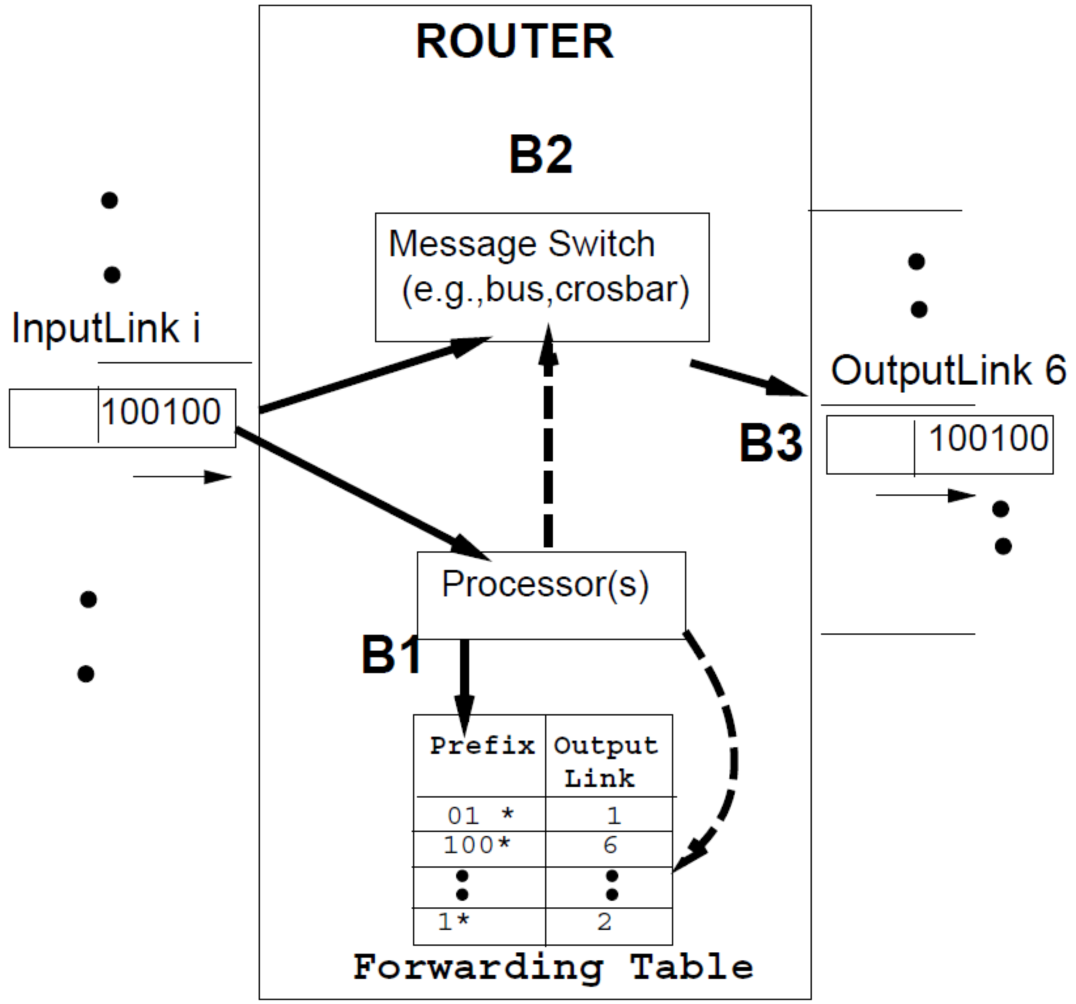
Routers

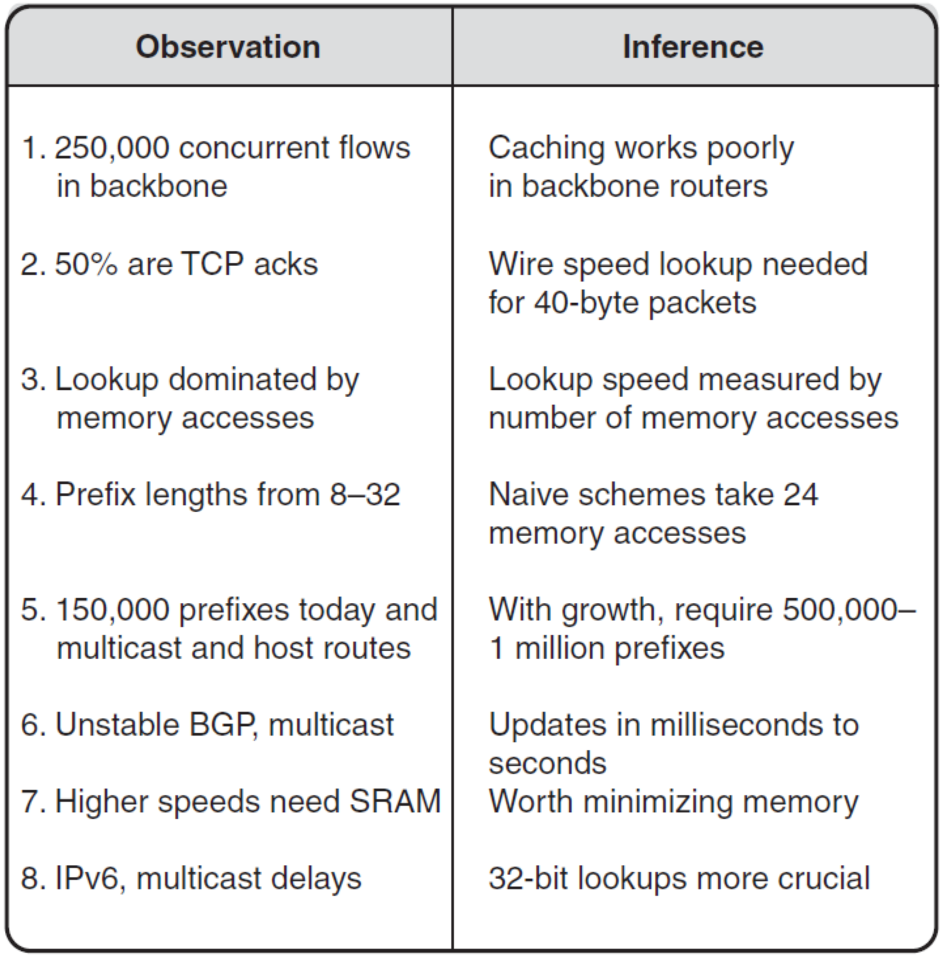
Prefix-Match Lookup

Prefix-Match Lookup



Prefix	Output Link
01 *	1
100*	6
⋮	⋮
1*	2

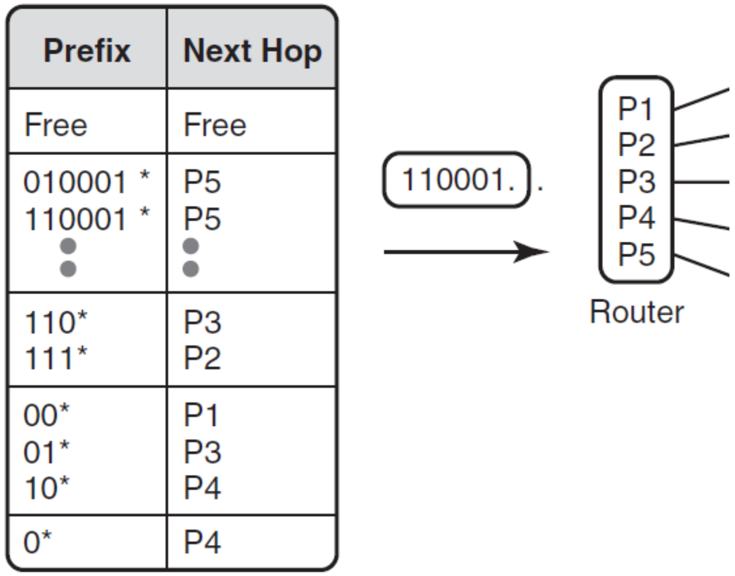
Context of the problem



Observation	Inference
1. 250,000 concurrent flows in backbone	Caching works poorly in backbone routers
2. 50% are TCP acks	Wire speed lookup needed for 40-byte packets
3. Lookup dominated by memory accesses	Lookup speed measured by number of memory accesses
4. Prefix lengths from 8–32	Naive schemes take 24 memory accesses
5. 150,000 prefixes today and multicast and host routes	With growth, require 500,000–1 million prefixes
6. Unstable BGP, multicast	Updates in milliseconds to seconds
7. Higher speeds need SRAM	Worth minimizing memory
8. IPv6, multicast delays	32-bit lookups more crucial

Ternary CAMs

- A string is ternary if contains either 0, 1 or * where * is a wildcard
- To simplify notation we will use * as a wildcard for any number of characters
- A Ternary Content-Addressable Memory (TCAM) is a memory containing ternary strings of given length.
