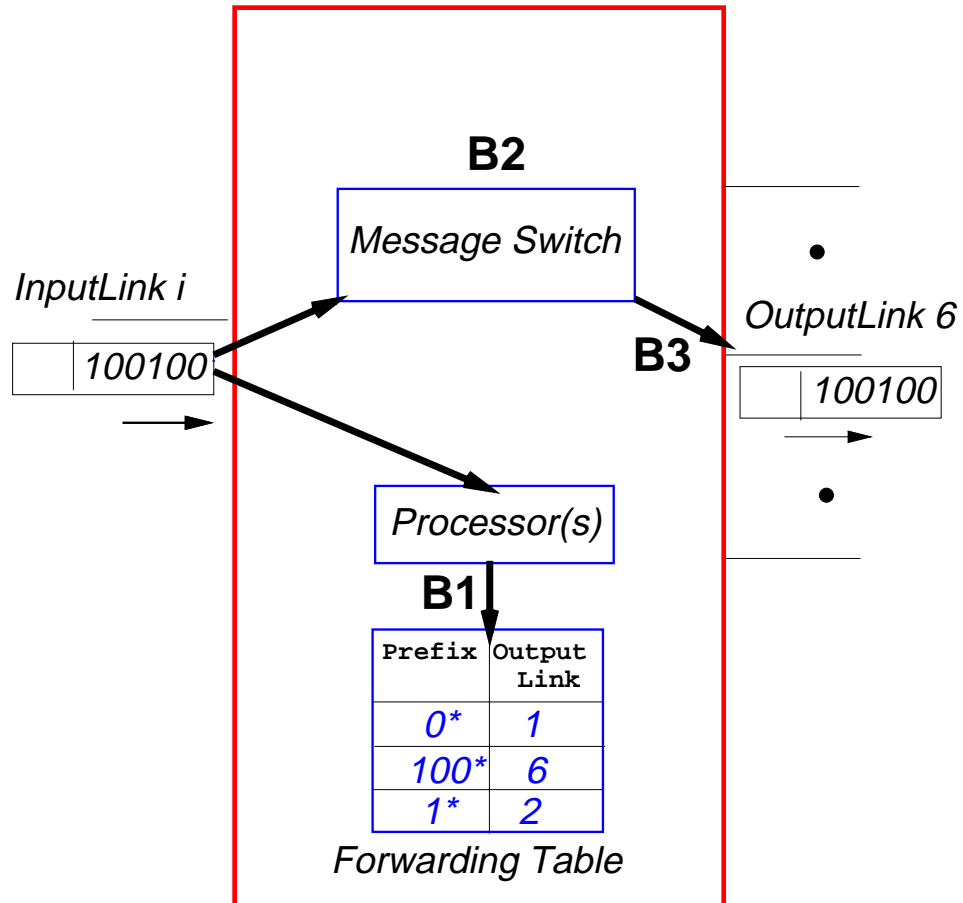


# Recent Results in Best Matching Prefix

George Varghese

October 16, 2001

# Router Model



## Three Bottlenecks for Router Forwarding Performance

Many chores (e.g., checksum calculation, hop count update) but major bottlenecks are:

- Looking up IP destination addresses (best matching prefix) and classification.
- Switching packet to the output link.
- Scheduling (e.g., fair queuing) at output. (also accounting and possible security checks)

## Routing Environment

- Backbone routers: 100,000 prefixes, all lengths from 8 to 32; density at 24 and 16 bits. Unstable BGP implementations may require route updates in 10 msec. May need programmability.
- Enterprise routers: (1000 prefixes), route changes in seconds, bimodal (64, 1519).
- IPv6 plans heavy aggregation but multihoming, Anycast etc, make large backbone databases likely.

## Hardware or Software?

- RISC CPUs, expensive to make main memory accesses (say 50 nsec), cheaper to access L1 or L2 cache. Network processors now very common mostly with multiple RISC cores and using multithreading where each RISC works on different packets to get around memory latency.
- Cheap Forwarding engine (say \$20 in volume). Fast (5-10 nsec) but expensive SRAM and cheap but slower (50-100 nsec) DRAM (roughly six times cheaper per byte). Can use on-chip variants as well. On-chip SRAM of around 32 Mbits with 2-5 nsec available on *custom* chips. Less on FPGAs.

