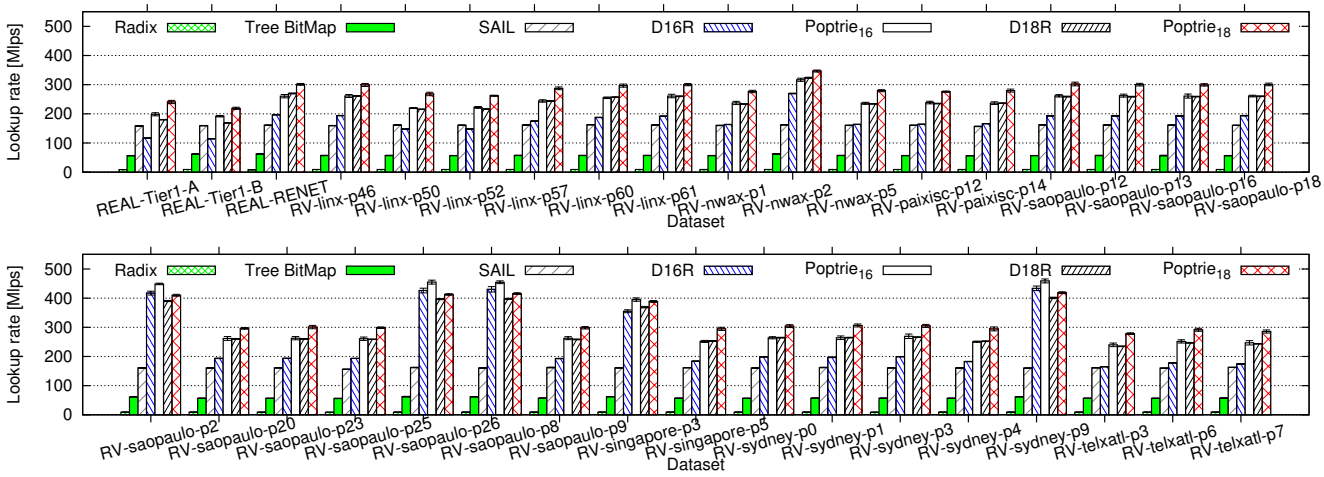


Figure 9: The average lookup rate for random IP addresses on RouteViews' tables.

BitMap node runs in O(k) on a 2^k-ary multiway trie while Poptrie searches a leaf within a node in O(1).

The lookup performance for the high locality traffic patterns is also compared. For **sequential**, all algorithms effectively utilized the CPU cache for the traffic pattern's high locality. For REAL-Tier1-B where Poptrie performed worse, the average lookup rate for **sequential** of SAIL, D16R, D18R, Poptrie$_{16}$, and Poptrie$_{18}$ were 1264, 628, 911, 955, and 1122 Mlps, respectively. The average lookup rate for **repeated** of SAIL, D16R, D18R, Poptrie$_{16}$, and Poptrie$_{18}$ on REAL-Tier1-B were 492, 382, 454, 470, and 480 Mlps, respectively. Since SAIL uses the memory access instead of executing some instructions to search down the tree, SAIL achieves high performance when the cache hit rate is high. Poptrie$_{16}$ and Poptrie$_{18}$ are slower than SAIL because our algorithms require bitwise instructions to find the index to the corresponding FIB entry, while SAIL directly accesses to a contiguous array. Moreover, DXR is slower than Poptrie even in the effective utilization of the CPU cache because DXR searches the corresponding FIB entry with the binary search that requires more number of steps.

From the comparison results above, Poptrie$_{18}$ achieves the high lookup rates, independent of the traffic patterns and datasets, while the performance of SAIL depends on the locality of traffic patterns. Furthermore, Poptrie outperforms DXR for any combination of traffic pattern and dataset.

