Router Architecture : Efficient Algorithms

Sandhya Dasu

Efficient Implementation of a Statistics Counter Architecture

– Sriram Ramabhadran -George Varghese

Is packet counting useful?

- Measuring categories of traffic
- Capacity planning
- Identify bottlenecks in network core
- Ratio of one packet type to another
- Identify/analyze attacks by counting packets for commonly used attacks (ICMP request-response in smurf attacks)

» Contd…

Is packet counting useful? contd…

- To decide peering relationships
- Accounting based on traffic type

Legacy Routers

- Provide per-interface counters – queried by SNMP
- Count only aggregate of all counters on an interface – so difficult to do traffic engineering
- Only crude form of accounting possible

New Technology

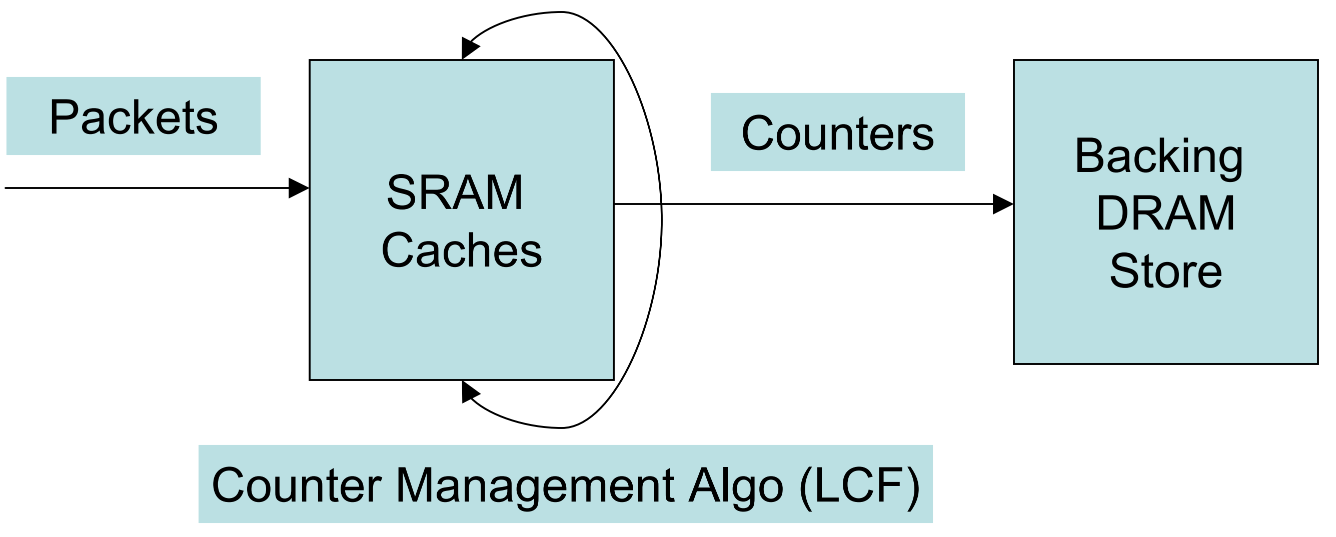
- Juniper’s filter based accounting
- Cisco’s Netflow based accounting – 5-tuple based and Express Forwarding

Why is counting hard?

- Large number of counters (currently 500,000 prefixes, future…)
- Multiple counter updates per packet
- High speeds – match line rates OC192(10Gbps) to OC768(40Gbps)
- Large counter widths

Building on work…..

Shah et al's Statistics Counter Architecture



Their hybrid arch using LCF CMA

- DRAM is used for all statistics counters – Full sized counters
- SRAM is used to support counter updates at line rates – Smaller sized counters
- Largest Counter First Counter Management Algo used to decide which counter gets written to the DRAM – exact sorting
- Highlight – Uses optimal amount of SRAM

Problem with this approach

- CMA needs to find the largest counter to be updated to the DRAM – needs sorting of counters
- Some solutions –
  - Examine each value
  - Index data structure that orders based on counter values – Eg. P-Heap
- Does not take care of counter increments greater than 1.