## Sample Database

$$P1 = 10^*$$

$$P2 = 111^*$$

$$P3 = 11001^*$$

$$P4 = 1^*$$

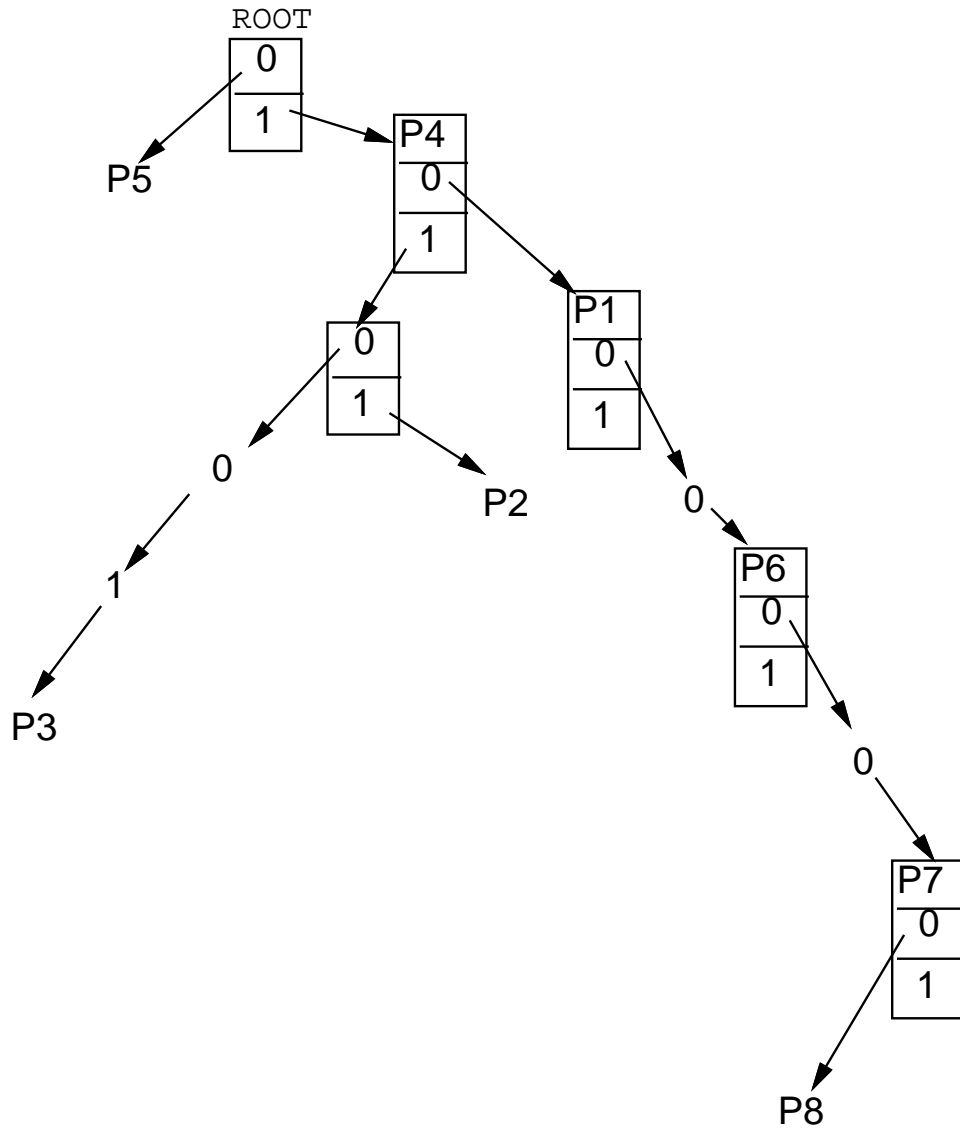
$$P5 = 0^*$$

$$P6 = 1000^*$$

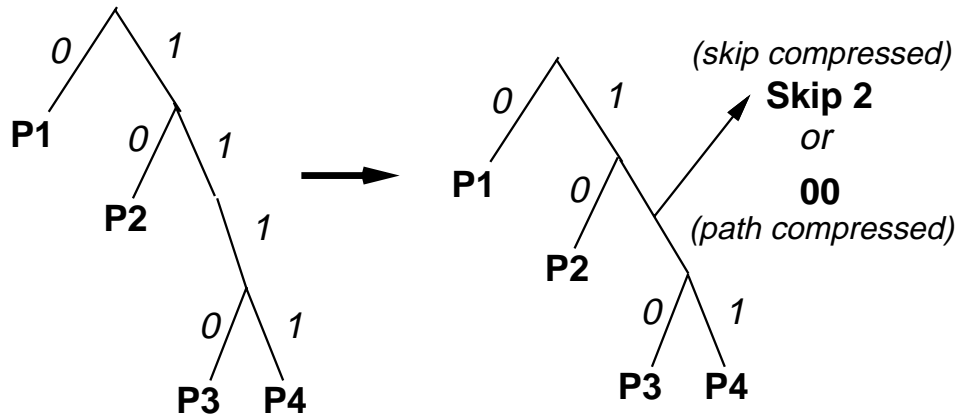
$$P7 = 100000^*$$

$$P8 = 1000000^*$$

# Compressed Trie using Sample Database

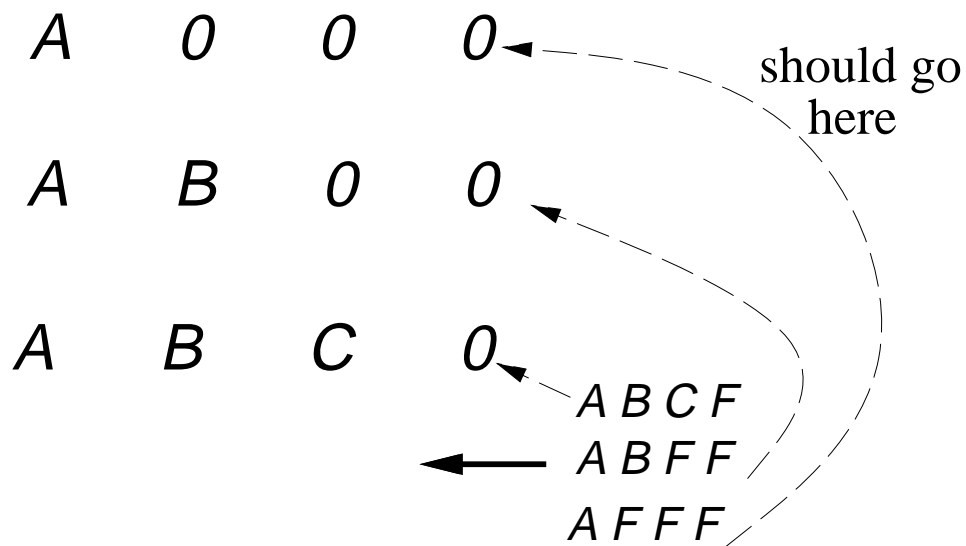


## Skip Count versus Path Compression



- Removing one way branches ensures that number of trie nodes is at most twice number of prefixes.
- Using a skip count requires exact match at end and backtracking on failure. Path compression simpler.