4.6 CPU Cycles per Lookup

In order to conduct the detailed analysis, we investigate the number of CPU cycles it takes for each table lookup. We use REAL-Tier1-A and REAL-Tier1-B for the dataset. We measured the per-lookup CPU cycles with the single task operating system (OS) that we are developing. The single task OS enables us to precisely measure the CPU cycles by eliminating the disturbances

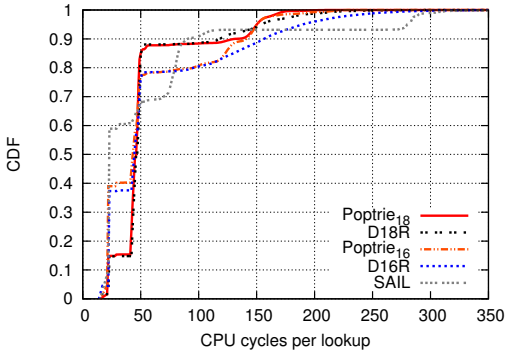
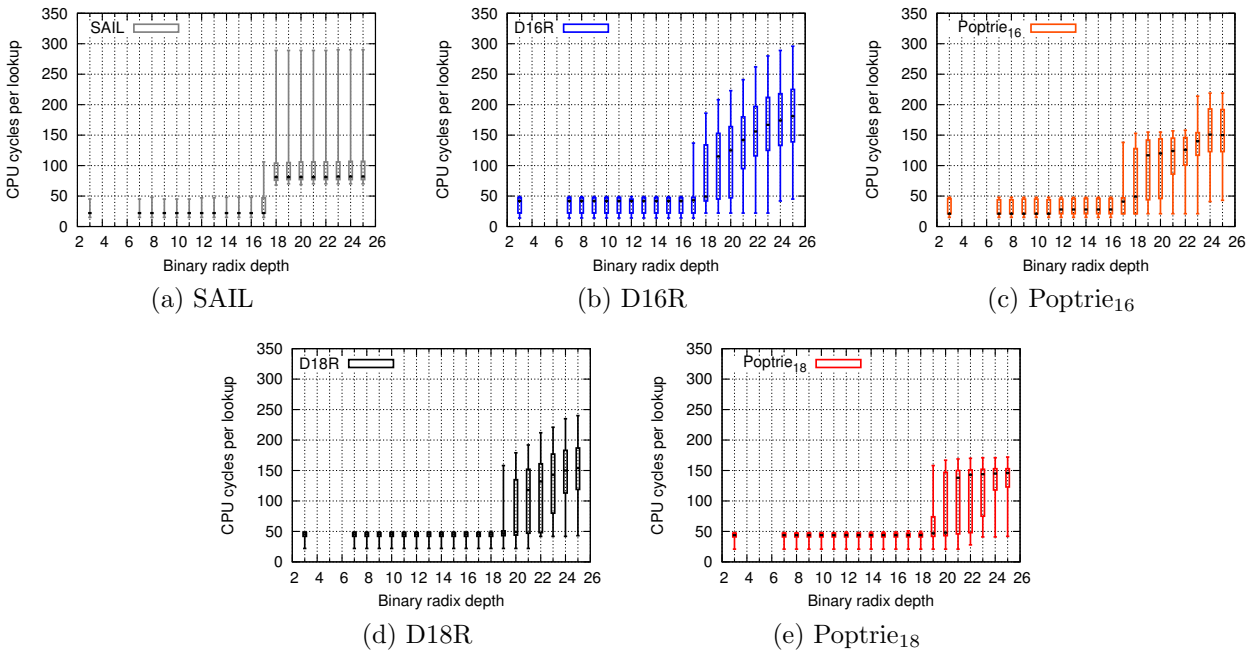


Figure 10: CDF of CPU cycles per lookup.

of context switches and the cache pollution caused by other tasks. Since the cache behavior cannot be controlled by the OS, we statistically analyze the distribution of the CPU cycles in a large number of lookups.

The CPU cycles are monitored with the performance monitoring counters (PMCs) of the CPU [3]. The overhead to read a PMC is constantly 83 cycles, and is excluded from the results. The CPU cycle measurement was conducted for 2^{24} **random** IP address lookups, for each algorithm. Note that we used the same seed for the random number generator to precisely compare different algorithms.

The CDF of CPU cycles per lookup for the REAL-Tier1-A dataset is shown in Figure 10. We see that D16R and Poptrie$_{16}$, and D18R and Poptrie$_{18}$ have almost identical distributions where the CPU cycle count is less than 120, except for a small difference around at 22 cycles between D16R and Poptrie$_{16}$, which will be discussed later. This is because the data structures and lookup algorithms of DXR and Poptrie are almost the same in looking up shorter prefixes. The CPU cycle count of D16R and Poptrie$_{16}$ exhibit the steeper gradients around 21–22 cycles than those of D18R and



(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.