LR(T) CMA

- Largest Recent with Threshold (T)
- Removes sorting bottleneck – approximate bin sorting
- Keeps a bitmap that tracks counters that are larger than threshold T
- Practically realizable with 2 bits extra per counter
- Uses same optimal amount of SRAM as LCF
- A simple pipelined data structure

LR(T) Algorithm

- All updates made to counters in SRAM
- After b updates, CMA picks one counter that is written to DRAM
- Updated counter is reset to 0
- “b” depends on relative access times of DRAM and SRAM

LR(T) Algorithm

Let j be the counter with the largest value after the last cycle of b updates

- If value[j] >= T,
  - Update counter j to DRAM and set it to 0 in the SRAM
- If value[j] < T ,
  - Find another counter with value atleast T and update to DRAM
  - If no counter found, then update counter j to DRAM

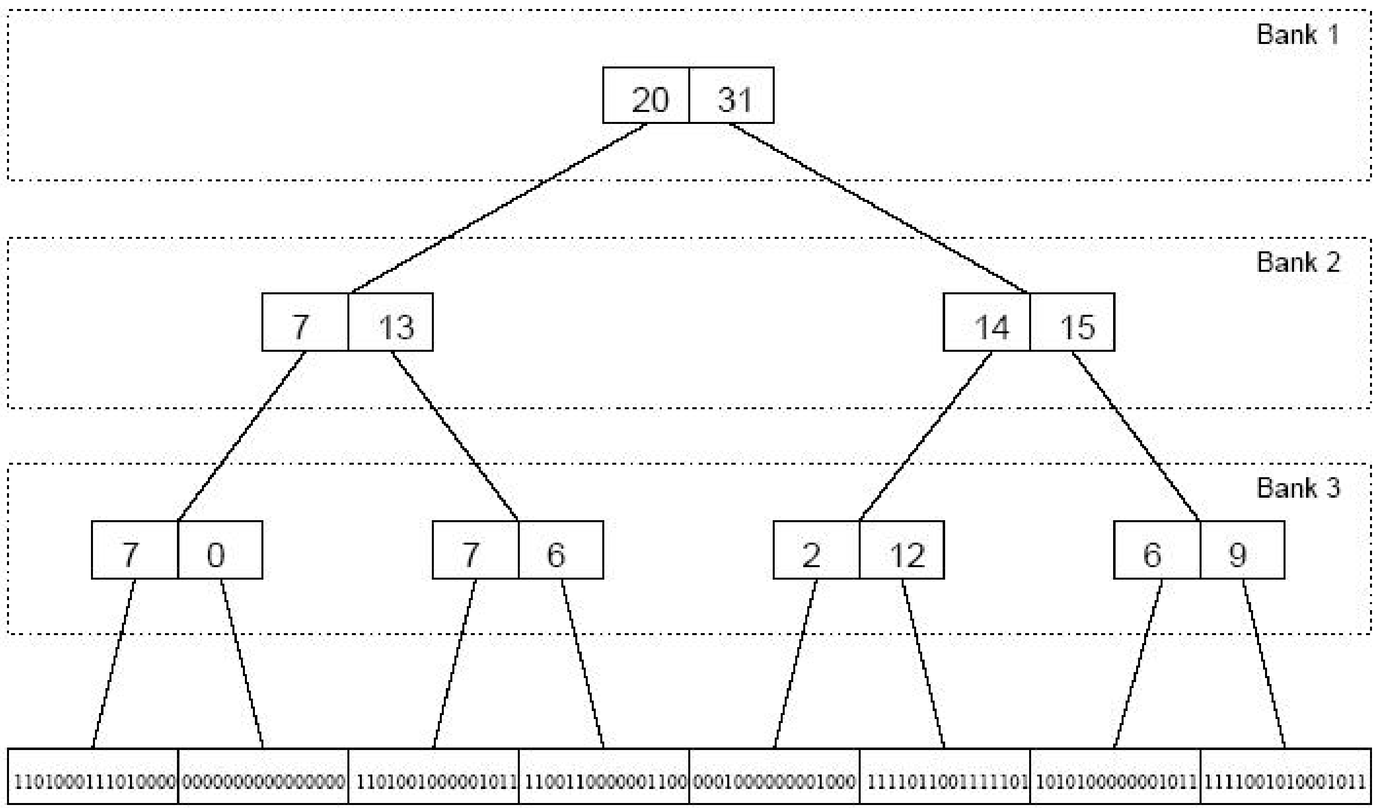
Implementation of LR(T) CMA Using Aggregated Bitmap

- A bitmap is used to indicate if a counter is above or below the threshold
- The following operations are required to be implemented on the bitmap to support LR(T)
  - Add(i) – To update bit for a counter to indicate its value is above threshold
  - Delete(i) – After updating a counter's value it, this operation is performed to indicate that its value is now below T
    - » Contd…

Implementation of LR(T) CMA Using Aggregated Bitmap contd..