## Binary Search and Prefix Lookup



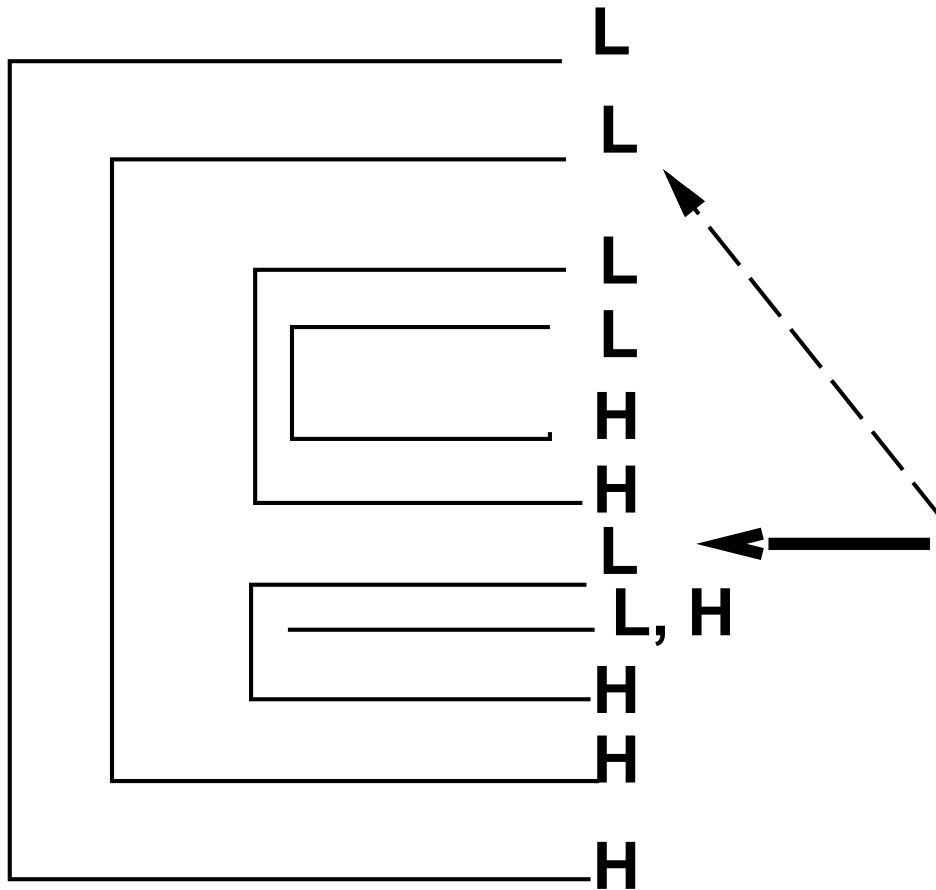
- Natural idea is to encode a prefix like  $A^*$  by padding with 0's. i.e.  $A000$ .
- **Problem:** Several different addresses, each of which should be mapped to *different* prefixes, can end up in *same* range of search table.

### 5.1.6 Modified Binary Search

<i>A</i>	<i>O</i>	<i>O</i>	<i>O</i>	
<i>A</i>	<i>B</i>	<i>O</i>	<i>O</i>	
<i>A</i>	<i>B</i>	<i>C</i>	<i>O</i>	← <i>A B C D</i>
<i>A</i>	<i>B</i>	<i>C</i>	<i>F</i>	← <i>A B D D</i>
<i>A</i>	<i>B</i>	<i>F</i>	<i>F</i>	← <i>A D D D</i>
<i>A</i>	<i>F</i>	<i>F</i>	<i>F</i>	

- **Solution:** Encode a prefix like  $A^*$  as a *range* by inserting *two* values,  $A000$  and  $AFFF$  into search table.
- Now when an identifier falls into a range, each range maps to a unique prefix. Mapping can be precomputed when table is built.

## Why Modified Search Works



- Any range in the binary search table corresponds to the earliest  $L$  that is not followed by an  $H$ . Easy to precompute using a stack.

## Modified Binary Search Table

					>	=
1)	<i>A</i>	<i>O</i>	<i>O</i>	<i>O</i>	1	1
2)	<i>A</i>	<i>B</i>	<i>O</i>	<i>O</i>	2	2
3)	<i>A</i>	<i>B</i>	<i>C</i>	<i>O</i>	3	3
4)	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	3	4
	<i>A</i>	<i>B</i>	<i>C</i>	<i>F</i>	2	3
	<i>A</i>	<i>B</i>	<i>F</i>	<i>F</i>	1	2
	<i>A</i>	<i>F</i>	<i>F</i>	<i>F</i>	-	1

- Need to handle case of exact match (= pointer) separately from case where key falls between two entries (> pointer).