Poptrie_{18}. The reason is that the memory footprints of D16R and Poptrie_{16} for the first memory access are distributed in the range of 256 KiB (4×2^{16}), which can be entirely contained in the L2 cache size, while the memory footprints of D18R and Poptrie_{18} for the first memory access are distributed in the range of 1 MiB (4×2^{18}), which exceeds the L2 cache size and incurs the slower access time. The CPU cycles of SAIL exhibits an even steeper gradient around 21–22 cycles than those of D16R and Poptrie_{16}. This is because the top level part of SAIL is 128 KiB (2×2^{16}), which is half of the L2 cache size. Consequently, it could take advantage of the L2 cache efficiency. However, the tail distribution of SAIL is expanded to the larger CPU cycles because the memory footprint of entire SAIL is larger than the L3 cache size and it leads to the L3 cache misses. Note that the similar characteristics were observed in the CDF of CPU cycles per lookup for the REAL-Tier1-B dataset.

Table 4 summarizes the mean, 50th (median), 75th, 95th, and 99th percentiles of per-lookup CPU cycles for each algorithm on REAL-Tier1-A and REAL-Tier1-B. The n-th percentile value means that n percent of lookups consumed no more CPU cycles than this value. The comparisons of the 95th and 99th percentiles are important because they indicate the worst case guarantees of the lookup performance except for corner cases. For the REAL-Tier1-B, SAIL consumes 124 cycles (74.7%) more than Poptrie_{18} at their 99th percentiles. This can be attributed to the cache misses and slow DRAM access. The 99th percentile of D18R is better than SAIL, but it is 21 cycles (12.7%) larger than that of Poptrie_{18}. This difference is attributed to the binary search stage in DXR.

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

Although the first stage (i.e., the direct pointing in Poptrie) is similar in SAIL, DXR, and Poptrie, when the binary radix depth is more than 16 or 18, their behavior should differ, since the second stage of each algorithm is different. Hence, we investigate each case per binary radix depth. Figure 11 shows the CPU cycle distribution for each binary radix depth. The wick of each candlestick represents 5th/95th percentile, the body represents the first and third quartile values, and the internal bar represents the median value. This figure demonstrates a significant difference at greater binary radix depth; for example, the 95th percentiles of Poptrie_{18} are no more than 172 cycles for any binary radix depth while those of SAIL and DXR exceed 234 cycles at the binary radix depth of 24 and 25. Fig-

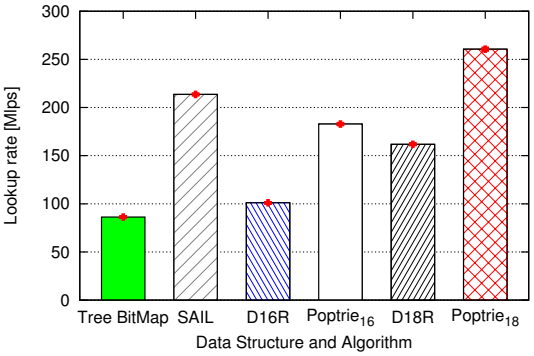


Figure 12: The average lookup rate for real-trace on REAL-RENET.

ure 11 also presents that DXR achieves better performance than Poptrie at some binary radix depths, when you look into the median values of depth of 21 in D18R and Poptrie$_{18}$. This is because the binary search stage of DXR can find the result with a smaller number of steps in cases that the number of prefixes in a range table is small. Overall, Poptrie succeeded in maintaining the lower number of CPU cycles in various cases, gaining superior performance. We found the similar trend also for the dataset REAL-Tier1-B.

When the binary radix depth is smaller than 16, all algorithms kept the CPU cycles consistently small, less than 50. Interestingly, the median of D16R is larger than that of the others. As shown in Figure 10 before, we also see the small difference in the distributions between D16R and Poptrie$_{16}$ around 22 cycles. We suspect that this can be attributed to the DXR's behavior; the binary search for its range table accesses memory many times so that the data structure for smaller binary radix depth are harder to stay in the L2 cache.

4.7 Performance Evaluation with a Real Internet Traffic Trace

Figure 12 shows the average lookup rate for **real-trace** on REAL-RENET. Poptrie$_{18}$ is 3.02, 1.61 and 1.22 times faster than Tree BitMap, D18R and SAIL, respectively. Additionally, we also confirm that Poptrie$_{18}$ outperforms Tree BitMap, DXR (D16R and D18R) and SAIL for **real-trace** on all the other RIB datasets although **real-trace** should be a different pattern from the real traffic on the other RIB datasets.

