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

Hirochika Asai(The University of Tokyo)

Yssuhiro Ohara(NTT Communications Corporation)

ACM SIGCOMM 2015, Aug. 18, 2015

Authors- Hirochika Asai

- A project assistant professor at the University of Tokyo.
- Received his PhD degree from the University of Tokyo in 2013.
- Networking Operating System
- Distributed System and Architecture
- Internet Topology and Traffic Measurement and Analysis
- More than 10 papers

Authors- Yssuhiro Ohara

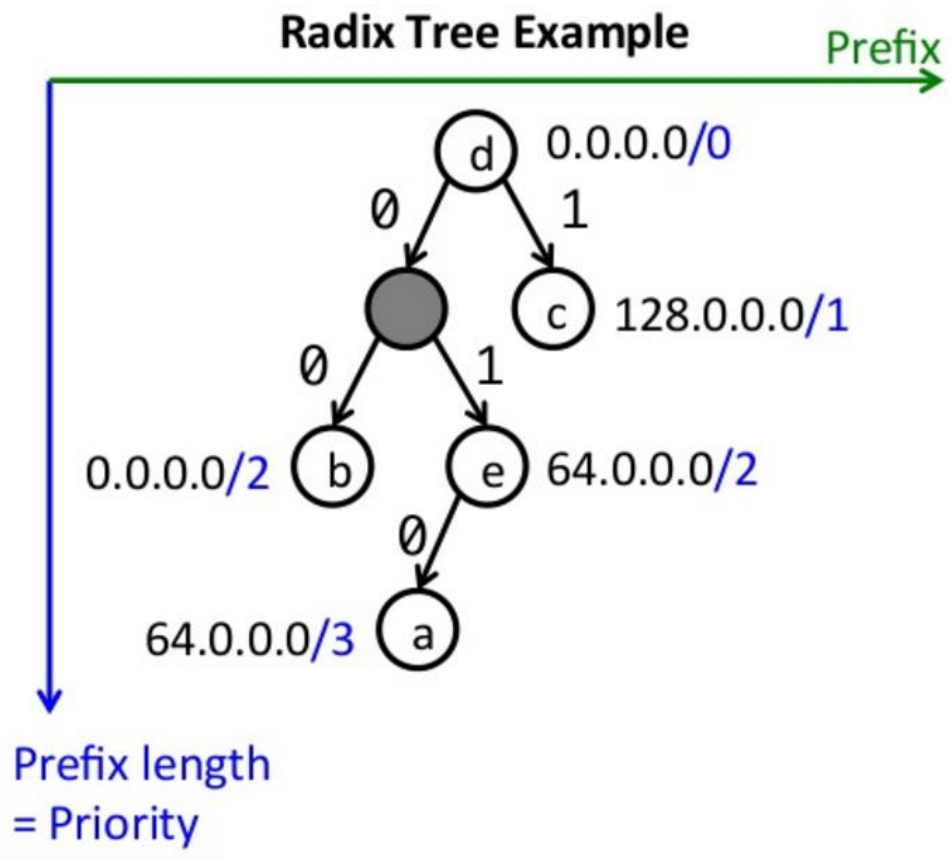
- NTT Communications Corporation
- Received his PhD from Keio University, Japan in 2008
- Distributed System and Architecture
- Storage system

IP Routing Table Lookup

- Principle
  - Longest prefix match
- Challenges
  1. Large and growing routing table
     - IPv4: 500K+ entries
     - IPv6: 20K+ entries
  2. High lookup rate requirement
     - e.g. 148.8Mpps <=> 6.7 ns per lookup
       (at the wire-rate on 100 GbE of minimum-size frames)

Radix Tree(Revisited)

Fundamental data structure and algorithm for longest prefix match



Problems with the radix tree

- Depth up to 32(for IPv4)
  - requires a number of memory access
- Large memory footprint due to pointers
  - causes CPI cache misses

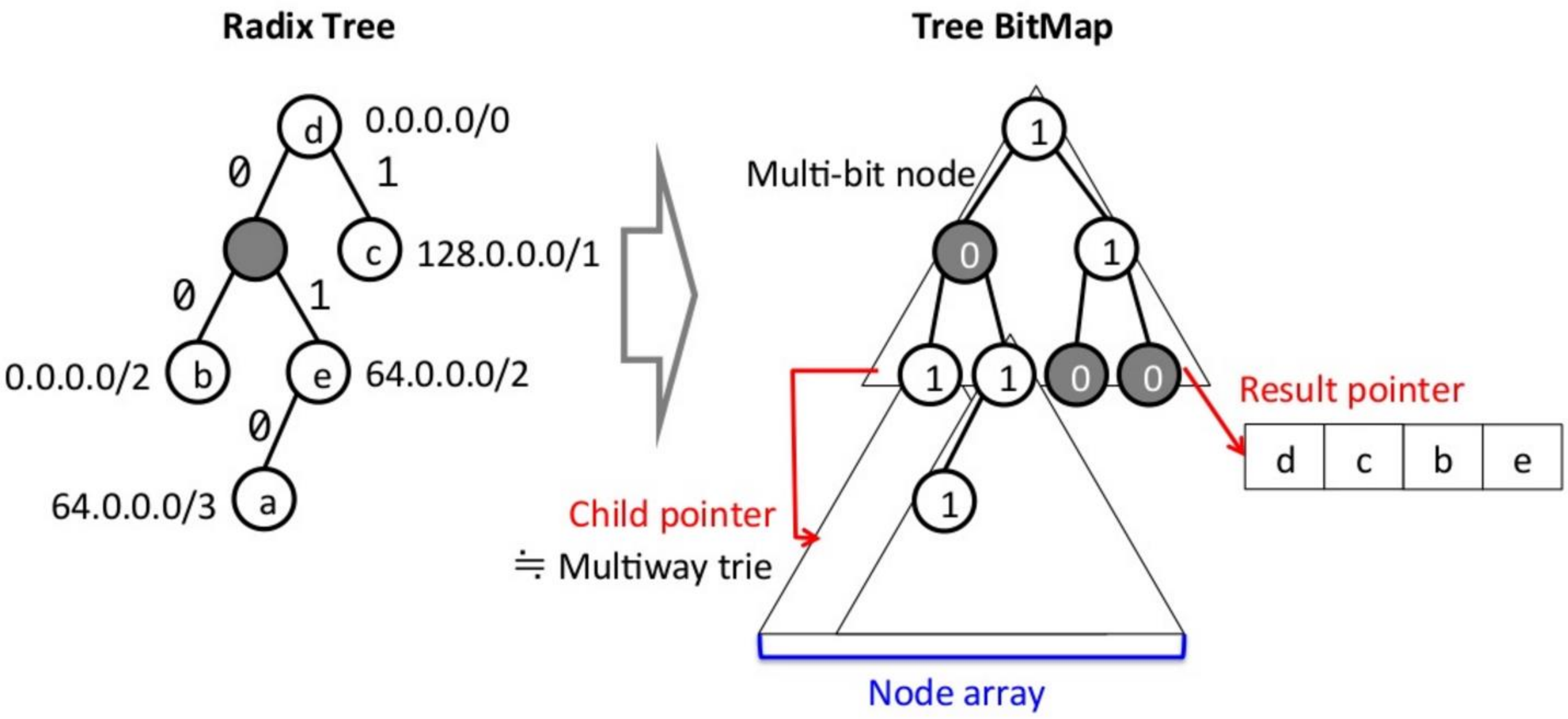
➡ Bad performance

Related Work: Fast IP Routing Table Lookup Algorithms

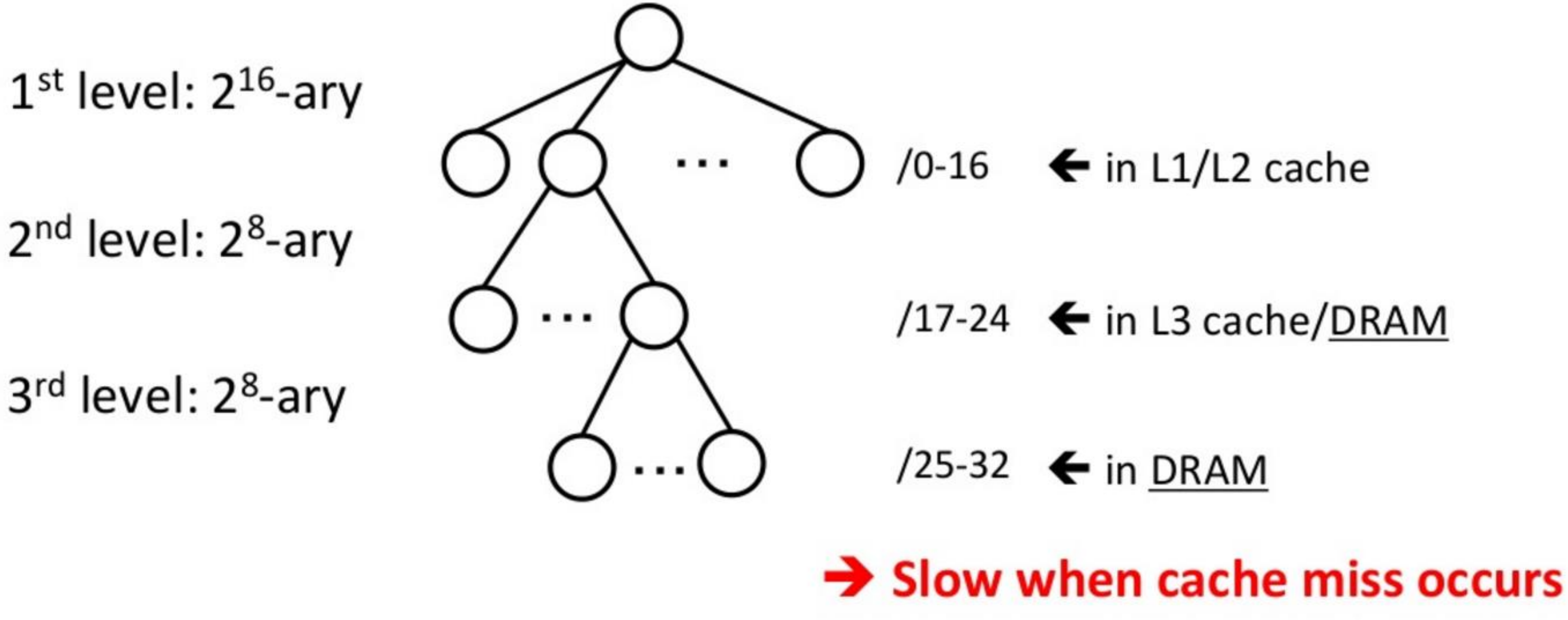
	Approach	Feature
Tree BitMap [1]	Trie (partly multiway)	Succinct data structure within CPU cache size
SAIL [2]	Trie (multiway)	Optimized multi-level trie (3-level for IPv4)
DXR [3]	Range	Small memory footprint and high L1 cache efficiency