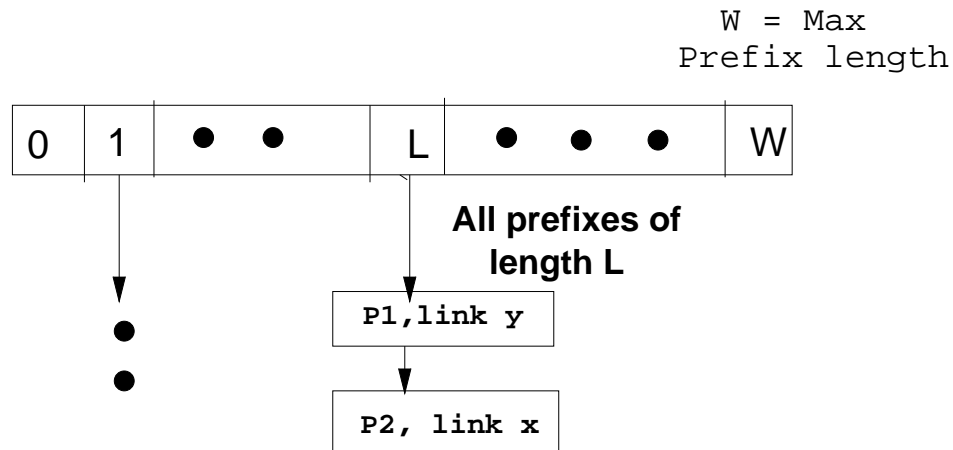
## Controlled Prefix Expansion

Reduce a set of prefixes with  $N$  distinct lengths to equivalent prefixes with  $M < N$  distinct lengths.

Recursively apply two simple ideas:

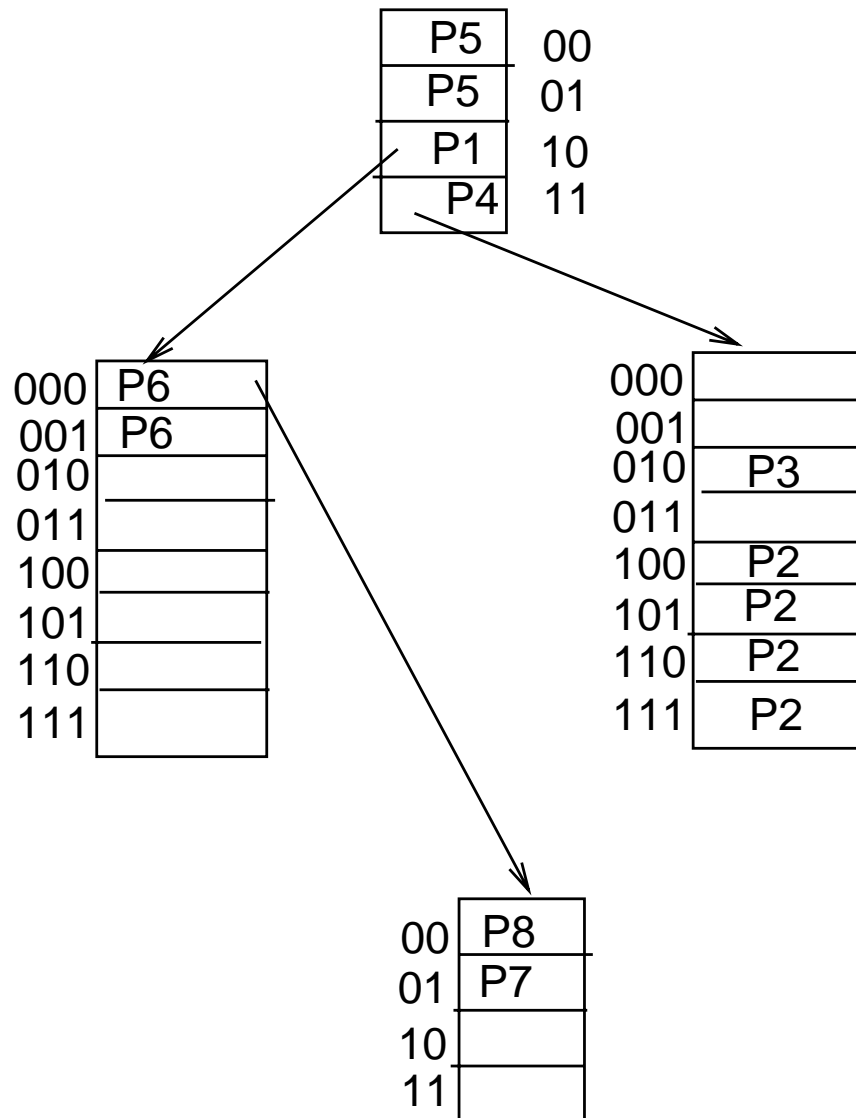
- **Expansion:** Expand prefix  $P$  of length  $L$  into two equivalent prefixes  $P0$  and  $P1$  of length  $L + 1$ .
- **Capture:** When a prefix expanded from Level  $L$  to  $L + 1$  has same value as a prefix at Level  $L + 1$ , discard expanded prefix.