- When presented with input, the TCAM searches all locations in parallel and returns the lowest location whose string that matches the key



Prefix	Next Hop
Free	Free
010001 *	P5
110001 *	P5
●	●
●	●
110*	P3
111*	P2
00*	P1
01*	P3
10*	P4
0*	P4

TCAMs

TCAMs are extremely fast, but

- Are large (10-12 transistors per bit)
- Consume power (because of parallel compare)
- Are expensive (corollary of the first two points)

Still unclear what is best for a router with millions of prefixes

Update in TCAMs

TCAMs require that longer prefixes are before shorter ones.

Problem: how to add an entry?

Naive solution: suppose the new prefix has length i

- Move up all the longer prefixes
- Create an hole at the beginning of the length i prefixes
- Insert the new prefix

This is too slow.

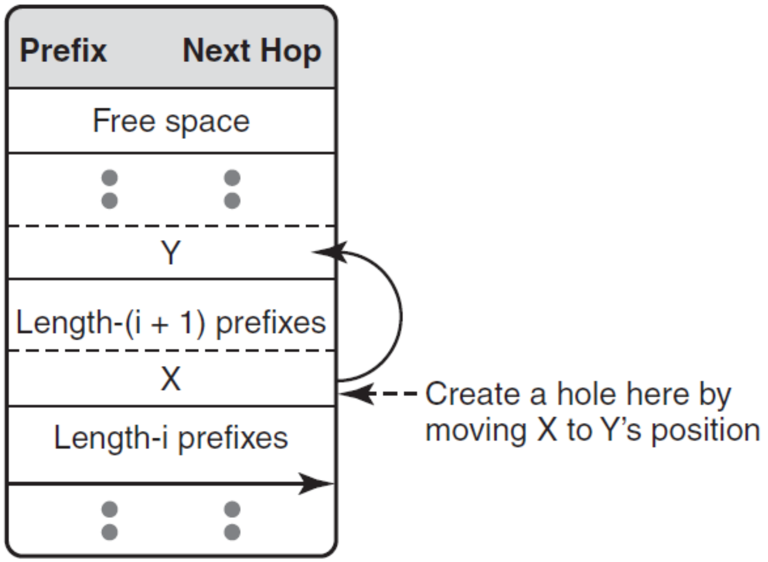
Update in TCAMs

Solution

- No need to sort prefixes with the same length
- Move away the lowest i+1 prefix (apply this algorithm recursively)
- Create an hole at the beginning of the length i prefixes
- Write the new prefix