[1] W. Eatherton et al., "Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates," ACM SIGCOMM CCR, 2004
[2] T. Yang et al., "Guarantee IP Lookup Performance with FIB Explosion," ACM SIGCOMM 2014
[3] M. Zec et al., "DXR: Towards a Billion Routing Lookups Per Second in Software," ACM SIGCOMM CCR, 2012

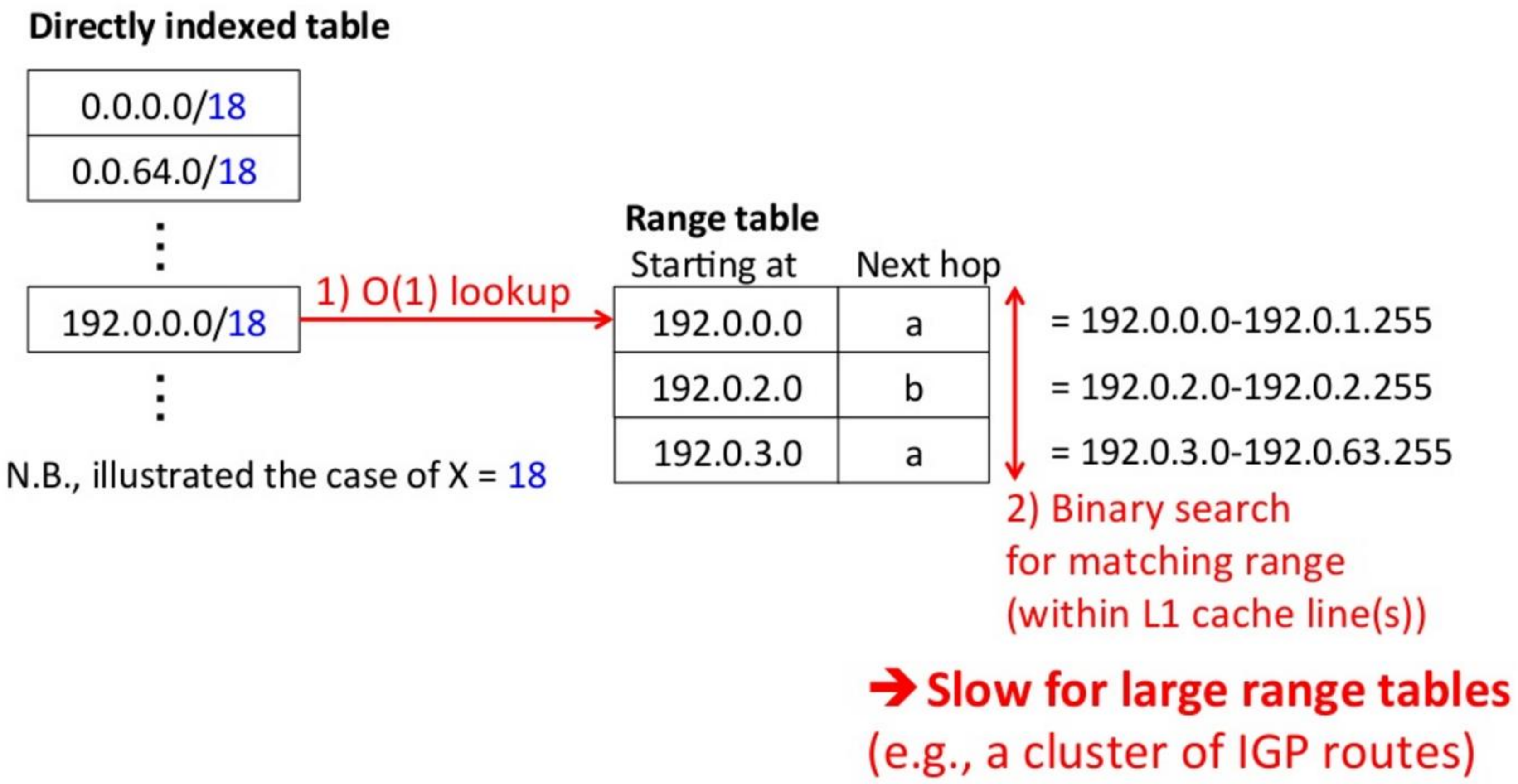
Tree BitMap: Succinct Data Structure



SAIL: 3-level Multiway Trie



DXR: Binary Search in Range Table



Key Ideas for High Lookup Rate

1. Reduce instructions including memory access
   - Reduce the lookup depth of the trie
2. Increase CPU cache efficiency
   - Compress the data structure for small memory footprint within CPU cache size

Poptrie

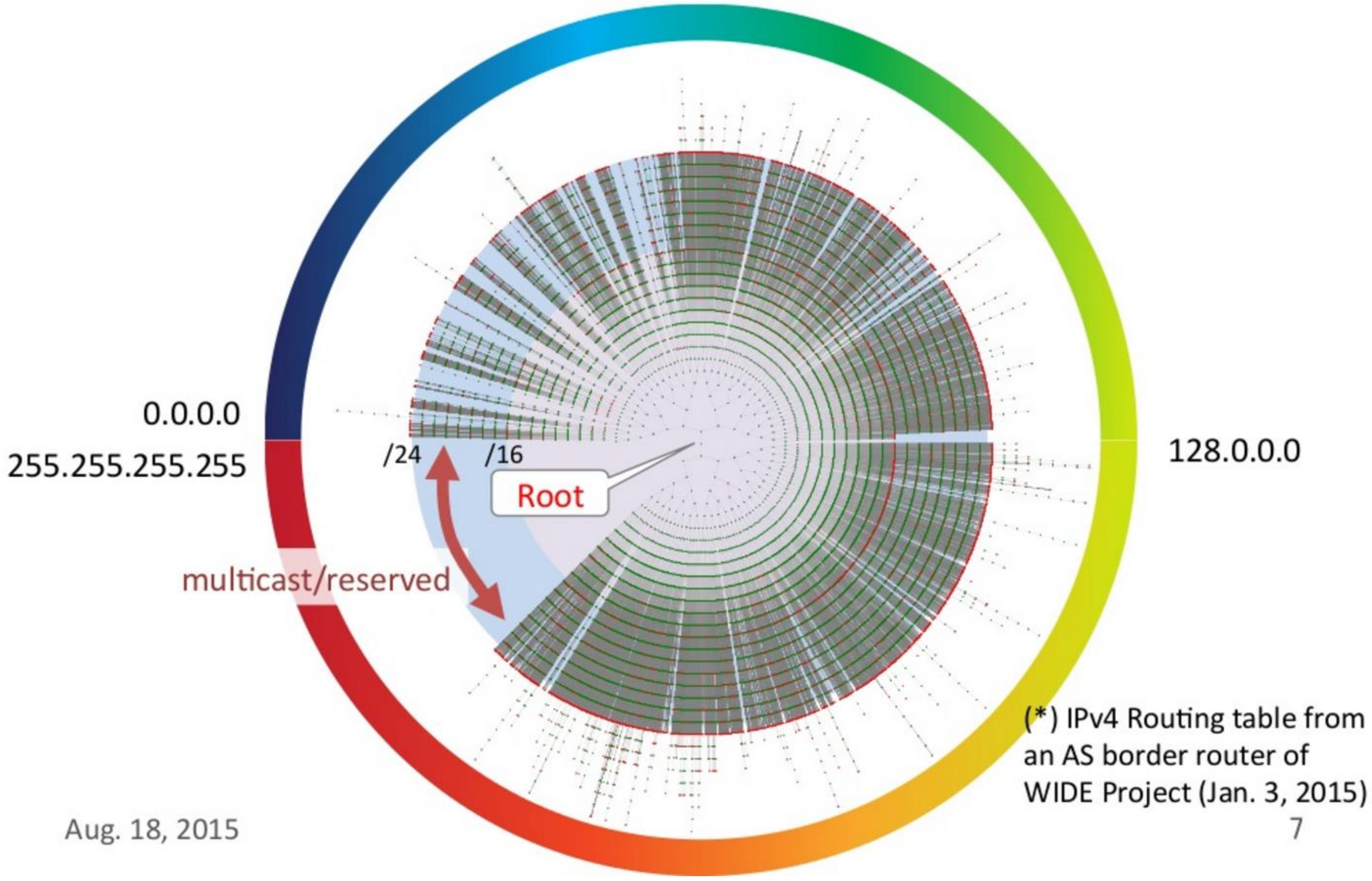
- Extended from the multiway <u>trie</u>
- Compressed with <u>pop</u>ulation count (CPU instruction)

Brief Summary of Poptrie

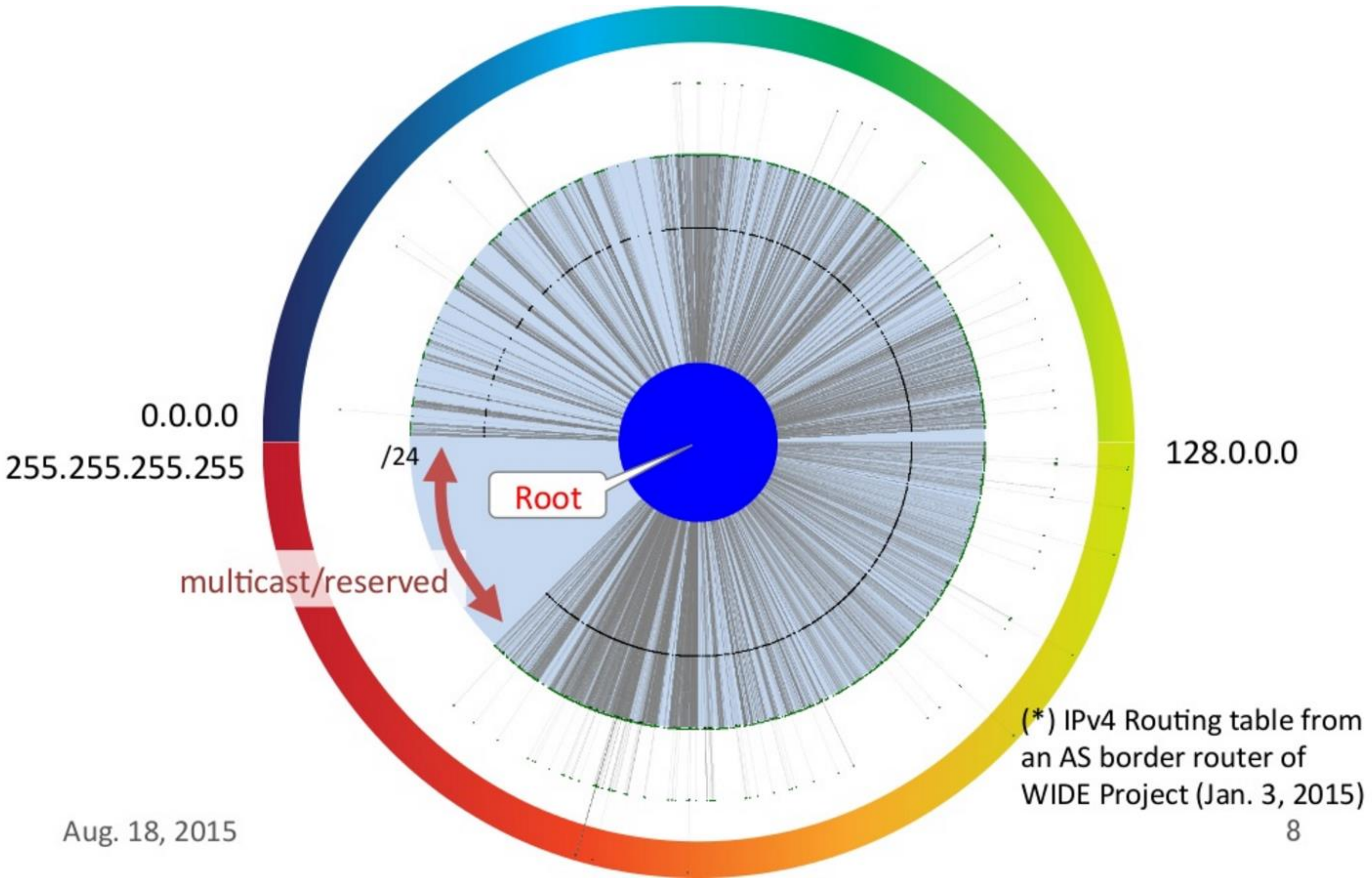
- Multiway trie for IP routing table lookup in software
  - ➢Small memory footprint with population count
    - e.g. 2.4 MiB for a global trier-1 ISP’s full route