# Expansion Algorithm



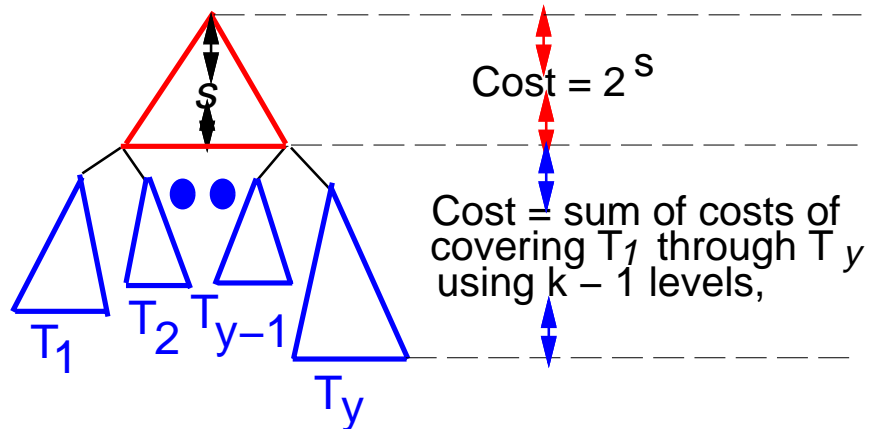
- Place all prefixes in table by length. Pick expansion levels  $L_i$ .
- Start expanding from 1 upward skipping expansion levels. At step we expand prefixes at Level  $i$  to  $i + 1$ , taking into account capture.

## Expanded Version of Trie



- Reduces search time from 7 memory accesses worst case to 3 accesses.

## Optimal Expanded Tries

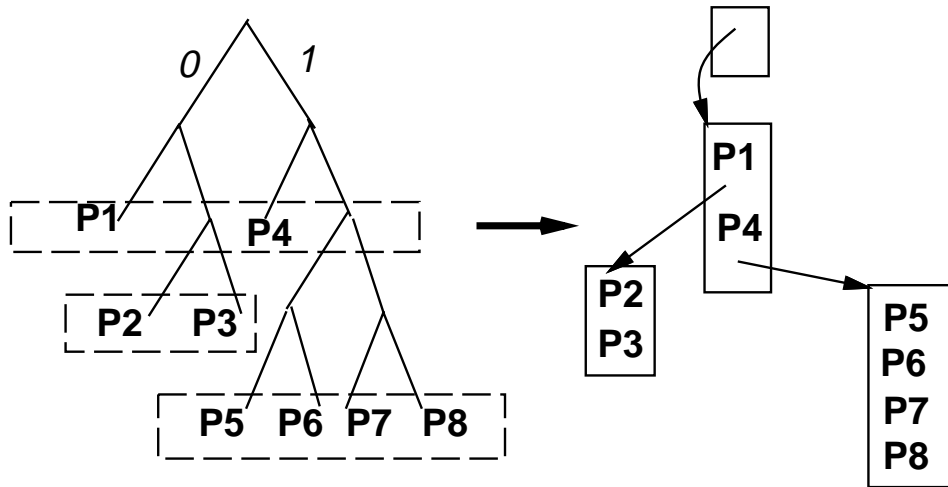


- Pick stride  $s$  for root and recursively solve covering problem for the subtrees rooted at Trie Level  $s + 1$  using  $k - 1$  levels.
- Choose stride to make update times bounded (say  $< 1$  msec)

## Stanford Algorithm

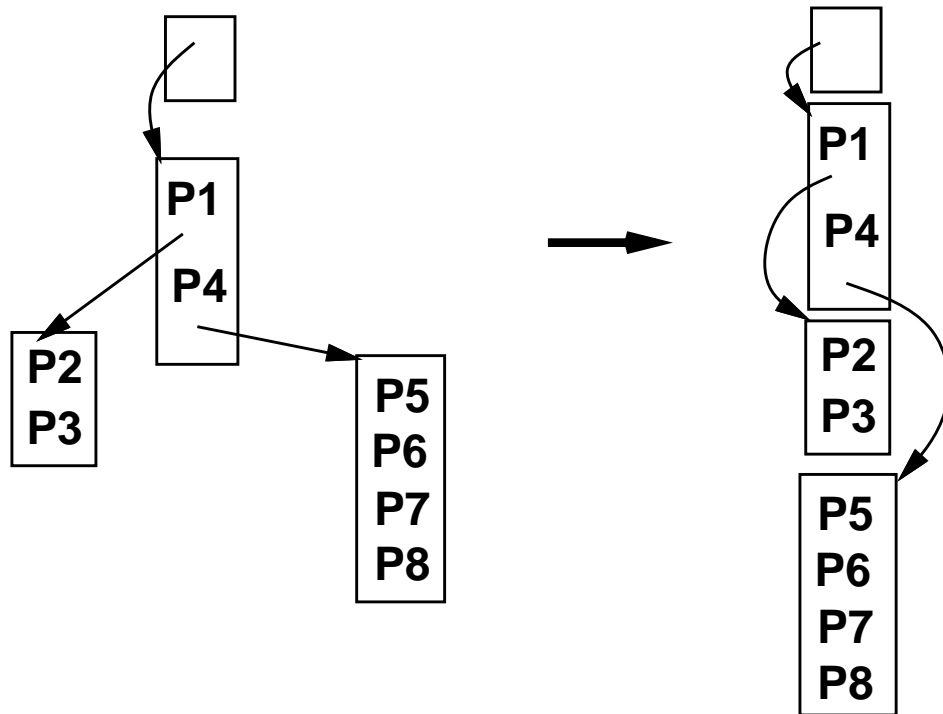
- Observes that most prefixes are 24 bits or less and that DRAM with  $2^{24}$  locations will be widely available soon.
- Suggests starting with a 24 bit initial array lookup. To handle prefixes of larger length they use a 24-8-8 multibit trie.
- A problem is updating up to  $2^{24}$  locations when a prefix changes. Suggest hardware techniques for parallel updates.