- Test(i) – to check if a counter's value is above T
- Find(i) – to find a counter with value above T

Aggregated bitmap for N elements and word size W



N = 128 W = 16

Tree Structure to Aggregate Bitmap Information

- Leaves of binary tree are formed by N/W nodes where N is total number of counters, W is the word size
- For a tree of height h+1, 2^h should be equal to N/W
- For a node with children as leaf nodes, lcount and rcount are number of bits set in the lchild and rchild respectively
    - » contd

Tree Structure to Aggregate Bitmap Information contd..

- For a node whose children are not leaf nodes, the lcount is the sum of the lcount and rcount fields of its left child and rcount…
- Functions on the bitmap can be performed on a top-down traversal of the tree
- Each of the internal nodes does not contain pointers to lchild and rchild, only lcount and rcount values

Memory for the bitmap

- Total number of node = $2^{(h+1)} - 1$
- Total memory = $(2^{(h+1)} - 1)$ W

$$= (2N/W - 1)W \quad = 2N - W < 2N$$

So, 2 most 2 bits per element

More Implementation Details

- Each level of the bitmap tree can be stored in a different memory bank allowing for pipelined implementation.
- Maintain largest counter and its value – an on-chip register in the CMA logic
- All counters above threshold T – using the aggregated bitmap stored in a separate SRAM
- Large counter updates – Update counter in each cycle with a probability

Comments?

- Ties broken arbitrarily coupled with the fact that only one counter update to DRAM per cycle may result in counter overflows.
- What happens to the bitmap in that case?
- Large counter updates …
- Optimal amount of SRAM? Do not take the 2 extra bits into consideration – an issue only in theory

Tree Bitmap : Hardware/Software IP Lookups with Incremental Updates

W. Eatherton, Z. Dittia,

G. Varghese

Terminology

- Wire Speed IP Forwarding – Ability to perform longest prefix matches for a burst of smallest size packets like ACKs at line rate
- CAMs

Some numbers

- Current routers have about 50,000 prefixes and growing….(hundreds of thousands soon)
- Wire Speed Forwarding at OC-192c rates requires 24 million IP lookups / second

Requirements of an ideal IP lookup scheme

- Requires few memory accesses to perform wire speed forwarding
- Small amount of high speed memory
- IP lookup algo implemented as a single chip solution
- All data structures to accomplish this should fit inside max on-chip memory
- Determinism in terms of lookup speed, storage and update times.
- Additional – Tunable software implementation