This solution requires at most 32-i memory accesses to update the table

Placing free space in the middle halves the accesses



Algorithmic solutions

Two large families

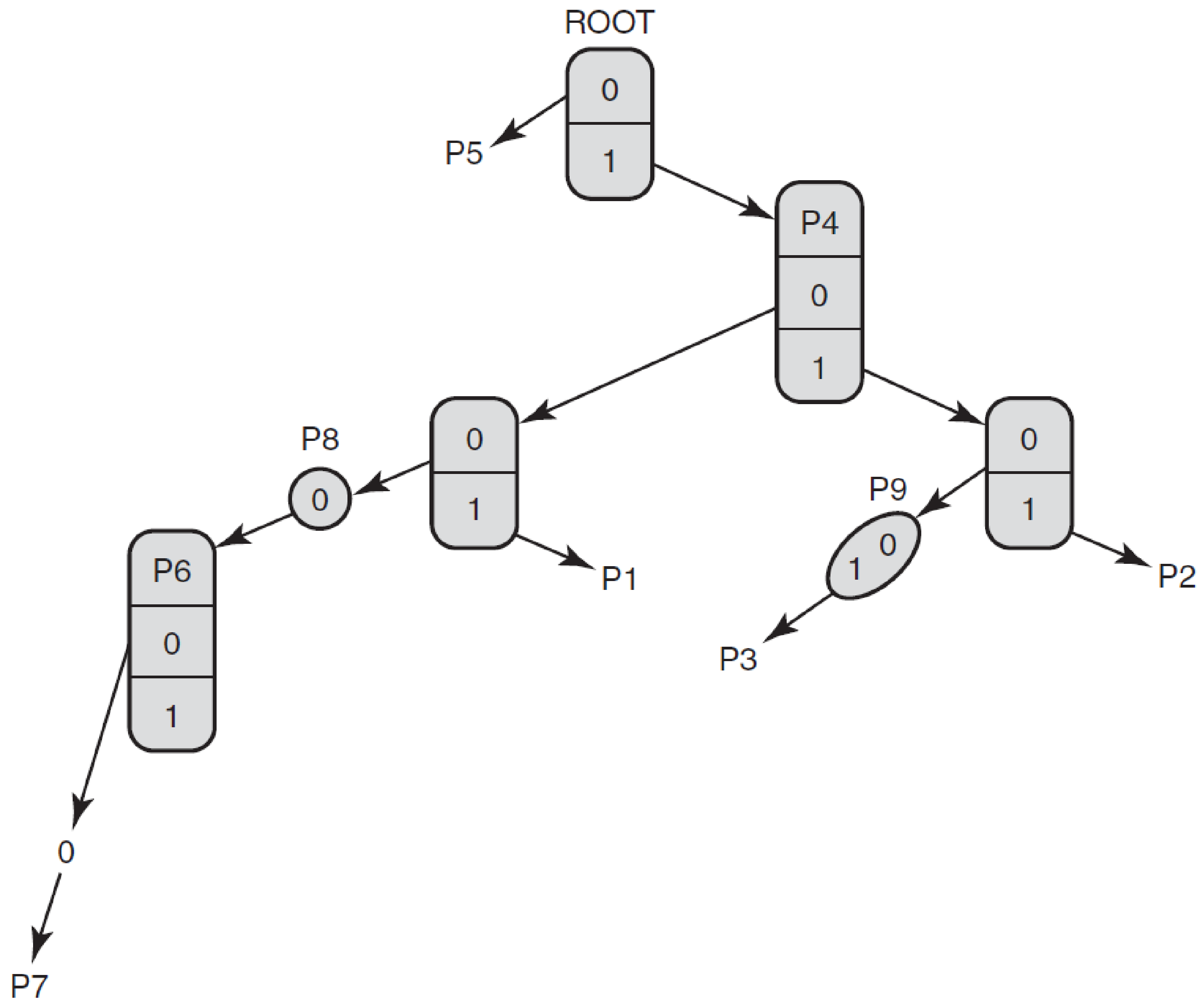
- Tries (or prefix tree, is a tree used to store an associative array)
  - Unibit Tries
  - Multibit Tries
  - Level-Compressed Tries
  - Tree Bitmaps
- Binary Search
  - on ranges
  - on prefix lengths

Performance metrics

- storage complexity
- time complexity (lookup and update)

Unibit Tries

P1=101*
P2=111*
P3=11001*
P4=1*
P5=0*
P6=1000*
P7=100000*
P8=100*
P9=110



Unibit Tries

The unibit trie is a tree in which each node

- stores associated data (next hop) and
- an array with a 0-pointer and a 1-pointer

A pointer either

- points to the associated data (next hop) or
- points to a subtrie

In the root, all the prefixes

- starting with 0 are in the subtrie pointed by the 0-pointer
- starting with 1 are in the subtrie pointed by the 1-pointer

Each subtrie is constructed recursively

Unibit tries

Two special cases

A prefix may be a subprefix of another prefix

- in the example P4 = 1* and P2 = 111*
- the shorter prefix is stored inside a node along the path

Some subtrees have one-way branches

- in the example P3 = 11001*: there are no branches for the last two bits
- wasted space and time

Unibit tries

How to search for the longest match for address D.