The lookup rates of Poptrie and DXR for **real-trace** are degraded compared to those for **random**. This is because a larger number of packets goes to IGP routes that are generally more specific than BGP routes in **real-trace**. 32.5% of the packets in **real-trace** on REAL-RENET have the binary radix depth more than 18, while for the whole IPv4 address space only 22.1% have the binary radix depth more than 18. These addresses cannot be looked up in the first stage of the algorithm of Poptrie$_{18}$ and D18R. Moreover, 21.8% of the packets of **real-trace** have binary radix depth more than 24, while only 1.66% of the whole IPv4 address space have binary radix depth more than 24.

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

SAIL performs better in the lookup rate for **real-trace** than for **random**. This is because SAIL could take advantage of the CPU cache due to the locality of the destination IP addresses, i.e., the sequences of packets with the identical destination IP address.

4.8 Scalability

We measure the performance on the synthetic RIBs (i.e., those with 'SYN' prefix) to evaluate the scalability to future routing table growth. SAIL cannot compile SYN2-Tier1-A and SYN2-Tier1-B due to its structural limitation; $C_{16}[i]$ in SAIL is encoded in the 15 bits of $BCN[i]$, but it exceeds 2^{15} for these datasets. The DXR also exceeds its structural limitation of the number of ranges that is supported up to 2^{19}. However, we can extend it to 2^{20} by absorbing one bit for the "short" format flag to the address range index. Thus, we modified DXR and conducted the evaluation. The structural scalability of Poptrie is discussed in Section 5.

The average lookup rates of each algorithm for the **random** traffic pattern on the synthetic RIBs are summarized in Table 5. Poptrie$_{18}$ outperforms SAIL and D18R, and the lookup rate of Poptrie$_{18}$ exceeds the 100 GbE wire-rate, 148.8 Mlps, for these RIBs, while DXR slows down to 102.59 Mlps for SYN2-Tier1-A. Thus, Poptrie is scalable to the routing table growth in lookup performance.

4.9 Update Performance

We also evaluate the performance of updating the Poptrie$_{18}$ data structure. The update is first performed to the radix tree for the RIB maintenance, and then replaces a part of the trie in Poptrie, as described in Section 3.5. We use four 15 minute update archive files (i.e., an hour in total) of RV-linx-p52 to evaluate the update performance. This dataset contains 23,446 route updates (18,141 announced and 5,305 withdrawn) in 7,824 messages. The average number of replacements for the top-level array in direct pointing, the leaf node, and the internal node, per update, are 0.041, 6.05 (12.1 bytes) and 0.48 (11.52 bytes), respectively. This means that an update replaces a small number of objects in the data structure. We also measured the time to complete

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)

the update; 58.90 milliseconds for all 23,446 updates, i.e., only 2.51 microseconds per update.

As another input data for update performance evaluation, we measured the insertion time of the full route in a routing table. Note that the order of the entries is randomized to eliminate any assumptions on the locality of updates. The average insertion time for REAL-Tier1-A and REAL-Tier1-B are 2.71 and 2.40 seconds, respectively, hence the average insertion time per prefix for these datasets are 5.10 and 4.57 microseconds, respectively. All these results demonstrate that the complexity of the update algorithm is practically acceptable.

4.10 Applicability to IPv6

One advantage of Poptrie is that it is general enough to apply to IPv6. For the evaluation, we use the IPv6 routing table from the same router as REAL-Tier1-A. The evaluation on IPv6 routing tables is currently less interesting than IPv4 because the number of prefixes in IPv6 routing table is not large; only 20,440 prefixes are available in the dataset. Table 6 summarizes the performance on this dataset for 2^{32} random addresses within `2000::/8`. The lookup rate of Poptrie with direct pointing (s=16,18) achieves more than the wire-rate of 100 GbE, i.e., 148.8 Mlps. Note that this experiment includes the non-negligible overhead of four xorshift 32-bit random number generation to generate a 128-bit random address. Although direct pointing was originally introduced to optimize the IPv4 lookup performance, Poptrie with direct pointing (s=16,18) achieves the higher performance than that without direct pointing (s=0), by taking advantage of CPU cache while reducing the lookup depth, even in the IPv6 case.