Block diagram of Lookup Reference Design

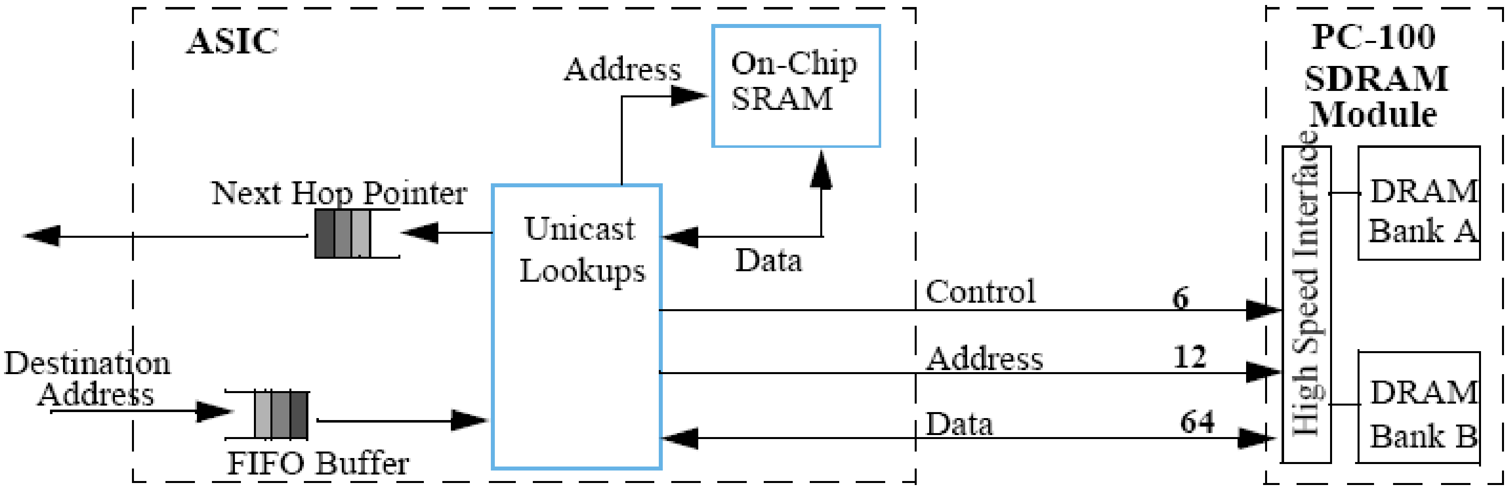


Figure 1: Block Diagram of Lookup Reference Design

Existing Trie based schemes

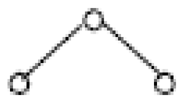
- Unibit Tries
- Expanded Tries
  - Controlled Prefix Expansion with(out) Leaf Pushing
- Lulea

Unibit Trie Representation

Legend

- ● Prefix Node
- ○ Place Holder Node

next bit=0 next bit=1



Prefix Database

P1	*
P2	1*
P3	00*
P4	101*
P5	111*
P6	1000*
P7	11101*
P8	111001*
P9	1000011*

Unibit Trie

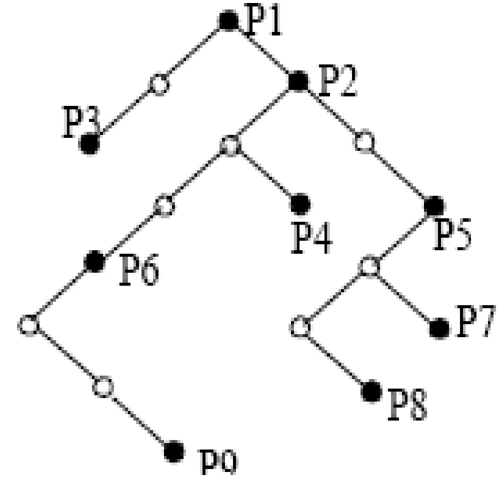
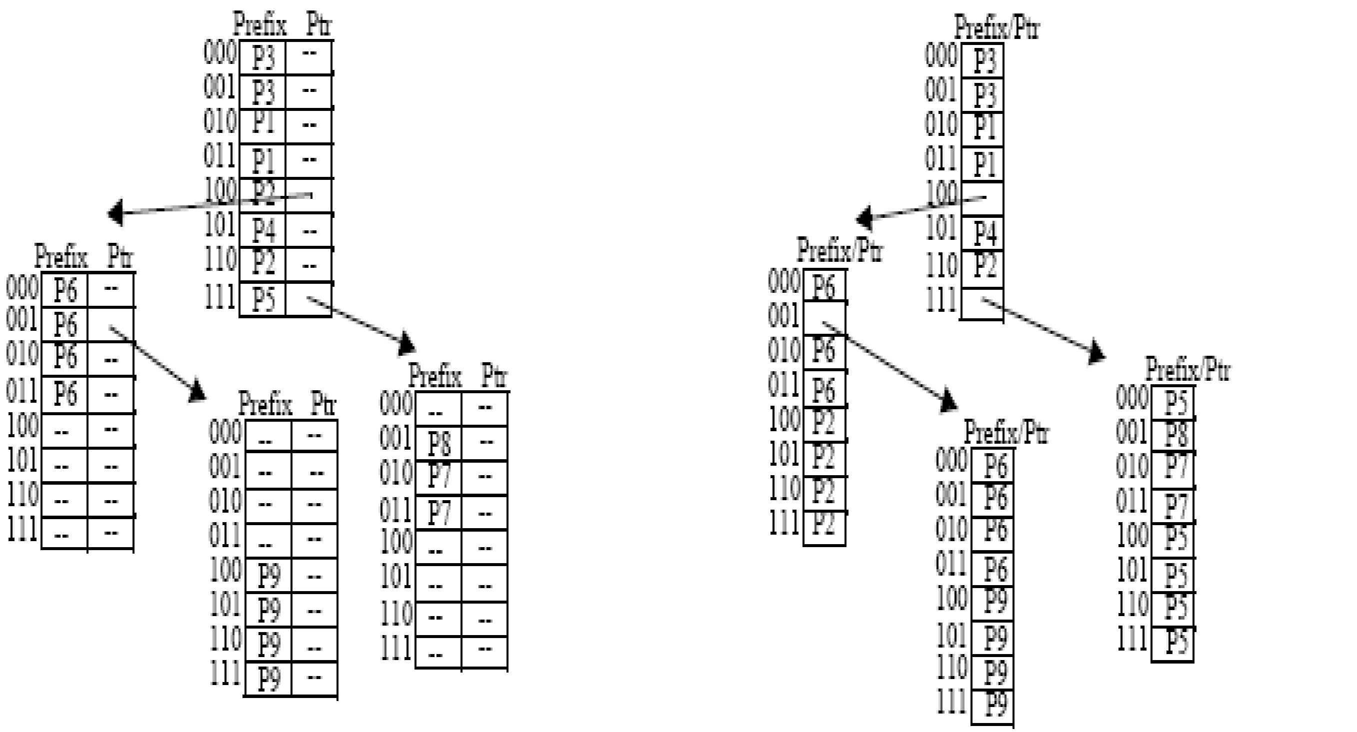