  - ➢Good performance through comprehensive evaluation
    - 4-578% faster than other state-of-the-art technologies
      - Private and public IPv4/IPv6 routing tables
      - Future-envisioned 800K+ IPv4 routes
    - Advantageous for longer prefixes
      - demonstrated through per-lookup performance analysis

At a Glance : Radix Tree



At a Glance : Poptrie



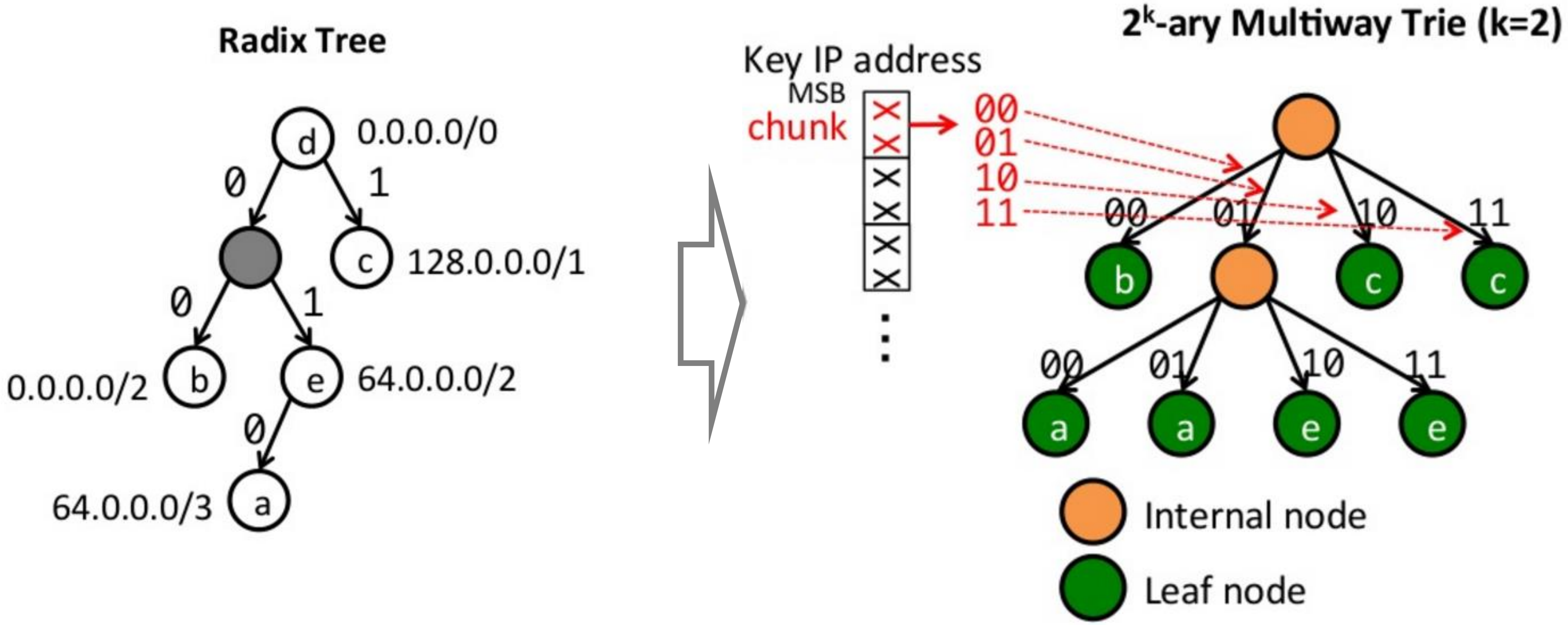
Poptrie

- Basic algorithm
  - Multiway trie
  - Pointer compression with population count
- Extensions
  - Compression with leaf bit-vector
  - Route aggregation
  - Direct pointing

Poptrie

- **Basic algorithm**
  - Multiway trie
  - Pointer compression with population count
- Extensions
  - Compression with leaf bit-vector
  - Route aggregation
  - Direct pointing

Poptrie(basic):2^k-ary Multiway Trie

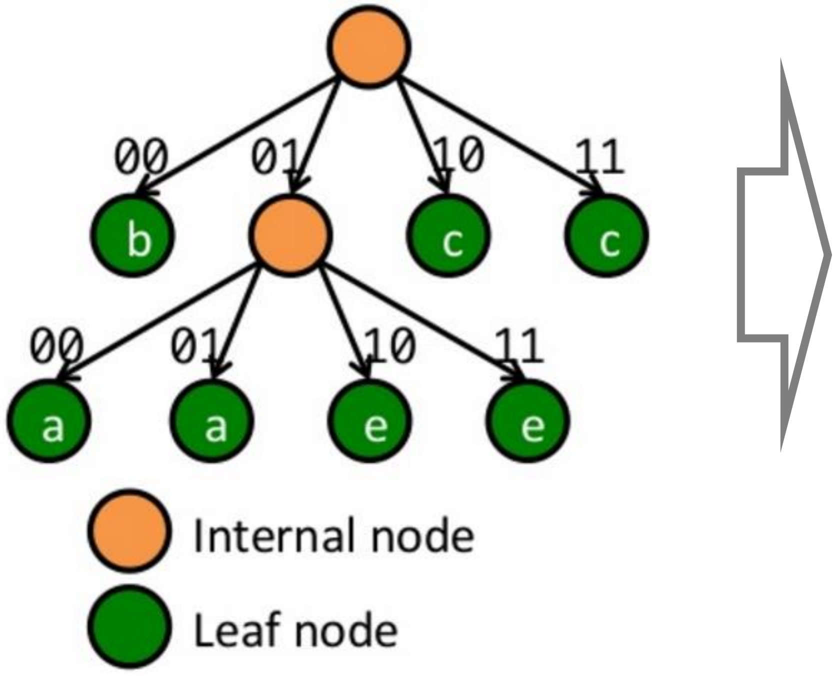


Push the next hop information in the internal nodes to leaf nodes

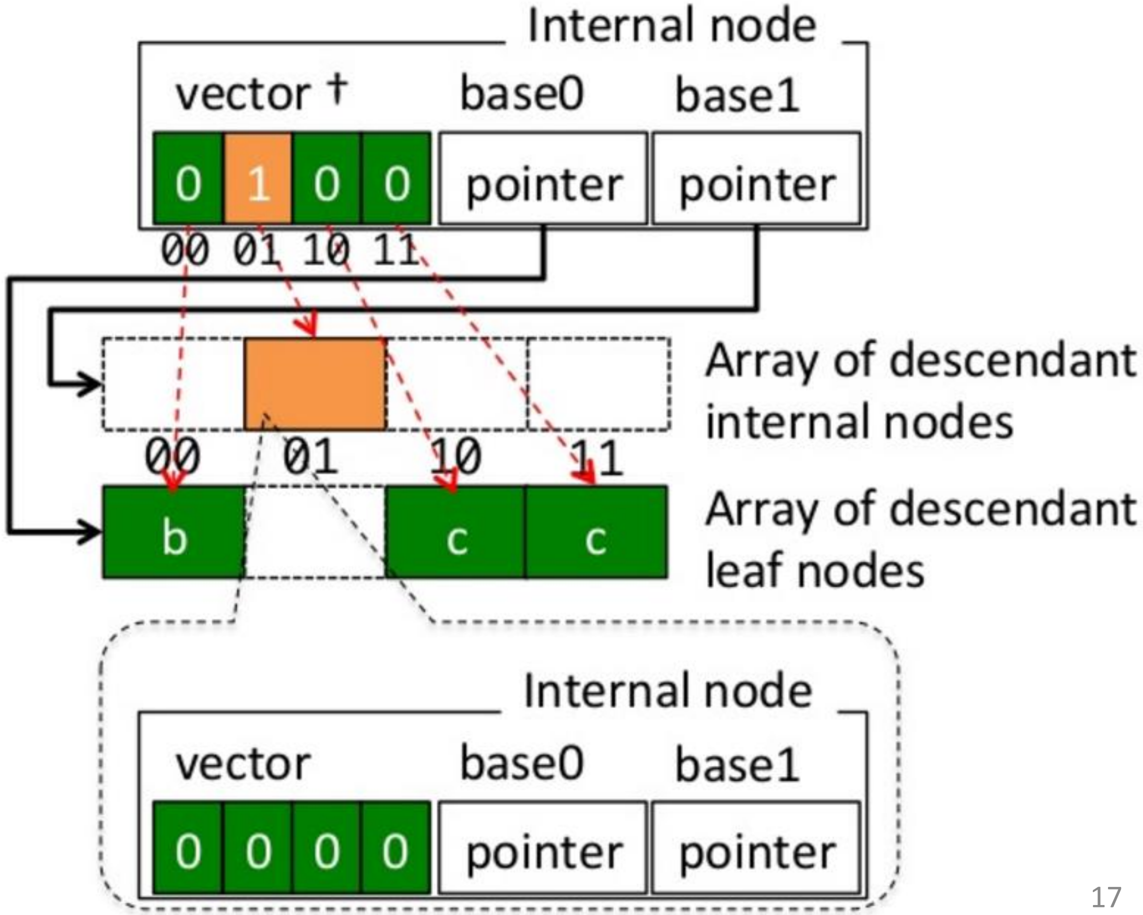
Poptrie(basic):Pointer Compression with Population Count(1/2)