Start with the root and continue until either you find

- an empty pointer
- a string that does not match

The algorithm keeps track of the last prefix along the path. When search fails, this is the longest match.

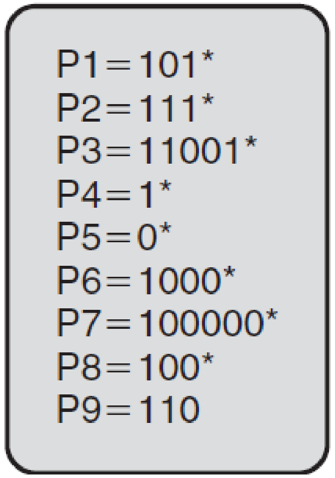
Multibit Tries

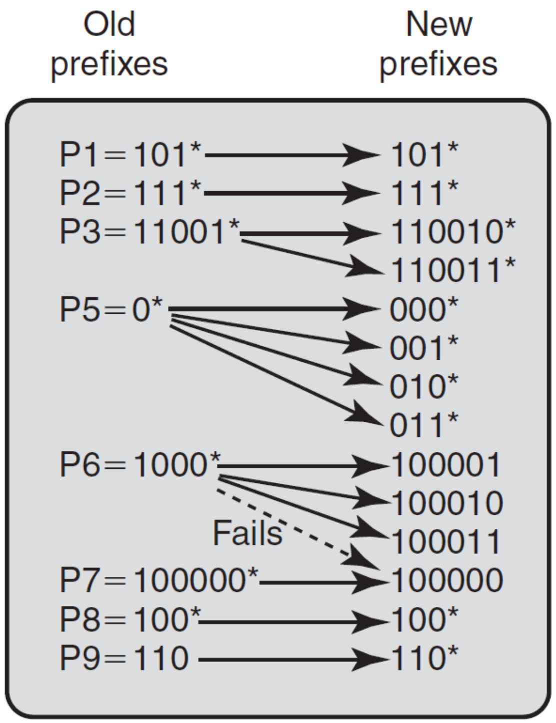
Unibit Tries have a drawback: in worst case 32 memory accesses (slow!)

Multibit Tries

- reduce the memory accesses by searching in strides of n bits (n>1)
- n can be fixed for all the tree or can change at each node
- if the prefix is not a multiple of the stride the prefix must be expanded