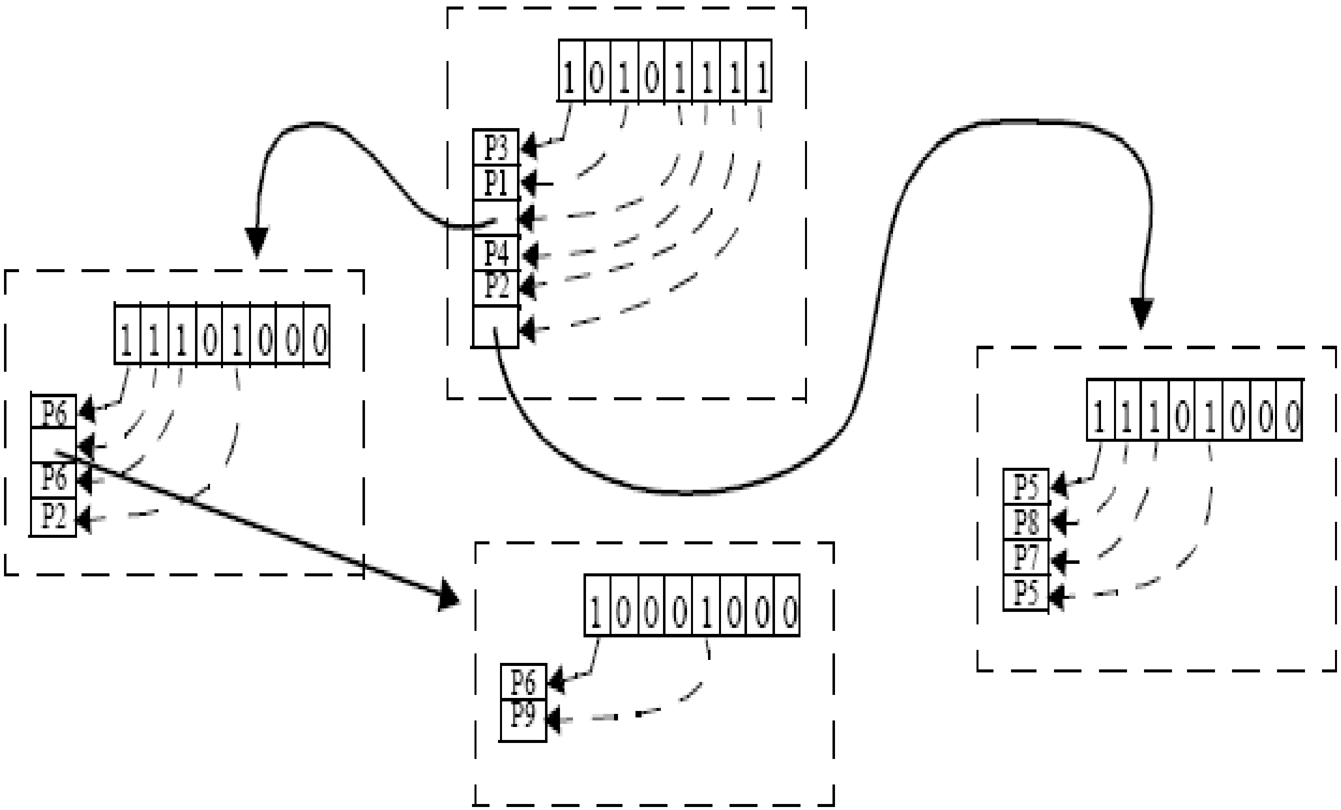
Figure 2: Sample Database with Unibit Trie Representation