† in little endian

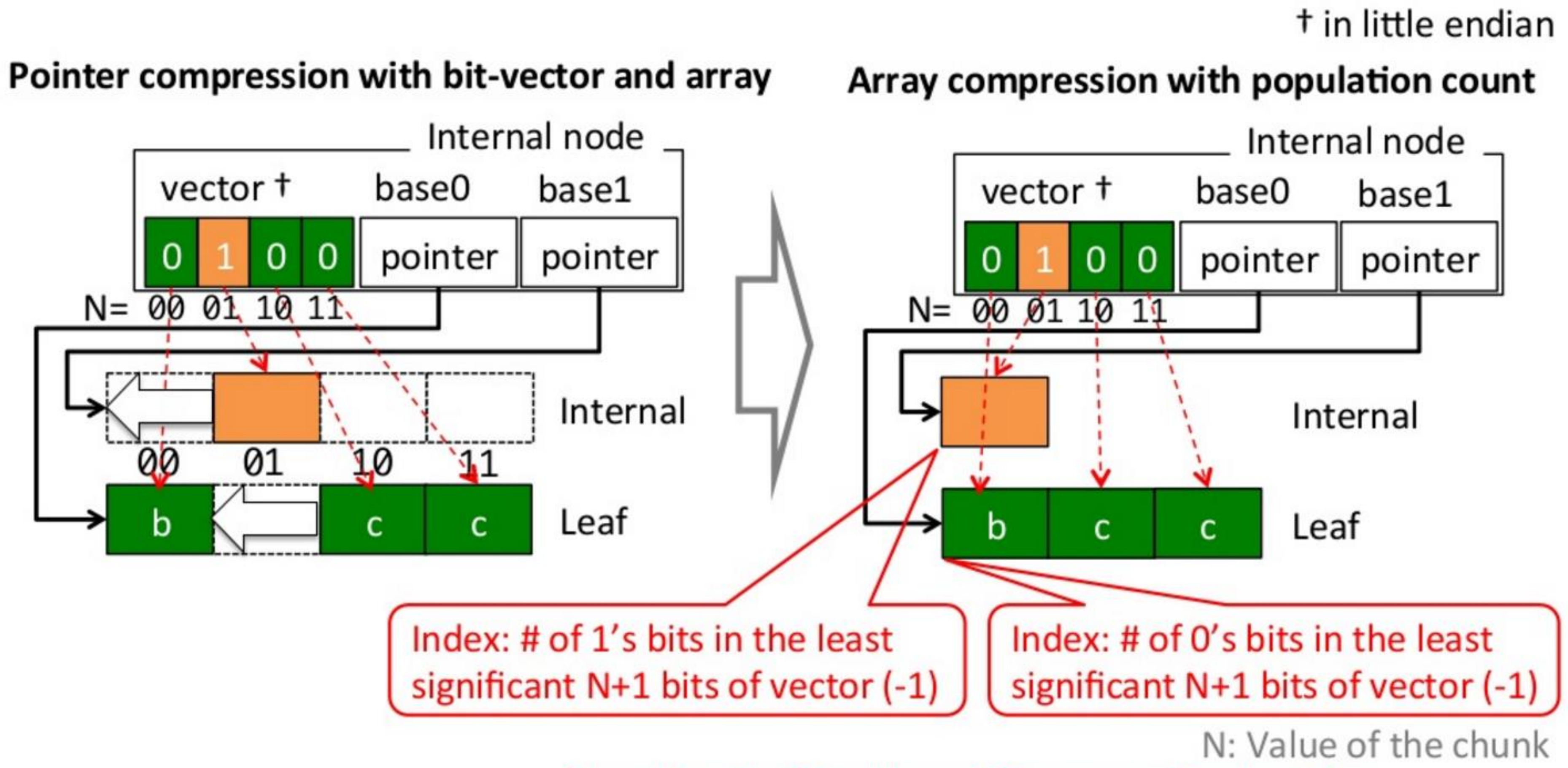
2^k-ary Multiway Trie (k=2)



Pointer compression with bit-vector and array



Poptrie(basic):Pointer Compression with Population Count(2/2)

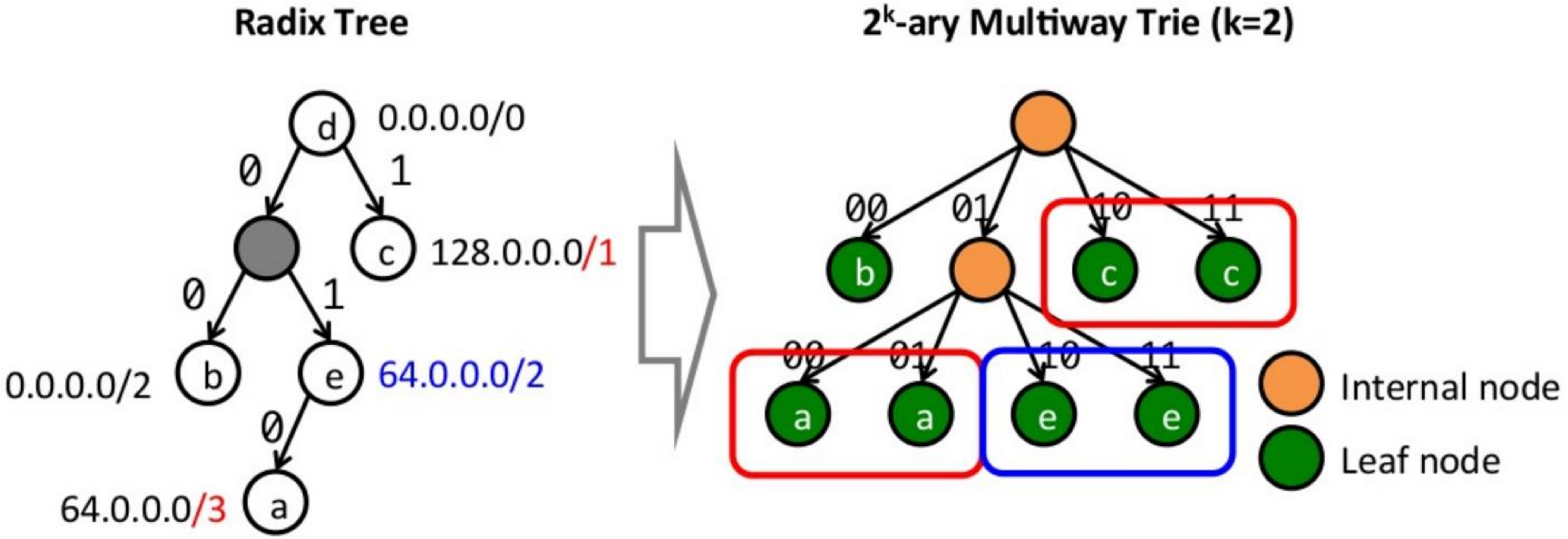


Counting # of 1s: Use x86's *popcnt* instruction

Poptrie

- Basic algorithm
  - Multiway trie
  - Pointer compression with population count
- **Extensions**
  - Compression with leaf bit-vector
  - Route aggregation
  - Direct pointing