## Level Compressed Tries



- Forms variable stride trie by decomposing 1-bit trie into *full* subtrees. Variable stride tries more flexible (e.g., 8,2;2 versus 4,2,4)
- Leads to *less* efficient use of memory and no *tunability* (e.g., allowing memory to be traded for speed)

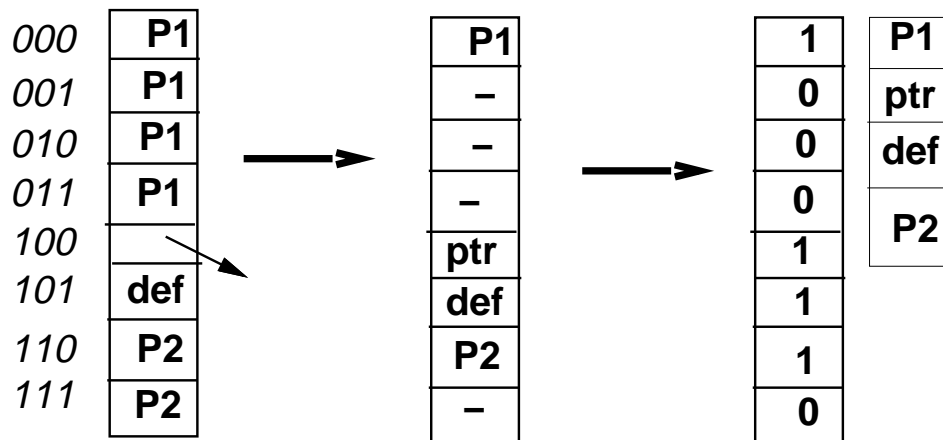
## Array Representation of LC Tries



- Array layout (by trie level) slows down insertion and deletion.
- Higher trie height (average  $\approx 6$ ) and skip count compression (backtracking) leads to slower search times.

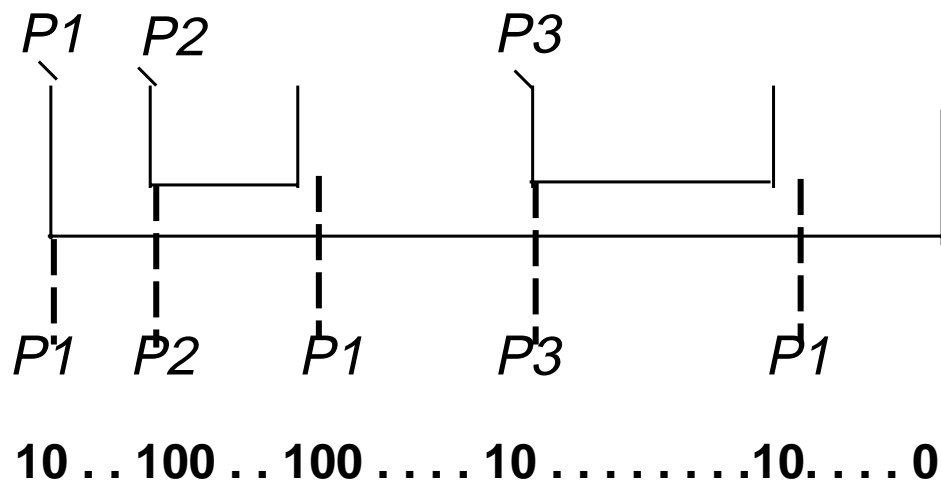
## Lulea Scheme: Node Compression of Multibit Tries

$$P1 = 0^*, \quad P2 = 11^*, \quad P3 = 10000^*$$



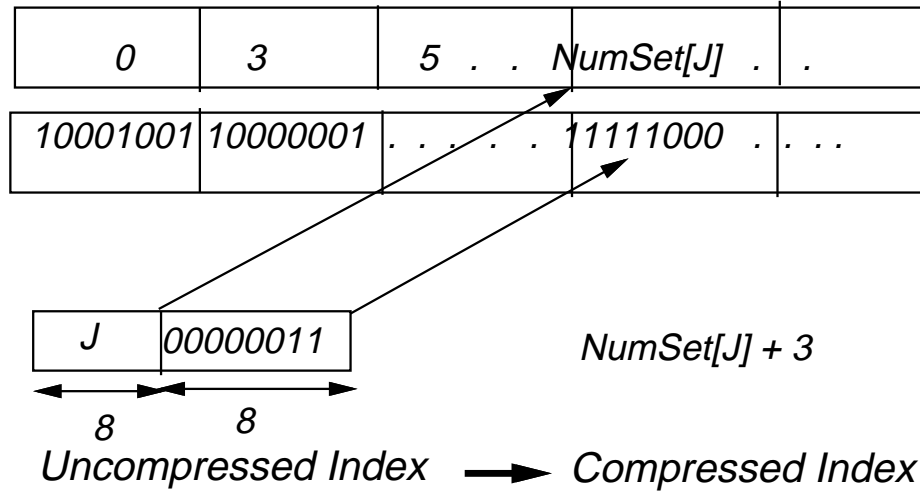
- Expansion can lead to a lot of filled entries in each multibit trie array.
- If we (effectively) do linear search for the earliest array entry with a valid prefix, we can use bit map compression.