Controlled Prefix Expansion



A) Controlled Prefix Expansion w/out Leaf Pushing B) Controlled Prefix Expansion with Leaf Pushing

Lulea Scheme



C) Lulea

Tree Bitmap Algorithm Goals

- Multibit Tree based
- A multibit node :
  - Points to children multibit nodes
  - Produces next hop pointers for longest matching prefixes that exist within that node
- Uses smaller strides (max 8 bits) to keep update times small
- Single node is retrieved by a single page access

Sample Database with Tree Bitmap

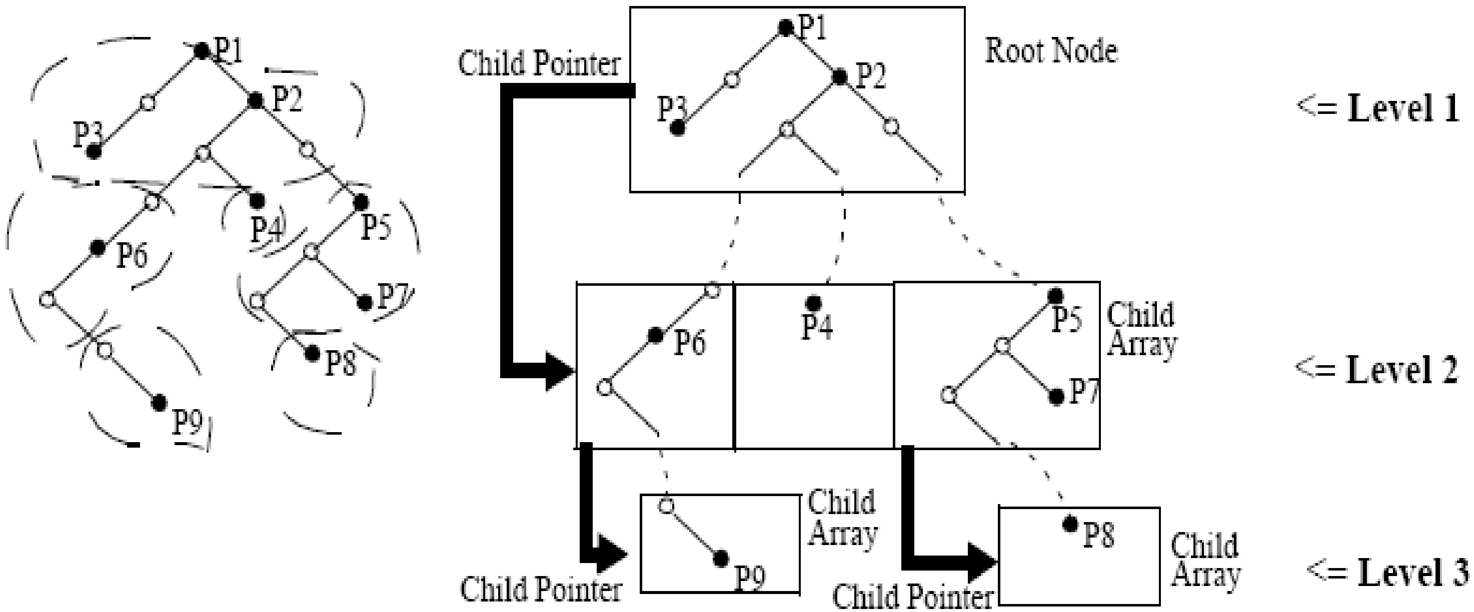


Figure 4: Sample Database with Tree Bitmap

Tree Bitmap Algorithm

- All child nodes of a trie node are stored contiguously
- 2 bitmaps per trie node:
  - Internal Tree Bitmap – for internally stored prefixes
  - Extending Paths Bitmap – for external pointers
- Keep the trie nodes small – use separate array to store next hops for internal prefixes (result array)
- A lazy strategy to access result array

Multibit Node Compression with Tree Bitmap

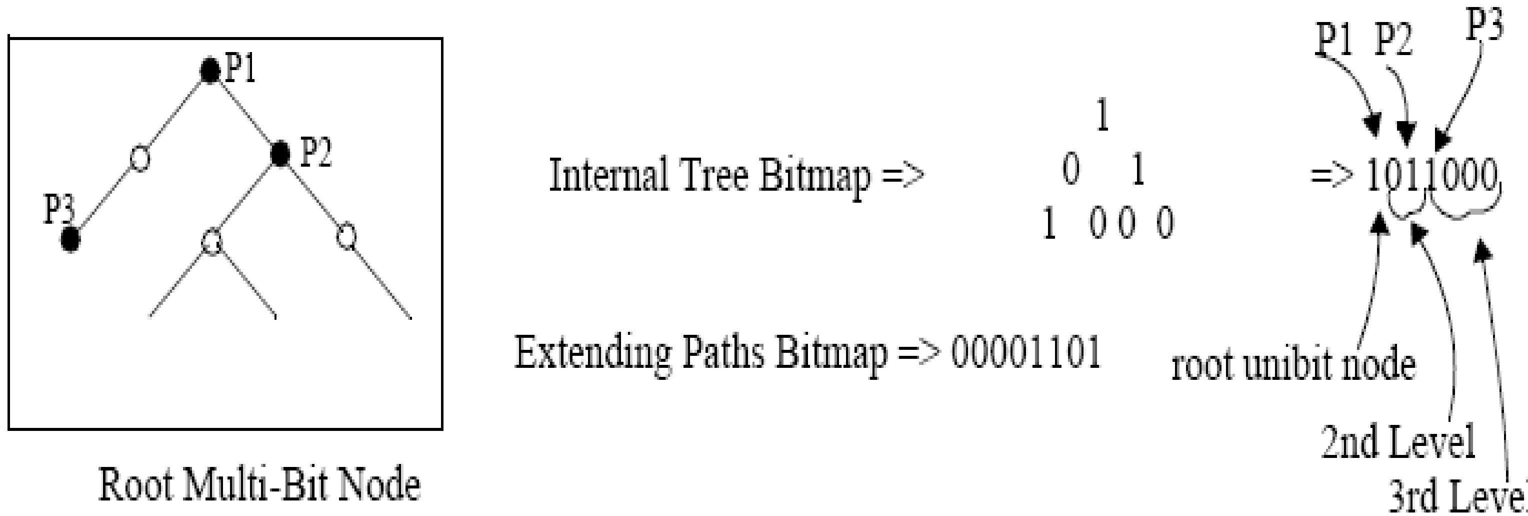


Figure 5: Multibit Node Compression with Tree Bitmap

Tree Bitmap Search Algorithm

```
node:= root;  (* node is the current trie node being examined; so we start with root as the first trie node *)
i:= 1;   (* i is the index into the stride array; so we start with the first stride *)
do forever
   if (treeFunction(node.internalBitmap,stride[i]) is not equal to null) then
                (* there is a longest matching prefix, update pointer *)
      LongestMatch:= node.ResultsPointer + CountOnes(node.internalBitmap,
              treeFunction(node.internalBitmap, stride[i]));
   if (externalBitmap[stride[i]] = 0) then    (* no extending path through this trie node for this search *)
       NextHop:= Result[LongestMatch]; (* lazy access of  longest match pointer to get next hop pointer *)
       break; (* terminate search)
    else (* there is an extending path, move to child node *)
       node:= node.childPointer + CountOnes(node.externalBitmap, stride[i]);
       i=i+1; (* move on to next stride *)
end do;
```