For the comparison, we extend DXR to support IPv6 by disabling the "short" format and extending the *size* by one bit to allow up to 2^{13} entries per chunk. We cannot compare the performance with SAIL because it does not support more specific routes than `/64`. The average lookup rates of D16R and D18R for random traffic with the IPv6 dataset are 163.07 and 169.91 Mlps, respectively; Poptrie_{18} is 1.24 times faster than D18R.

We also evaluate the lookup rate using 13 public RIBs archived at 2014-12-25 00:00 local time by RouteViews that contains more than 20K prefixes and more than one distinct next hops. The worst average lookup rates of Poptrie_{16} and Poptrie_{18} are 209.98 and 211.32 Mlps, respectively, still exceeding the 100 GbE wire-rate. In summary, Poptrie also achieves high lookup rate for IPv6 routing tables.

5. DISCUSSION

Structural scalability: The capacity of routes and next hops are important for real world deployment. In the implementation of Poptrie, the size of a leaf node is 16 bits, hence the number of FIB entries is limited to 2^{16}, though we can simply extend the size at the expense of larger memory footprint. Although we cannot provide the details due to space limitation, we estimate the limitation on the number of internal nodes, leaf nodes, and next hops, and project that Poptrie can support a hundred million and 7 million routes for IPv4 and IPv6, respectively. This is in contrast to DXR and SAIL which already reached at their limitations in our synthetic RIB evaluations.

Evaluation with a different generation of CPU architecture: We confirm that Poptrie is not optimized for one CPU model used in this paper through the performance evaluation with another generation of CPU architecture, Intel(R) Xeon X3430 2.40 GHz with 8 MiB cache. Poptrie outperforms SAIL and DXR for the **random** traffic pattern on all of the 35 routing tables from RouteViews and real network; for example on REAL-Tier1-A, Poptrie_{18} is 1.27 and 1.17 times faster than D18R and SAIL, respectively.

6. CONCLUSION

We proposed Poptrie extended from the 64-ary multiway trie for fast and scalable IP routing table lookup on general purpose computers. Poptrie leverages the population count instruction to give the indirect indices to the descendant nodes in order to keep the small memory footprint within the CPU cache. In this paper, we demonstrated that Poptrie outperformed the existing algorithms for random and real traffic from experiments on three private routing tables of core routers and 32 RouteViews' public BGP routing tables. Poptrie stored a FIB of the tier-1 ISP routing table in small memory footprint, and achieved 241 Mlps lookup performance for random traffic, using just one CPU core. It is suitable for parallel processing, and exhibited as fast as 914 Mlps using four cores of the CPU for random traffic. The CPU cycle analysis revealed the characteristics of each IP routing table lookup algorithm, and demonstrated the advantage of Poptrie for longer prefixes; Poptrie_{18} requires significantly less CPU cycles in the worse case, i.e., the 95th percentile, for longer prefixes than the others. Our evaluations on the future-envisioned larger routing tables also demonstrated the superiority of Poptrie in the structural scalability as well as the lookup performance. Poptrie is efficient and scalable so various applications using longest prefix matching such as software firewall in NFV can be expected.

Acknowledgment

We are grateful to Masafumi Oe and Rodney Van Meter for their continuous and generous support. We also thank the anonymous reviewers, and our shepherd, Luigi Rizzo, for their invaluable comments on our paper.

7. REFERENCES

[1] WIDE Project. http://www.wide.ad.jp/.

[2] WIDE Project MAWI Working Group Traffic Archive. http://mawi.wide.ad.jp/mawi/.

[3] Intel(R) 64 and IA-32 Architectures Software Developer Manuals (version 052), Sept. 2014.

[4] M. Bando, Y.-L. Lin, and H. J. Chao. FlashTrie: Beyond 100-Gb/s IP Route Lookup Using Hash-based Prefix-compressed Trie. *IEEE/ACM Trans. Netw.*, 20(4):1262–1275, 2012.

[5] A. Basu and G. Narlikar. Fast Incremental Updates for Pipelined Forwarding Engines. *IEEE/ACM Trans. Netw.*, 13(3):690–703, June 2005.

[6] M. Chiosi et al. Network Functions Virtualisation – Introductory White Paper. *ETSI*, 2012.