## Why Lulea Compression is Effective



- If we consider each prefix as a range, the number of breakpoints in the function that maps indices to prefixes is at most twice the number of prefixes.
- Also basis for binary search on prefixes (Lampson et al, Infocom 98)

## Why Lulea Speed Loss is not too bad

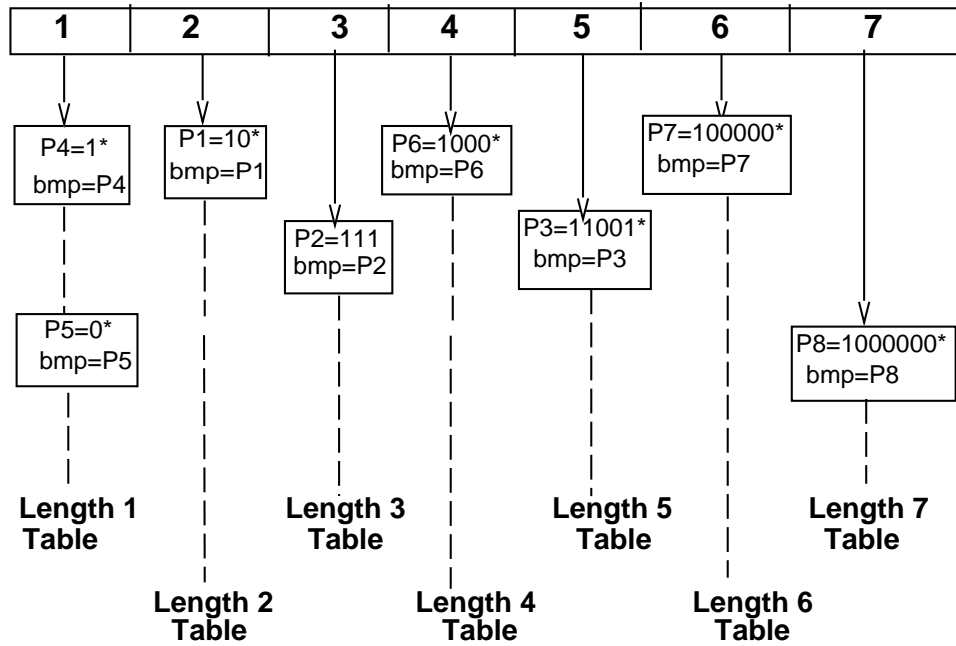


- For large arrays of bits we store with each word  $j$ ,  $NumSet[j]$ , the number of bits set in words  $< J$ . High order bits of index yield  $j$  and counting bits in Word  $j$  give compressed Index.
- Takes 3 memory accesses in software and 2 in hardware.

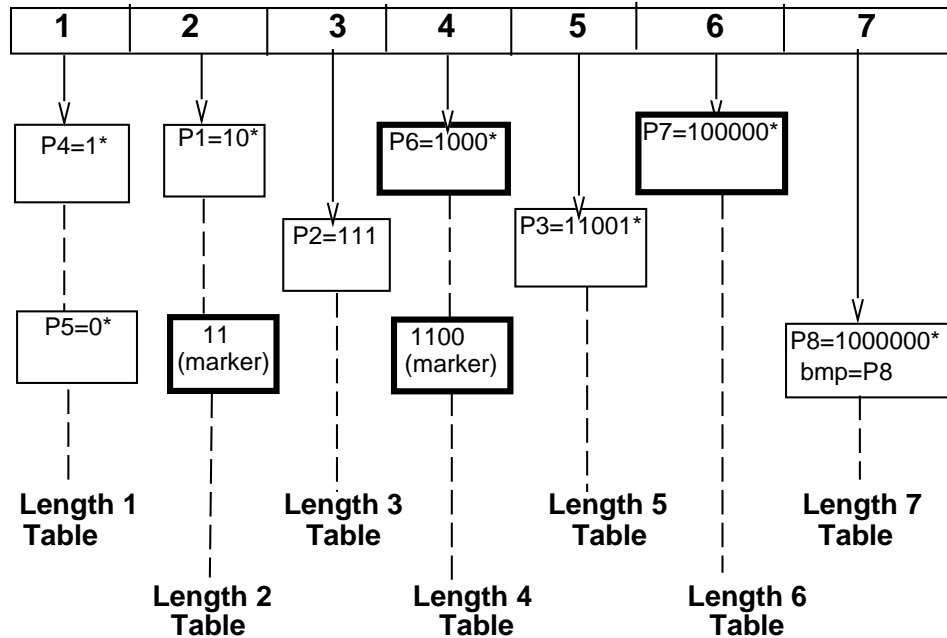
## Wash U Scheme: Binary Search of Possible Prefix Lengths

- Naive search of all possible prefix lengths ( $W$  exact matches,  $W$  possible lengths). Want to do binary search ( $\log_2 W$  exact matches).
- Different from binary search on prefix ranges that would take ( $\log_2 N$  memory accesses, which is worse for large  $N$ )
- Need to add *markers* to guide binary search and precompute best matching prefixes for these markers.