Example:
Prefix expansion to multiples of stride n = 3


P1=101*
P2=111*
P3=11001*
P4=1*
P5=0*
P6=1000*
P7=100000*
P8=100*
P9=110



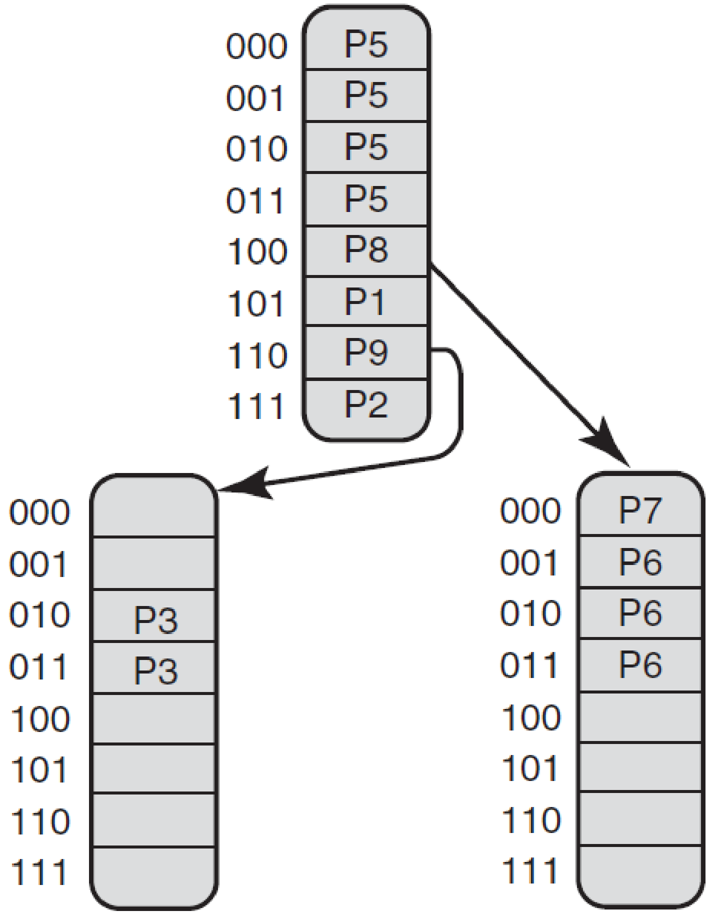
Fixed-Stride Multibit Tries

At each nodes, part of the address is an index in an array

Each element contains

- a prefix (P1, P2, …)
- a pointer

When an element is empty, the last prefix encountered is returned



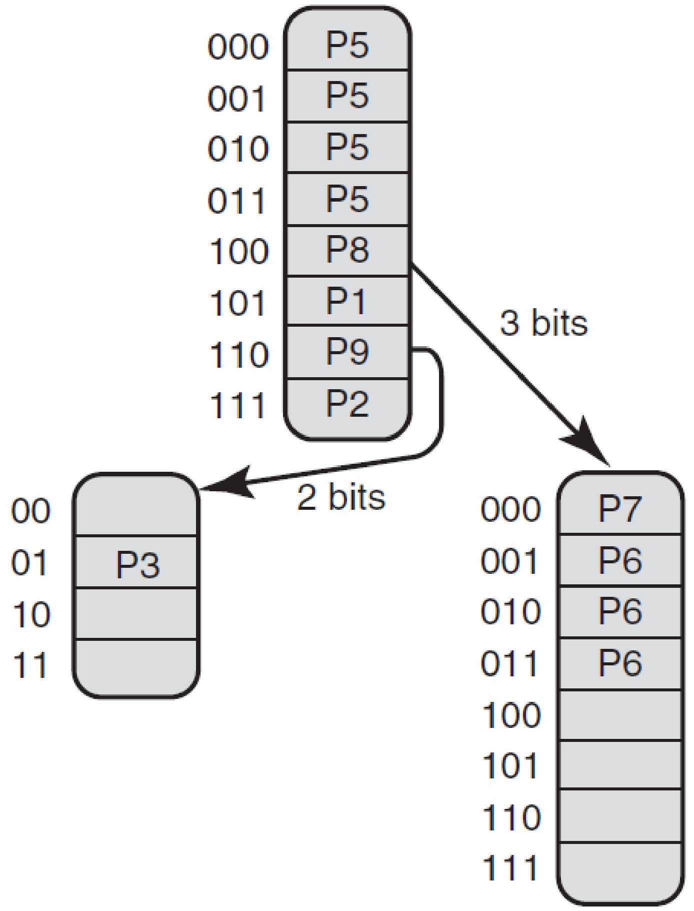
Variable-Stride Multibit Tries

With fixed stride there is a waste of space (in expanding we traded memory for time)

Variable stride reduces the storage given the maximum height of the trie (fast SRAM is expensive)

Problems:

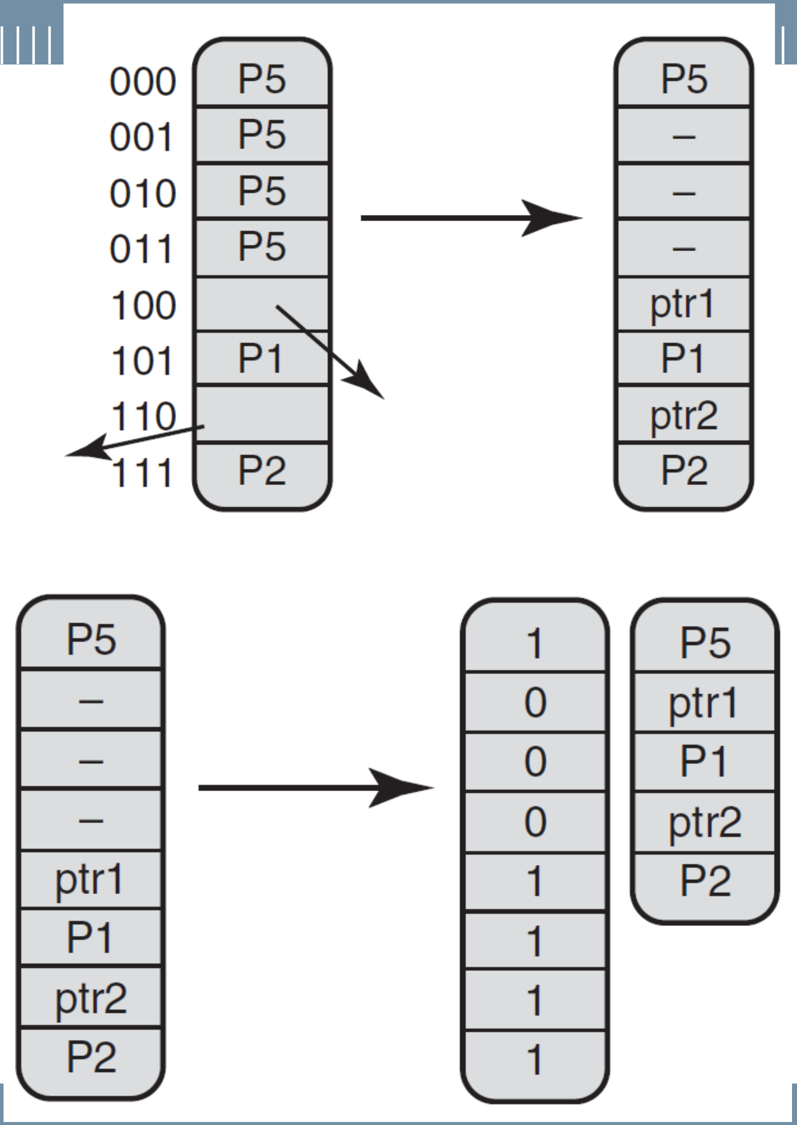
- Find optimal strides O(N W^2 h)
  - N = no. prefixes
  - W = width of dest. addresses
  - h = max height
- Incremental update O(W) + O(S)
  - S = max size of a trie node



Lulea-Compressed Tries (1)

A multi-bit trie in which repeated information in nodes is compressed (to minimize storage)

1) Each node entry can contain either one prefix or one pointer. Prefixes are pushed towards the leaves
2) A bitmap of the node contains 1 if the element is new, 0 if it is repeaded. Repeated elements are deleted from the node