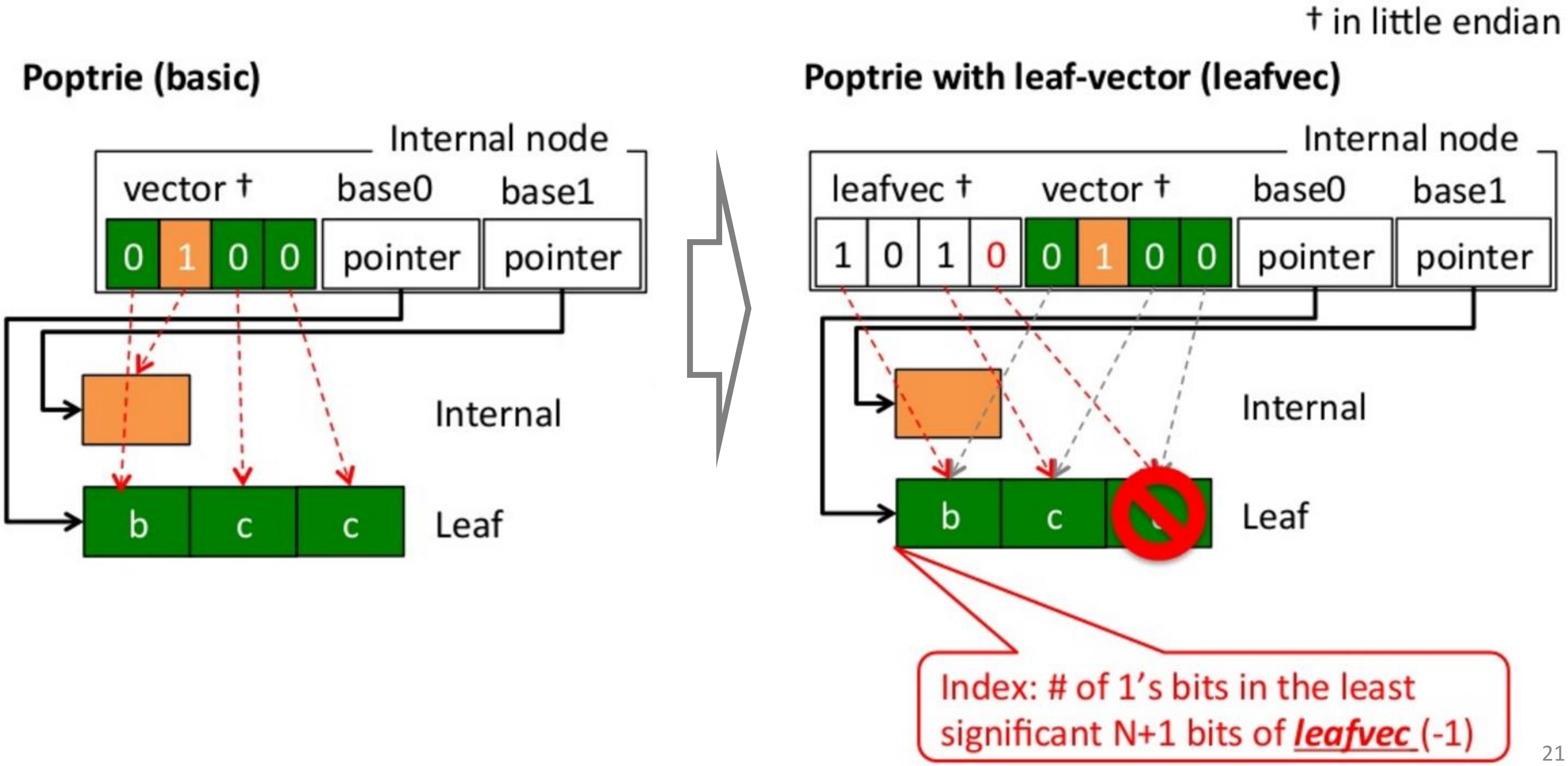
Poptrie: Compression with Leaf-Vector



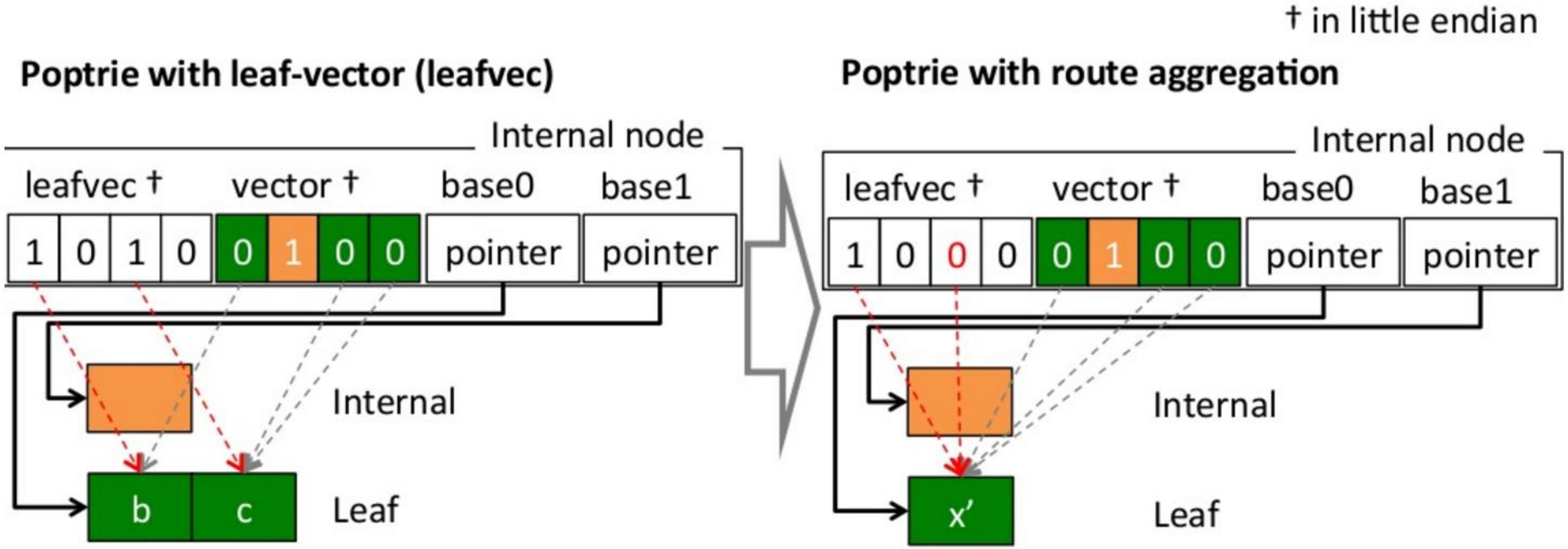
A problem with the basic data structure: Redundant leaf nodes

- for prefixes that do not match k-bit boundary
  - e.g. /1(/7, etc. as well) may create 32 redundant leaf nodes when k=6
- for hole-punched prefixes

Poptrie: Compression with Leaf-Vector



Poptrie: Route Aggregation



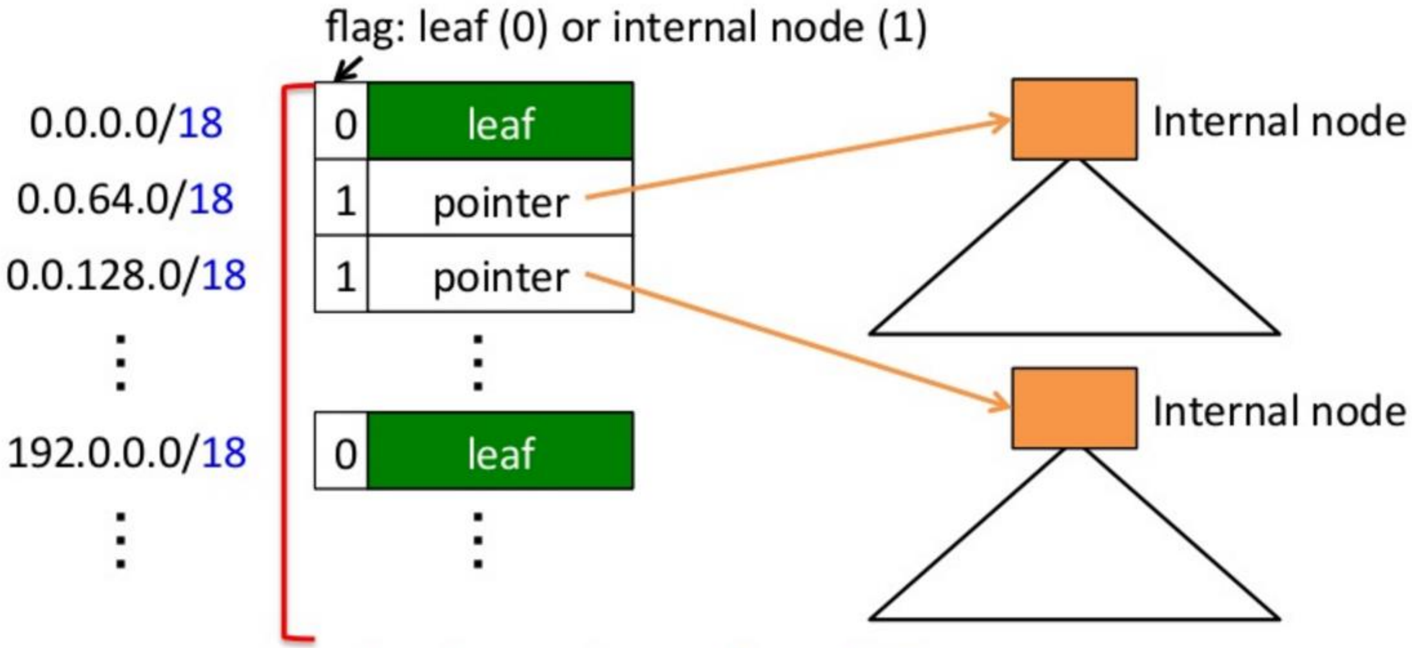
When the next hop info. of b and c is identical;
e.g.,

b: prefix 0.0.0.0/2, next hop X
c: prefix 128.0.0.0/1, next hop X

⇨ Aggregate b and c to x'
(x': prefix aggregated [0.0.0.0/0], next hop X)

Poptrie: Direct Pointing

Extract and lookup ***s*** bits at the first stage (like other algorithms such as DXR)



Top-level array: Array of length 2^s

N.B., illustrated the case of *s* = 18

Poptrie$_x$: Poptrie with $\boldsymbol{s}$ = x

Imlementation

- Data size
  - Internal node (with leafvec): 24 bytes
    - vector, leafvec: 8 bytes(k=6)
    - base0, base1: 4 bytes
      - Index in the contiguous $array^{+}$ instead of memory address
  - Leaf node: 2 bytes
  - Direct pointing entry: 4 bytes

- Code in C: https://github.com/pixos/poptrie

+ The contiguous arrays of the internal and leaf nodes are managed by the buddy memory allocator to allocate an array of descendant nodes.

Evaluations

- Effect of Extensions in Poptrie
- Multicore Scalability
- Comparison with Other Algorithms
  - Average lookup rate
    - Traffic pattern: Sequential, repeated, random, real trace
    - Routing tables: 3 private, 32 public, 4 synthetic
  - CPU cycles per lookup
- Update
- Performance in IPv6

Equipment for the evaluations:
Intel(R) Core i7 4770K (3.9 GHz, 8 MiB cache)
with four 8 GB DDR3-1866 RAM
(using a single core)

Effect of Extensions in Poptrie

Algorithm / Extensions	s	# of internal nodes	# of leaves	Memory [MiB]	Rate [Mlps]
Radix	–	–	–	30.48	8.82
Poptrie (basic) without route aggregation	0	64,009	4,032,568	8.67	87.71
	16	172,110	10,862,901	23.60	130.72
	18	61,282	3,911,422	9.40	170.69
Poptrie (leafvec) without route aggregation	0	64,009	280,673	2.00	89.15
	16	172,110	347,449	4.85	154.33
	18	61,282	269,320	2.91	191.95
Poptrie	0	43,191	263,381	1.49	96.27
	16	86,171	274,145	2.75	198.28
	18	40,760	245,034	2.40	**240.52**

Mlps
= Million lookups
per second

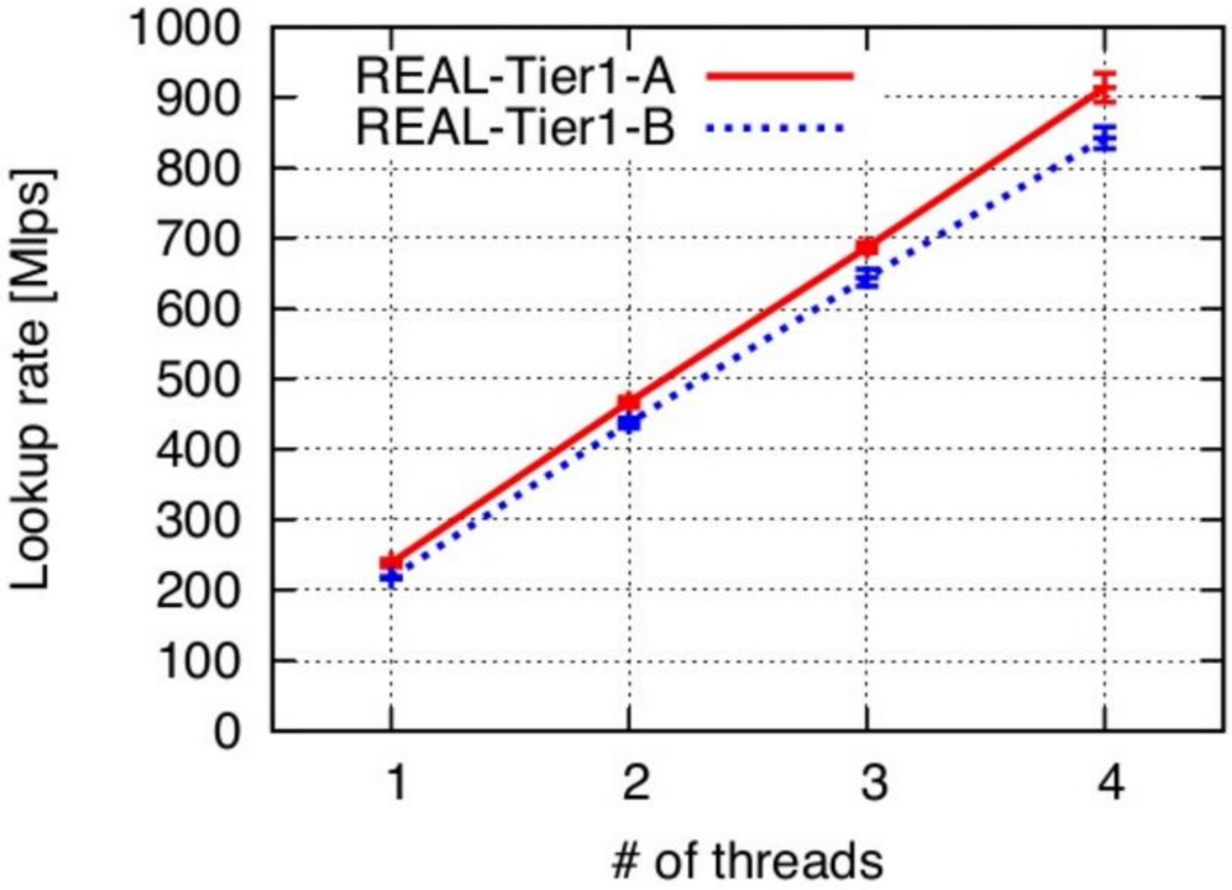
Routing table: Backbone core router of tier-1 ISP (Jan. 9, 2015), Traffic pattern: Random

1. *leafvec* significantly contributes the memory footprint reduction.
2. Route aggregation contributes to reduce the internal nodes (i.e., average depth).