# Why Markers are Needed



## How backtracking can occur

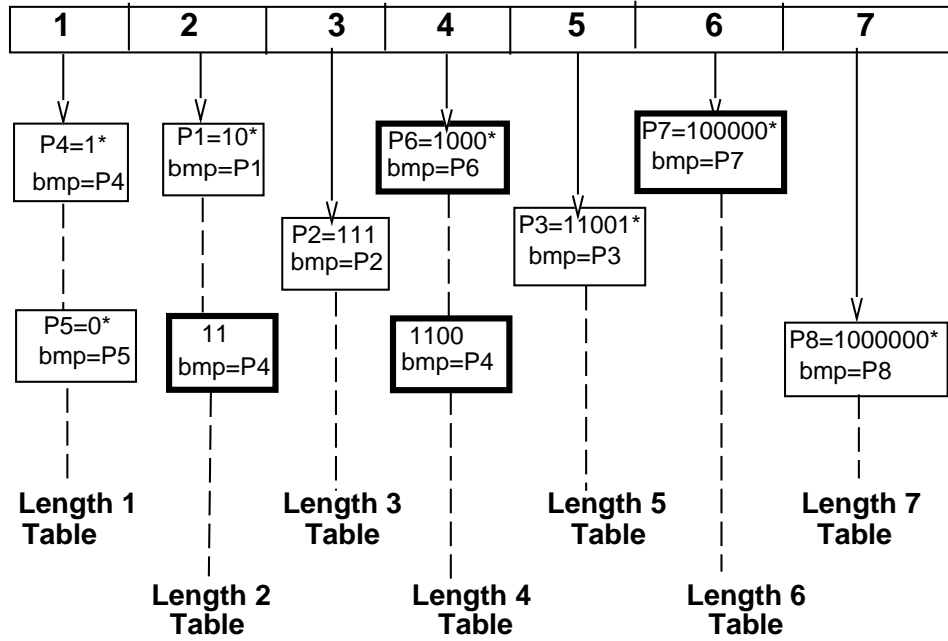


- Markers announce “Possibly better info on right. No guarantees.” Can lead to wild goose chase.

## Avoiding Backtracking by Precomputation

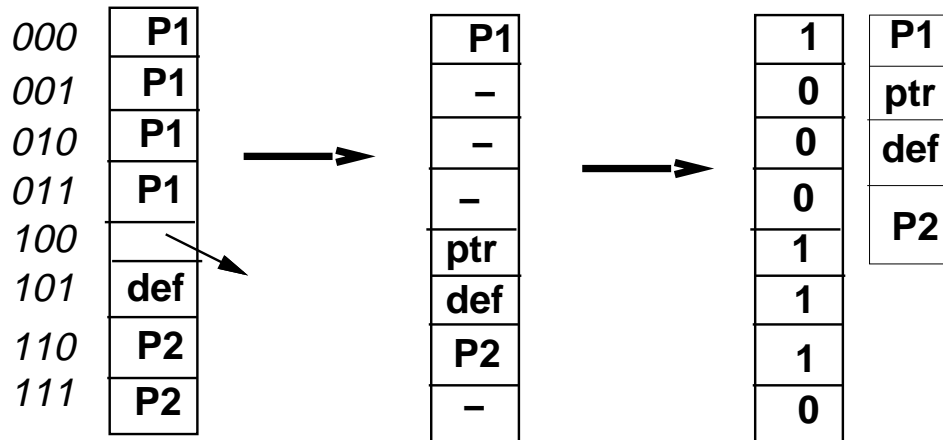
- We precompute the best matching prefix of each marker as *Marker.bmp* .
- When we get a match with a marker we do search to the right but we remember the *bmp* of the latest marker we have encountered.

# Final Database



## Eatherton Scheme: Compression alternative to Lulea

$$P1 = 0^*, \quad P2 = 11^*, \quad P3 = 10000^*$$



- Lulea does leaf pushing and uses huge arrays (16 bits). Hard to update. Instead, Eatherton does no leaf pushing and uses small arrays (no larger than 8 bits, even smaller).
- Encodes pointers and next hops using two *different arrays* using two different bit maps. Gets around use of smaller strides by accessing bit maps from previous node and accessing all bit maps in one access. 1 memory access per node compared with 2 for Lulea. Can afford half the

stride length.

- Lots of tricks to reduce memory access width.  
One example, the access to the next hop pointer can be done lazily at the end. Described in Eatherton's M.S. thesis at UCSD CSE, EE department. Parts of thesis may be patented by Cisco as Eatherton joined there.

## 2001 Conclusions

- Prefix tables getting very large. Up to 500,000 seems required.
- Controlled prefix expansion works fine with DRAM. Lots of licensees of Wash U patent. Using RAMBUS like technologies with net processors works fine.
- With SRAM like technologies, want to limit use of fast memory. Can do simple algorithms and make them parallel using say pipelining (Juniper patent seems to use unibit tries?) or using SRAM.
- Binary search not too bad using B-trees with large width (uses memory well, can search in rough  $\log_W(N)$  time, where  $W$  is number of prefixes that can be compared in one access.
- Lulea's memory use is remarkable. Selling custom solutions (maybe more expensive) for their patent: [www.effnet.com](http://www.effnet.com). But slow updates and requires L2 cache. Alternative bitmap compressions like

Eatherton may be effective though patent coverages are very unclear.

- Wash U patents for prefix expansion. fickense@msnotes.wustl.edu. Only controlled expansion mostly used because of non-determinism of Binary Search on Prefix lengths
- Tag switching still alive and well for QoS (classification tags really) in MPLS. Classification mostly done today by CAMs and custom hardware (PMC Sierra's ClassiPI is an alternative). CAMs still too small and power-hungry for lookups.