Lulea-Compressed Tries (2)

In tries each chunk (a subgroup of the bits of the address) is used as an index in the node to find a prefix or a pointer

In Lulea this is not true

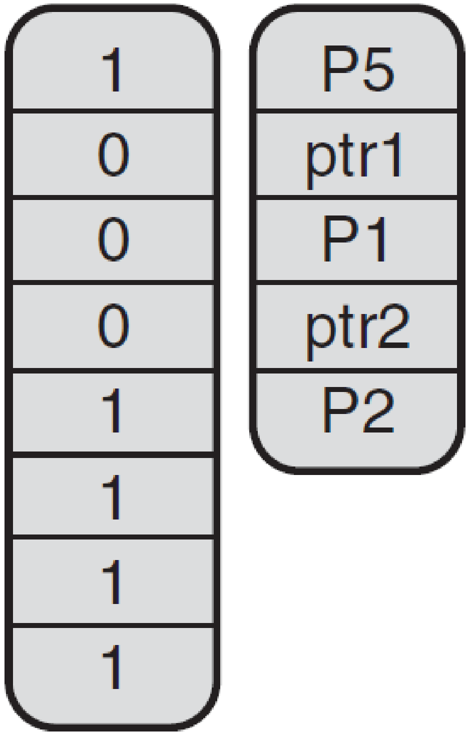
In Lulea the number of 1s in the bitmap up to the position indexed by the chunk is the index in the node

To speed up counting one can precompute cumulative sums with some stride

Problems:

•counting adds one reference

•insertion can be slow (because of leaf pushing)



Tree Bitmap (1)

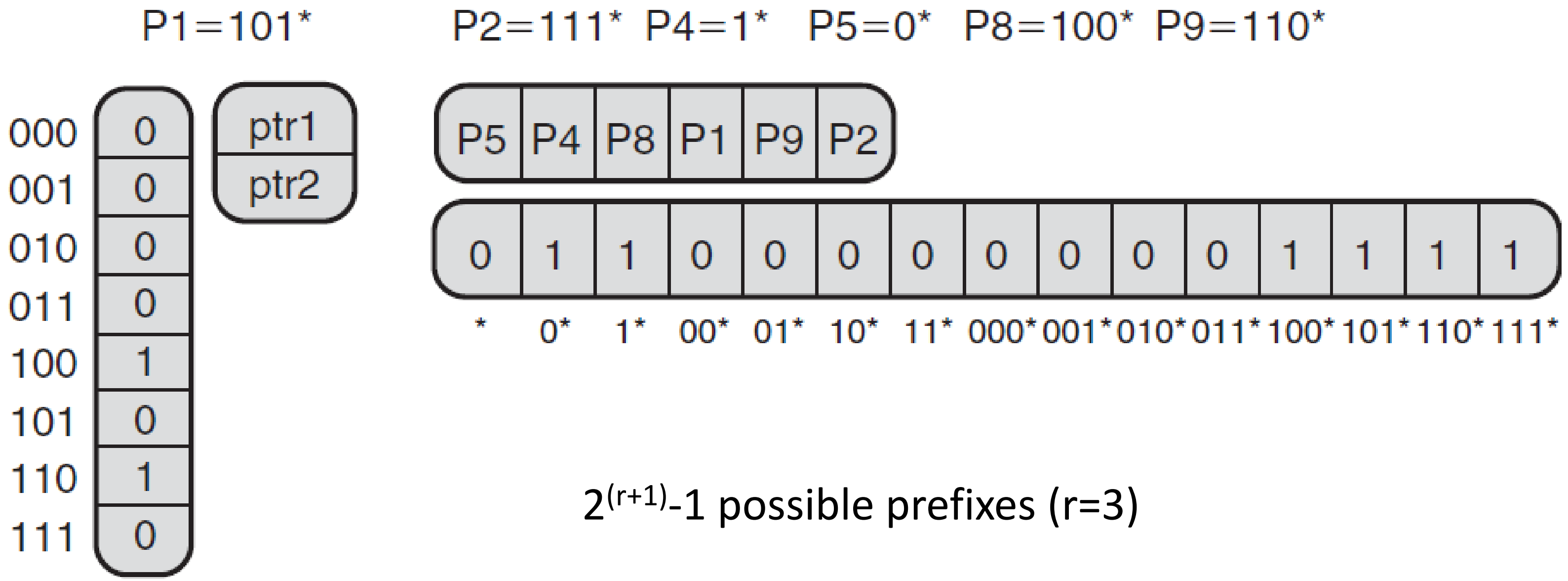
•Tries to achieve storage and speed performance similar to lulea, but with faster insertions

•In lulea we have in each node either a pointer or a prefix. Leaf pushing is necessary to comply with this rule.

•Three ideas in tree bitmaps:

- 1) In tree bitmaps we have two bitmaps per node, one for prefixes and one for pointers (the latter is a trie in linearized form).