Multicore scalability Evaluation for Random Traffic

REAL-Tier1-A: Global Tier-1's BGP Router

REAL-Tier1-B: Domestic ISP's BGP Router



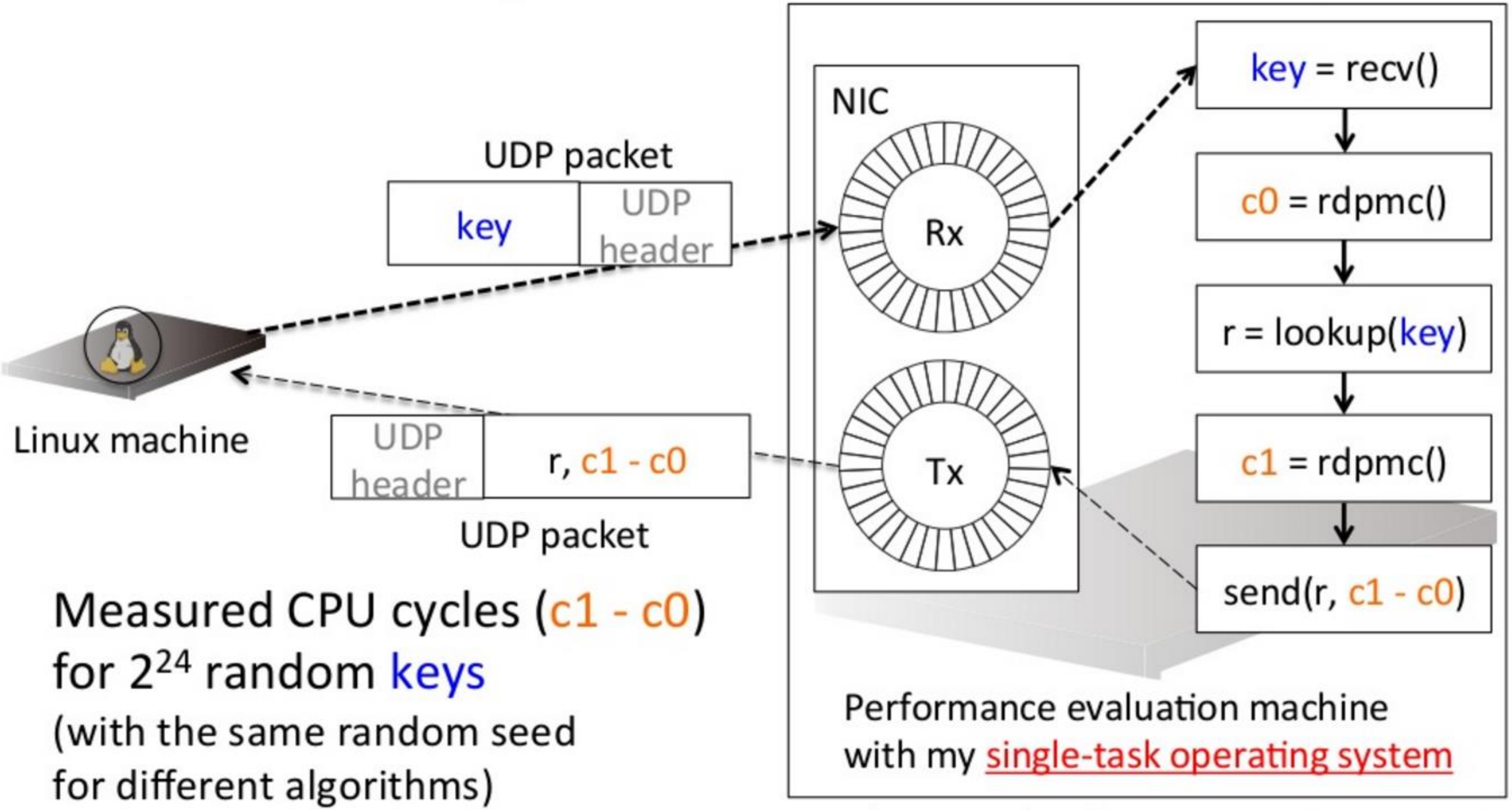
Comparison with Other Algorithms (for random traffic pattern)

Algorithm	Memory [MiB]	Rate [Mlps]
Radix	30.48	8.82
Tree BitMap	2.62	56.24
Tree BitMap (64-ary)	3.10	61.61
SAIL	44.24	158.22
D16R	1.16	116.63
D18R	1.91	179.92
Poptrie_{16}	2.75	198.28
Poptrie_{18}	2.40	240.52

Routing table: Backbone core router of tier-1 ISP (Jan. 9, 2015), Traffic pattern: Random

Poptrie_{18} runs <u>1.34–27.3x</u> faster than the other algorithms

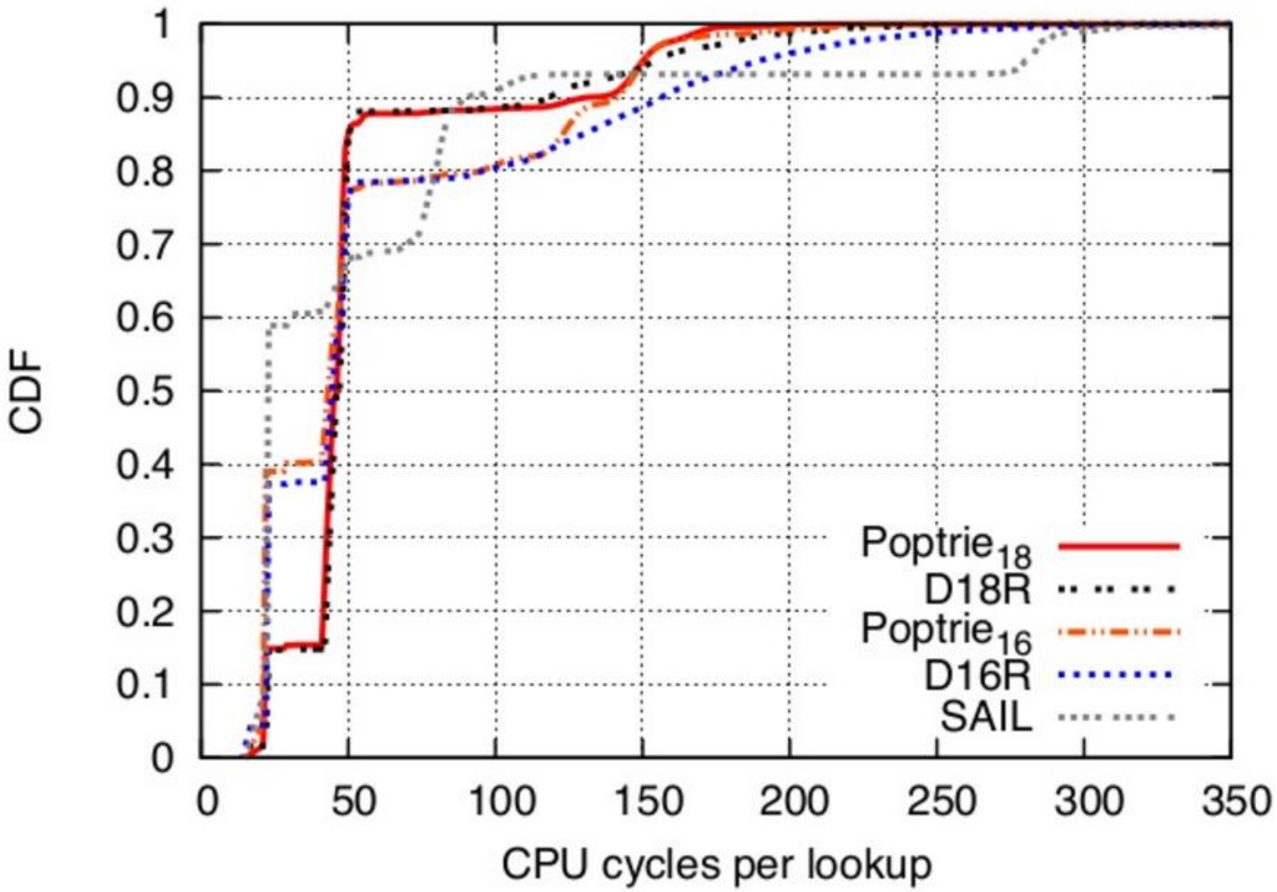
Per-Lookup Performance Analysis



Measured CPU cycles (c1 - c0)
for 2^{24} random keys
(with the same random seed
for different algorithms)

rdpmc : read performance monitoring counter
(i.e., CPU cycles)

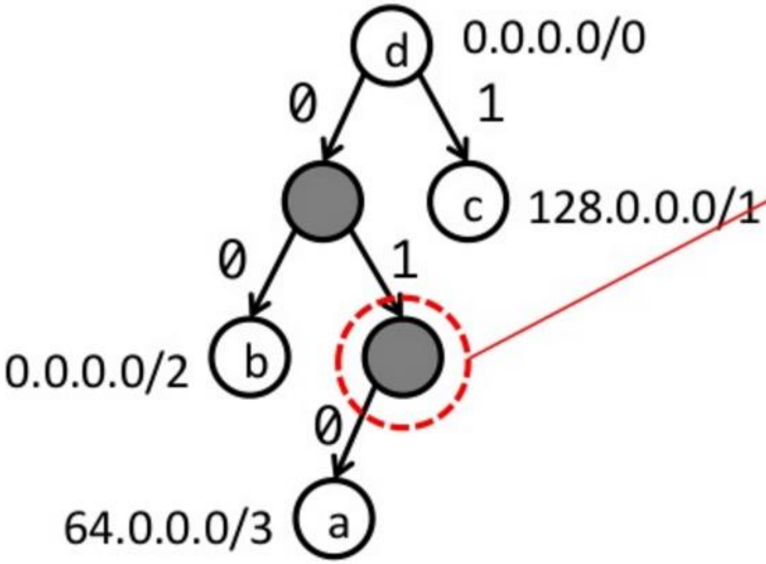
Per-Lookup Performance Analysis: Cumulative Distribution of CPU Cycles



It is worth to analyze per-lookup performance than the average!