[7] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[8] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. In *ACM SIGCOMM*, pages 3–14, 1997.

[9] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest Prefix Matching Using Bloom Filters. *IEEE/ACM Trans. Netw.*, 14(2):397–409, Apr. 2006.

[10] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *ACM SOSP*, pages 15–28, 2009.

[11] W. Eatherton, G. Varghese, and Z. Dittia. Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates. *ACM SIGCOMM Comput. Commun. Rev.*, 34(2):97–122, 2004.

[12] M. J. Freedman, M. Vutukuru, N. Feamster, and H. Balakrishnan. Geographic Locality of IP Prefixes. In *ACM IMC*, pages 13–13, Berkeley, CA, USA, 2005.

[13] P. Gupta, S. Lin, and N. McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *IEEE INFOCOM*, volume 3, pages 1240–1247, 1998.

[14] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *ACM SIGCOMM*, pages 195–206, 2010.

[15] J. Hasan and T. N. Vijaykumar. Dynamic Pipelining: Making IP-lookup Truly Scalable. In *ACM SIGCOMM*, pages 205–216, 2005.

[16] W. Jiang, Q. Wang, and V. Prasanna. Beyond TCAMs: An SRAM-Based Parallel Multi-Pipeline Architecture for Terabit IP Lookup. In *IEEE INFOCOM*, pages 1786–1794, 2008.

[17] K. C. Knowlton. A Fast Storage Allocator. *Commun. ACM*, 8(10):623–624, Oct. 1965.

[18] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 2nd edition, 1998.

[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[20] S. Kumar, M. Becchi, P. Crowley, and J. Turner. CAMP: Fast and Efficient IP Lookup Architecture. In *ACM/IEEE ANCS*, pages 51–60, 2006.

[21] Y. Li, D. Zhang, A. X. Liu, and J. Zheng. GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers. In *ACM/IEEE ANCS*, pages 1–12, 2013.

[22] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[23] A. McAuley and P. Francis. Fast Routing Table Lookup using CAMs. In *IEEE INFOCOM*, volume 3, pages 1382–1391, 1993.

[24] D. R. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968.

[25] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger. Compressing IP Forwarding Tables: Towards Entropy Bounds and Beyond. In *ACM SIGCOMM*, pages 111–122, 2013.

[26] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, pages 9–9, 2012.

[27] L. Rizzo and G. Lettieri. VALE, a Switched Ethernet for Virtual Machines. In *ACM CoNEXT*, pages 61–72, 2012.

[28] L. Rizzo, G. Lettieri, and V. Maffione. Speeding Up Packet I/O in Virtual Machines. In *ACM/IEEE ANCS*, pages 47–58, 2013.

[29] R. Sedgewick. *Algorithms in C, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching*. Addison-Wesley Professional, 3rd edition, 1997.

[30] K. Sklower. A Tree-Based Packet Routing Table for Berkeley Unix. In *USENIX Winter Conference*, pages 93–104, 1991.

[31] V. Srinivasan and G. Varghese. Fast Address Lookups Using Controlled Prefix Expansion. *ACM Trans. Comput. Syst.*, 17(1):1–40, 1999.

[32] W. Sun and R. Ricci. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *ACM/IEEE ANCS*, pages 25–36, 2013.

[33] University of Oregon. Route Views Project. http://www.routeviews.org/.

[34] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *ACM SIGCOMM*, pages 25–36, 1997.

[35] H. S. Warren. *Hacker's delight*. Pearson Education, 2013.

[36] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee IP Lookup Performance with FIB Explosion. In *ACM SIGCOMM*, pages 39–50, 2014.

[37] F. Zane, G. Narlikar, and A. Basu. CoolCAMs: Power-Efficient TCAMs for Forwarding Engines. In *IEEE INFOCOM*, volume 1, pages 42–52, 2003.

[38] M. Zec, L. Rizzo, and M. Mikuc. DXR: Towards a Billion Routing Lookups Per Second in Software. *ACM SIGCOMM Comput. Commun. Rev.*, 42(5):29–36, 2012.

[39] K. Zheng, C. Hu, H. Lu, and B. Liu. A TCAM-based Distributed Parallel IP Lookup Scheme and Performance Analysis. *IEEE/ACM Trans. Netw.*, 14(4):863–875, Aug. 2006.