Tree Bitmap (1)

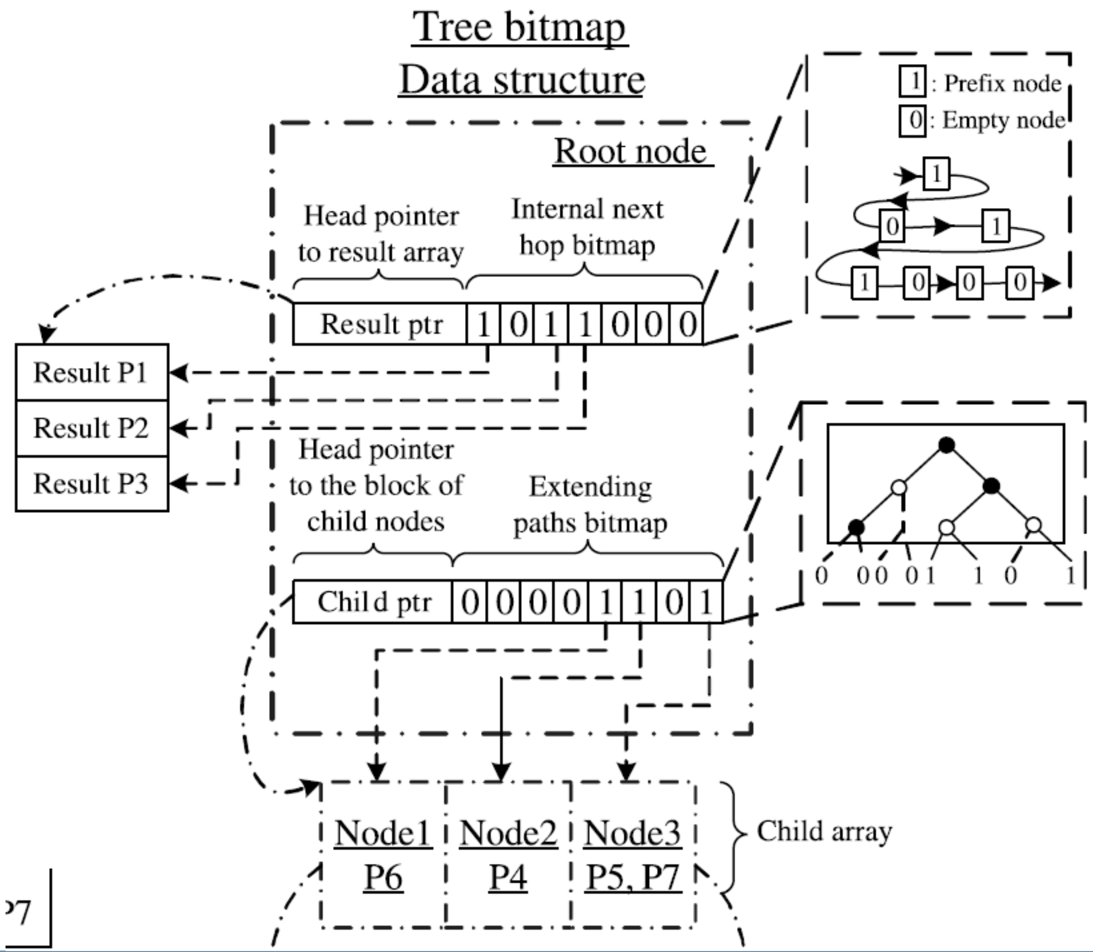


$2^{(r+1)}$-1 possible prefixes (r=3)

Tree bitmap (2)

- 2) keep node sizes small
  - Next hop and other associated data are stored in another array and accessed only when the search terminates
- 3) one memory access per node
  - using small strides (e.g. 4 bits)
  - nodes of fixed size
  - fech entire node with a single read
  - bit counting via combinatorial logic

Tree bitmap (3)

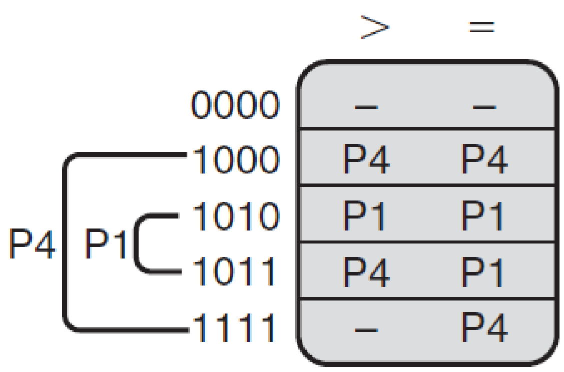


Binary search on ranges

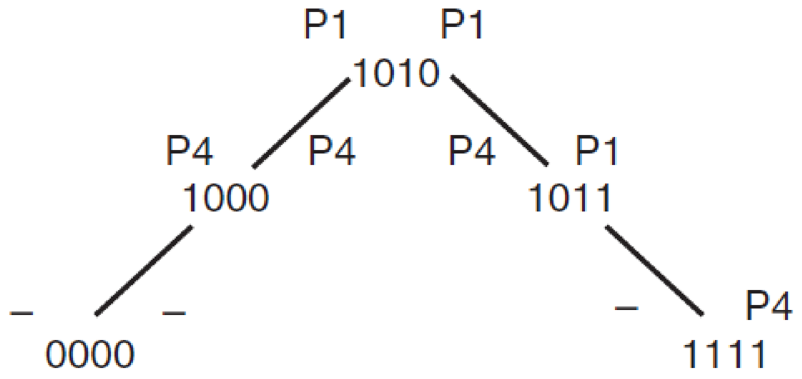
- •Completely different paradigm, not based on tries or trees, but a generalization of exact match
- •Binary search on ranges represents each prefix as a range. N prefixes partition the address space in up to 2N+1 disjoint intervals.
- •Use binary search to find the interval in which a destination address lies.
- •Complexity is O(log 2N); can be reduced with B-trees or other tricks.
- •Insertion/deletion is O(N); but can be reduced to O(log N)
- •Pro: patent-free   Con: worse performance than tries