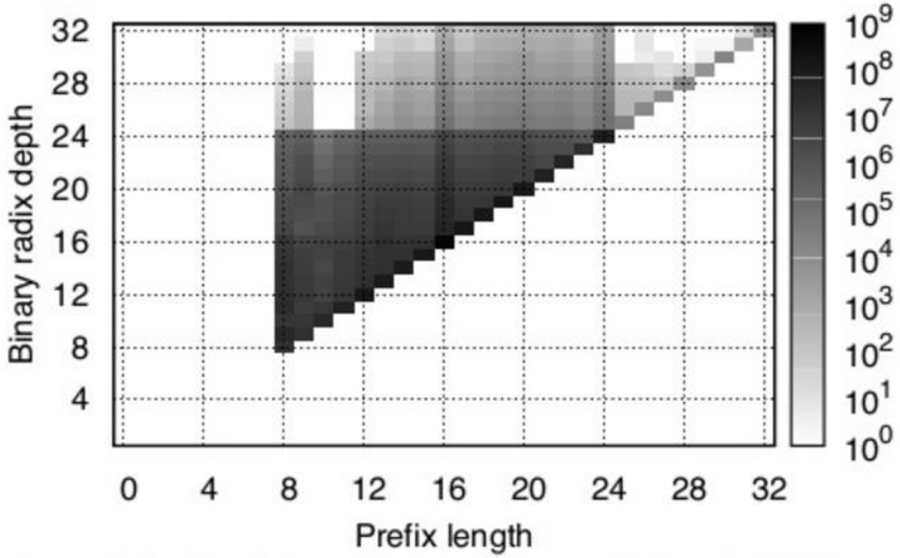
Per-Lookup Performance Analysis
Binary Radix Depth

Radix Tree



Binary radix depth = Search depth in the radix tree

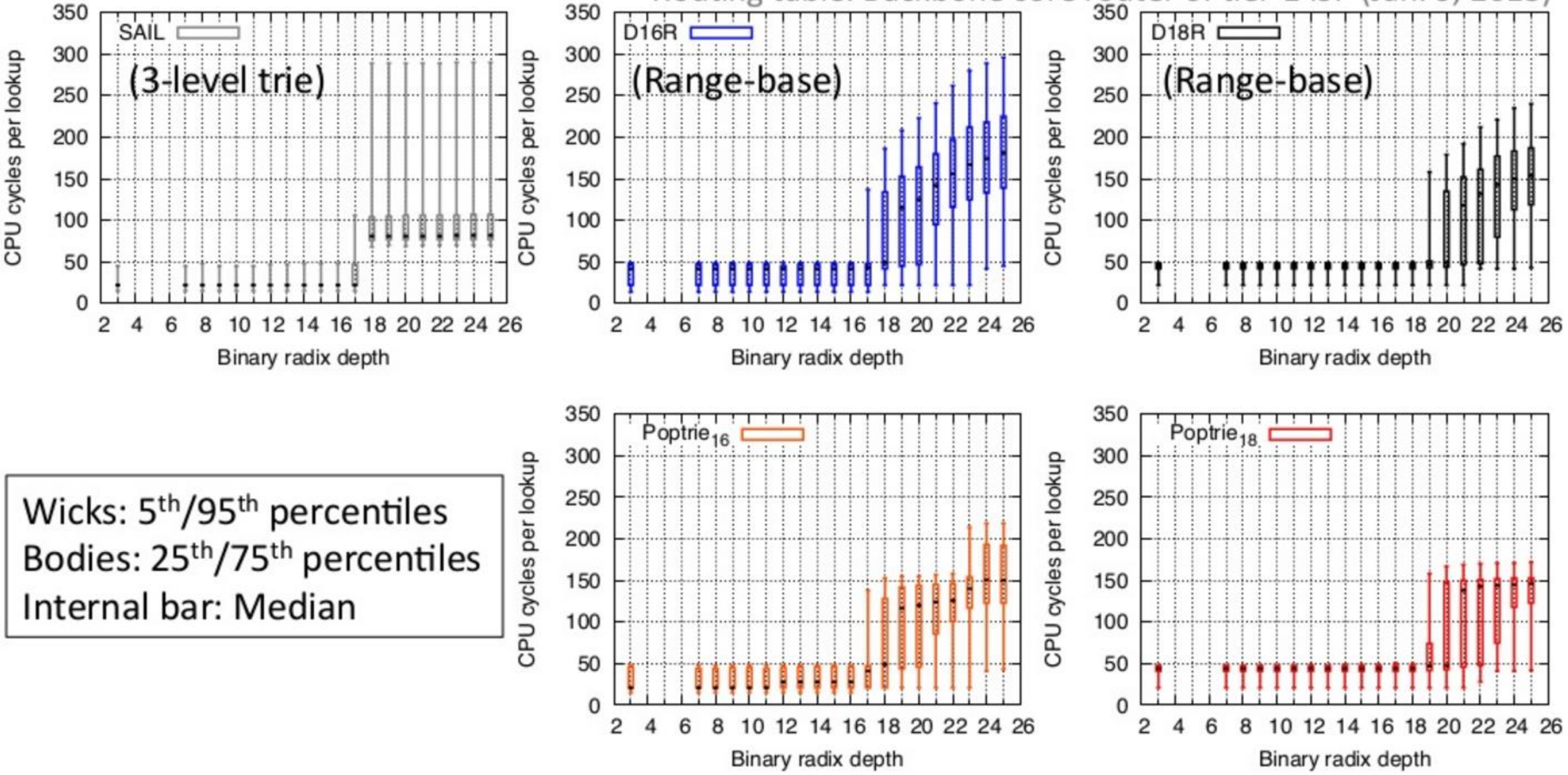
- 0.0.0.0/2 : 2
- 64.0.0.0/3 : 3
- **96.0.0.0/3 : 2** (matching 0.0.0.0/0)
- 128.0.0.0/1 : 1



Routing table: Backbone core router of tier-1 ISP (Jan. 9, 2015)

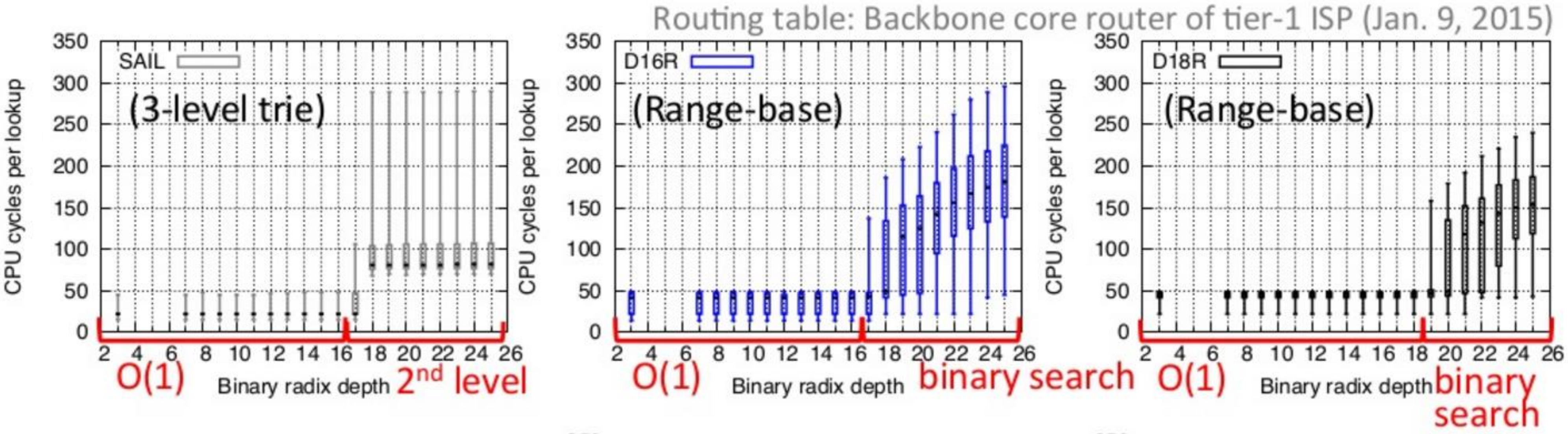
Per-Lookup Performance at Different Binary Radix Depth

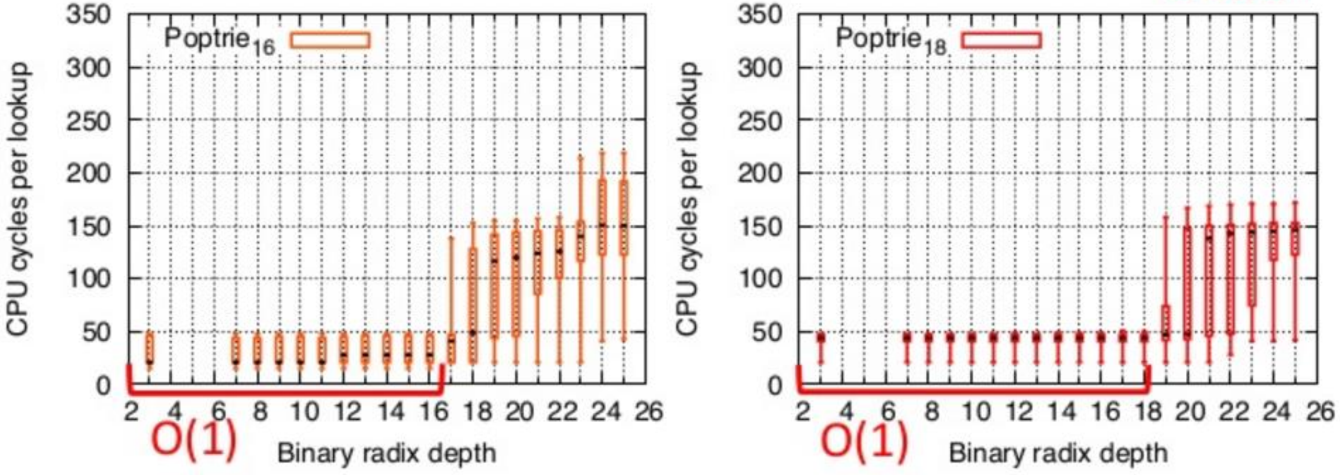
Routing table: Backbone core router of tier-1 ISP (Jan. 9, 2015)



Wicks: 5th/95th percentiles
Bodies: 25th/75th percentiles
Internal bar: Median

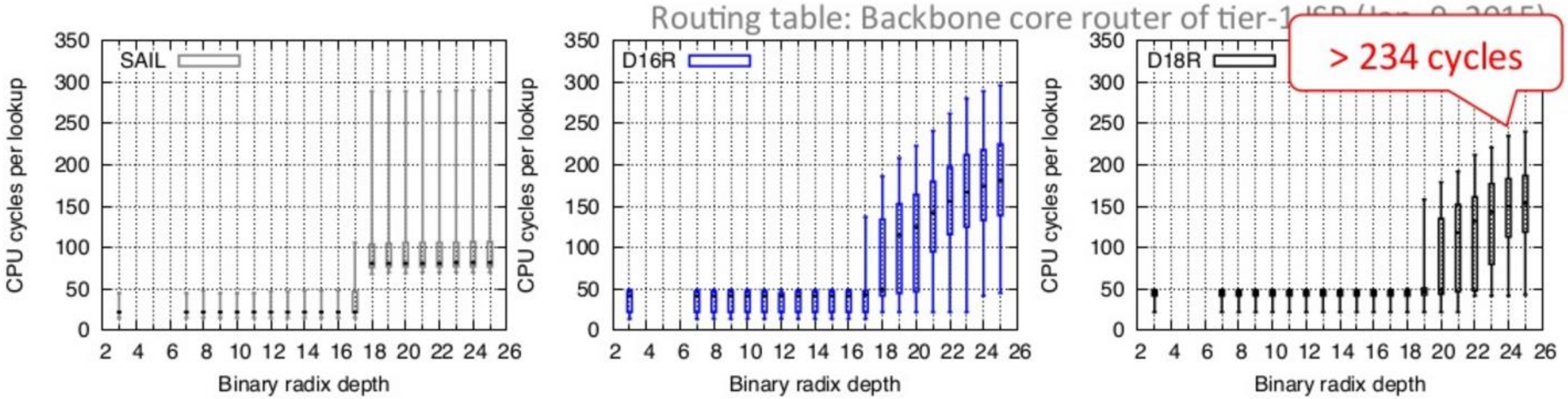
Per-Lookup Performance at Different Binary Radix Depth



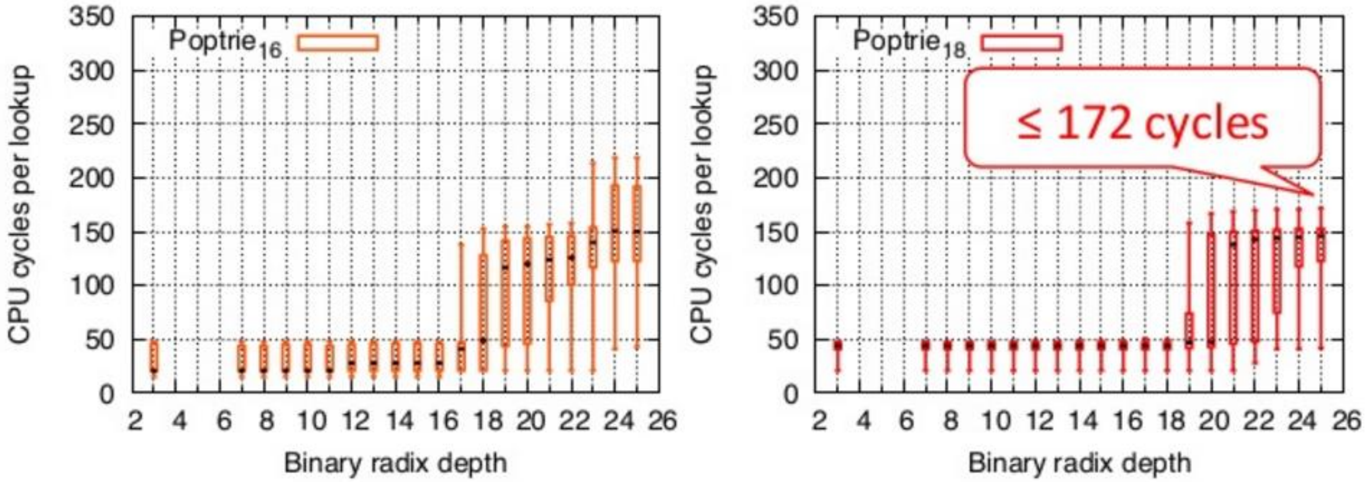


Wicks: 5th/95th percentiles
Bodies: 25th/75th percentiles
Internal bar: Median

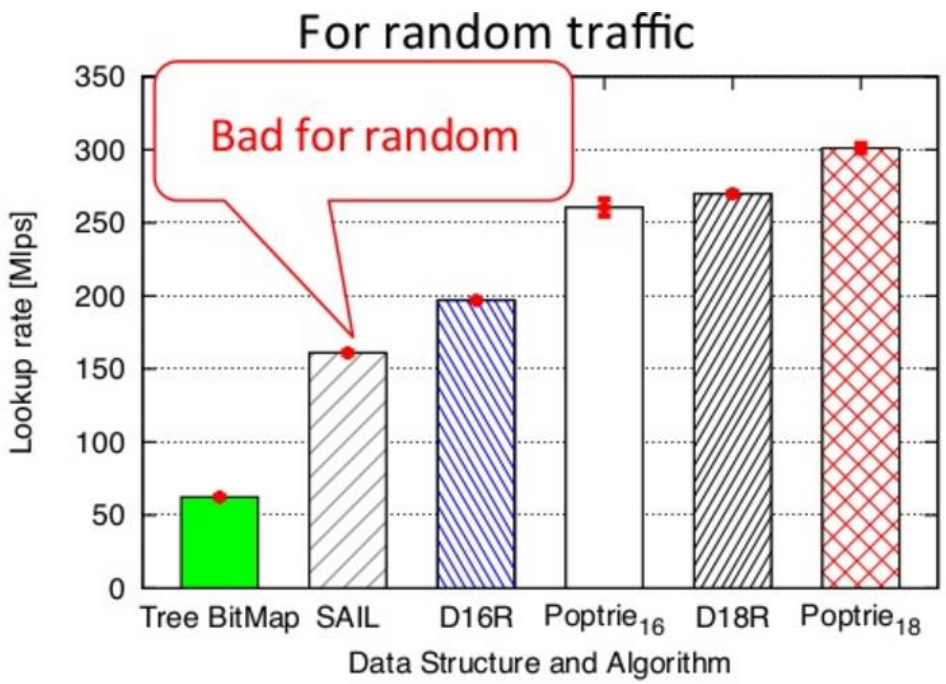
Per-Lookup Performance at Different Binary Radix Depth

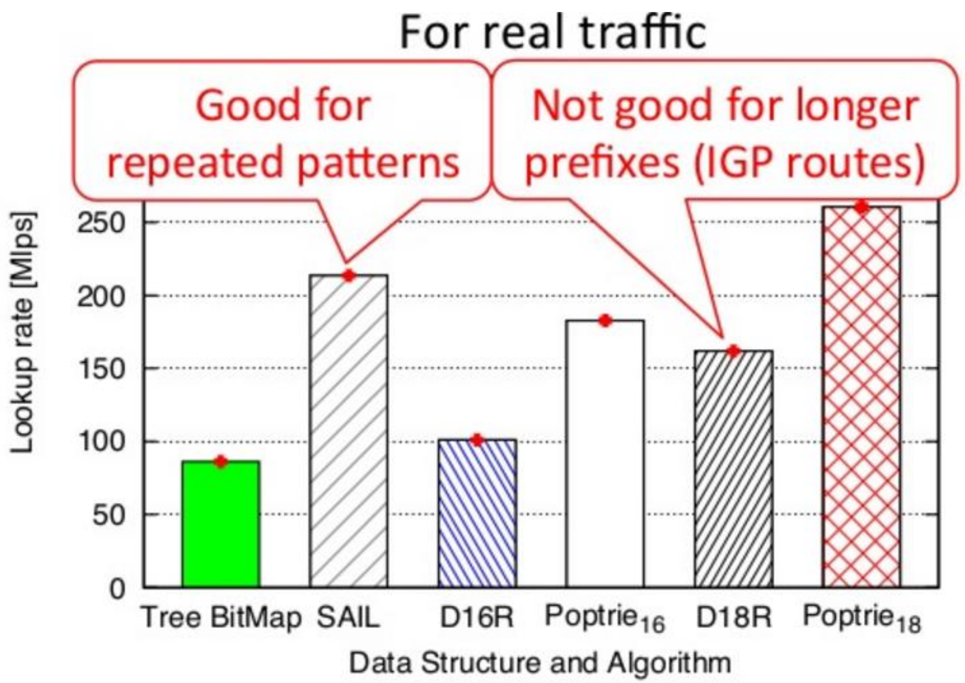


Wicks: 5th/95th percentiles
Bodies: 25th/75th percentiles
Internal bar: Median



Average Lookup Rate: Random vs. Real Traffic





Binary radix depth	Random	Real
>18	22.1%	32.5%
>24	1.7%	$\boxed{21.8\%}$

Many packets go to IGP routes.

(*) Routing table is dumped at an AS border router of WIDE project, and traffic trace is captured at the transit link of the router.

Performance on Large Routing Tables

Two synthetic datasets from the routing table of a backbone core router of tier-1 ISP

SYN1:
- /0-15 : split into four prefixes
- /16-23 : split into two prefixes

SYN2:
- /0-15 : split into eight prefixes
- /16-19 : split into four prefixes
- /20-23 : split into two prefixes

Algorithm	SYN1 (with 764,847 routes)	SYN2 (with 885,645 routes)
SAIL	102.86 Mlps	N/A (Overflowed)
D18R (modified)	115.45 Mlps	102.59 Mlps
Poptrie_{18}	188.02 Mlps	174.42 Mlps

Traffic pattern: Random

Poptrie_{18} is <u>1.63–1.70x</u> faster than D18R.

Performance on IPv6 Routing Table

Algorithm	Average Lookup rate [Mlps]
SAIL	N/A (no support for more specific routes than /64)
D16R	163.07
D18R	169.91
Poptrie_{16}	209.84
Poptrie_{18}	211.32

- On a routing table dumped at a tier-1 ISP (with 20,440 prefixes)
- For 2^32 random addresses within 2000::/8

Poptrie_{18} is 1.24x faster than D18R.

Conclusion

- Poptrie: Multiway trie for IP routing table lookup in software
  - Small memory footprint with population count
    - e.g. 2.4 MiB for a global tier-11 ISP's full route
  - Good performance through comprehensive evaluation
    - 4-578% faster than other state-of-the art technologies
      - Private and public IPv4/IPv6 routing tables
      - Future-envisioned 800K+ IPv4 routes
    - Advantageous for longer prefixes
      - demonstrated through per-lookup performance analysis

Thank you

&

Questions