Binary search on ranges

Prefixes P4=1*, P1=101*



	>	=
0000	–	–
1000	P4	P4
1010	P1	P1
1011	P4	P1
1111	–	P4

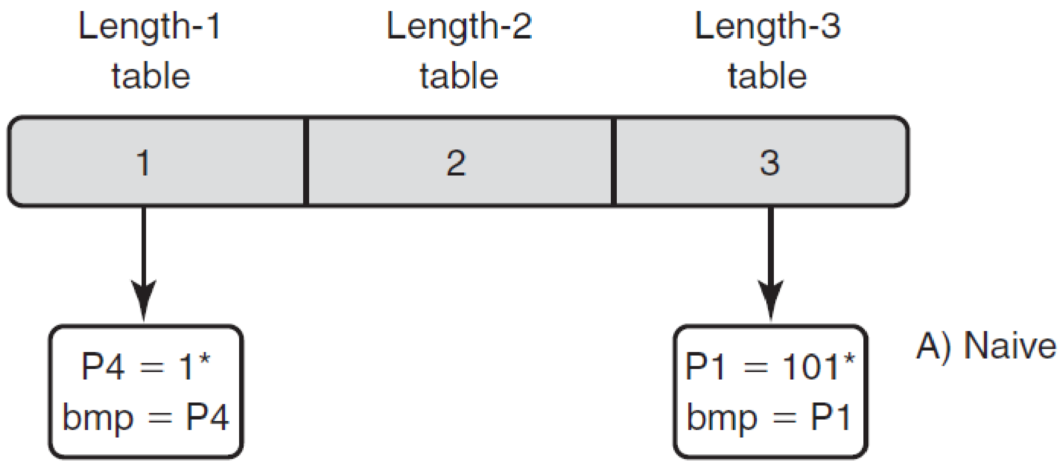


Binary search on prefix lengths (1)

- Variation of exact match with hashing.
- Performs log2(W) hashes, where W is the maximum prefix size.
    - Good performance for IPv6, better than some trie-based solutions
- Uses and array L[i] of hash tables. Each table contains the prefixes of length i
- Naive solution: lookup in L[W] and proceed down to L[1]. Stop if a match is found. Complexity O(W * cost of hash lookup)
- Complexity can be made O(log W) with a binary search on prefix lengths

Binary search on prefix lengths (1)

Routing Table:
1* P4
101* P1



Binary search on prefix lengths (2)

•Algorithm:

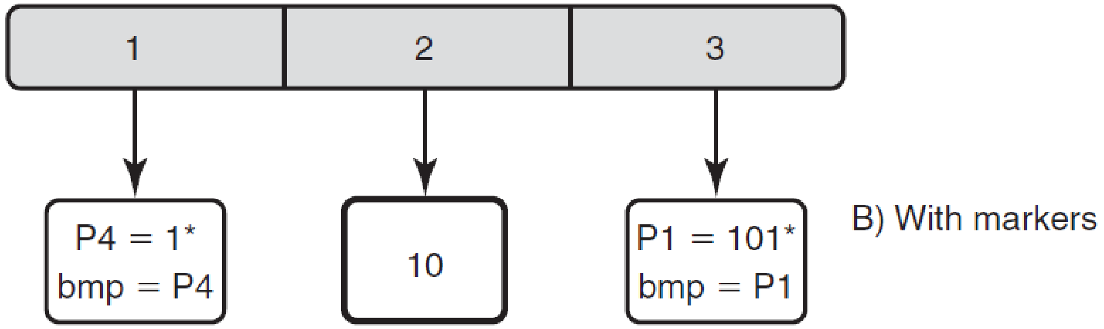
- Start at the median prefix length and search in table
- If a match is found, search for a longer match
- If a match is not found, search for a shorter match

•There is a problem: a prefix of length x may be missing, but a longer match may be present

- A marker for a prefix P must be placed at all the lengths that will be searched for P
- Still not enough, a marker may lead to false leads

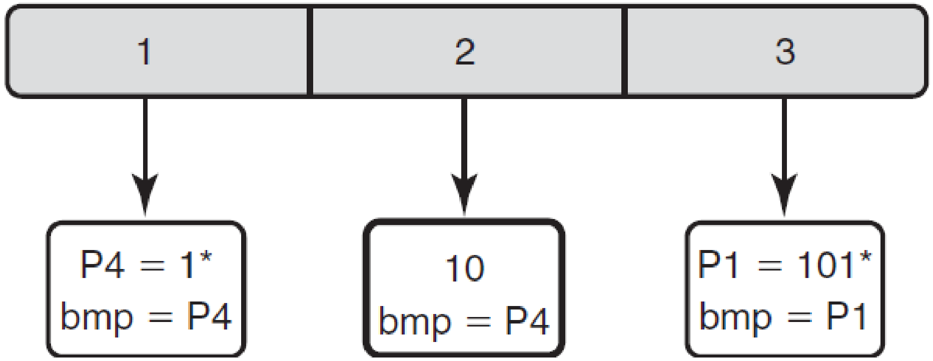
Binary search on prefix lengths (2)

100* not found



Binary search on prefix lengths (3)

- Store with the marker:
  - the max prefix length for that marker
  - the match for that marker precomputed by searching on the left side
- If a search reaches the max length with no results, then the match is stored in the last marker found



C) Markers and precomputation