Optimizations

- The above mentioned scheme required 128 bytes per trie node……hence need optimizations
- Initial Array Optimization
- End Node Optimization
- Split Tree Bitmaps
- Segmented Bitmaps

End Node Optimizations

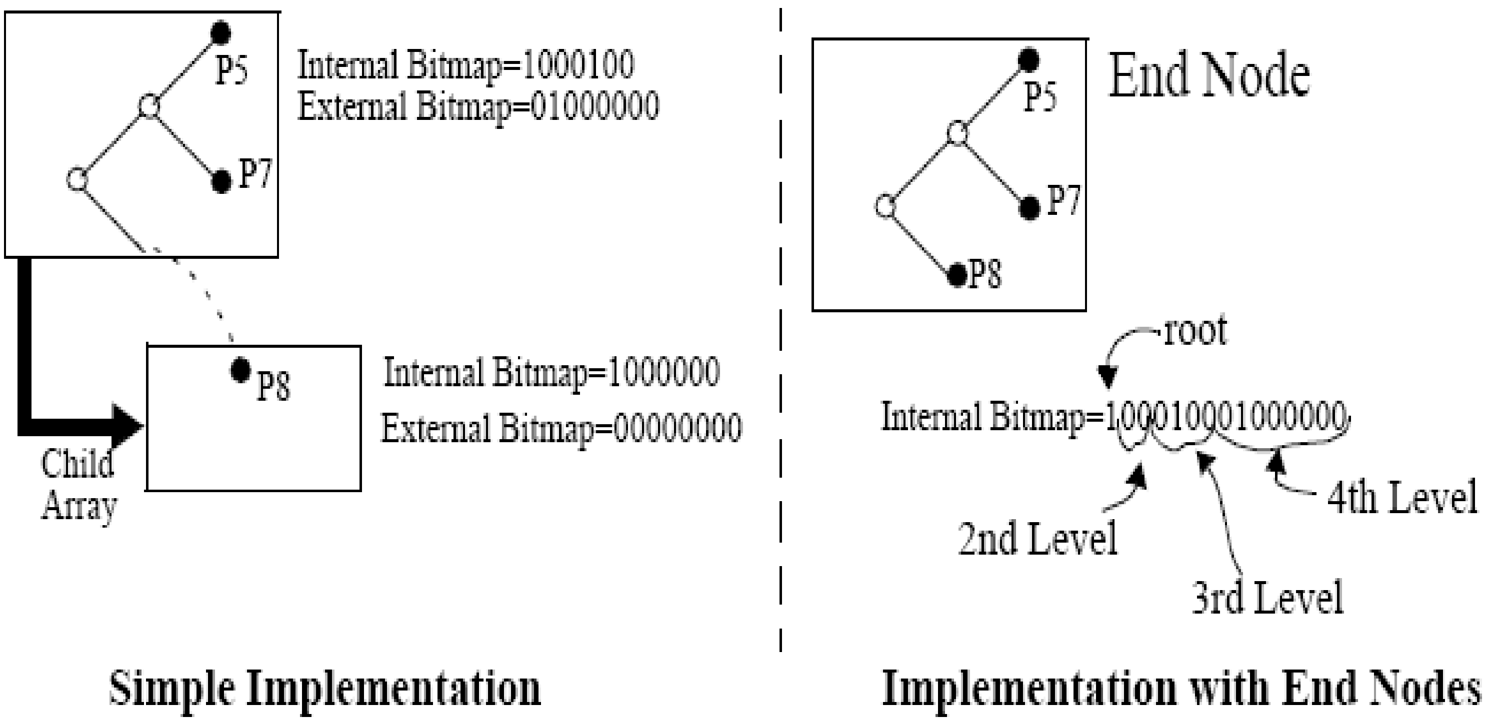


Figure 7: End Node Optimization

Split Tree Bitmap Optimization

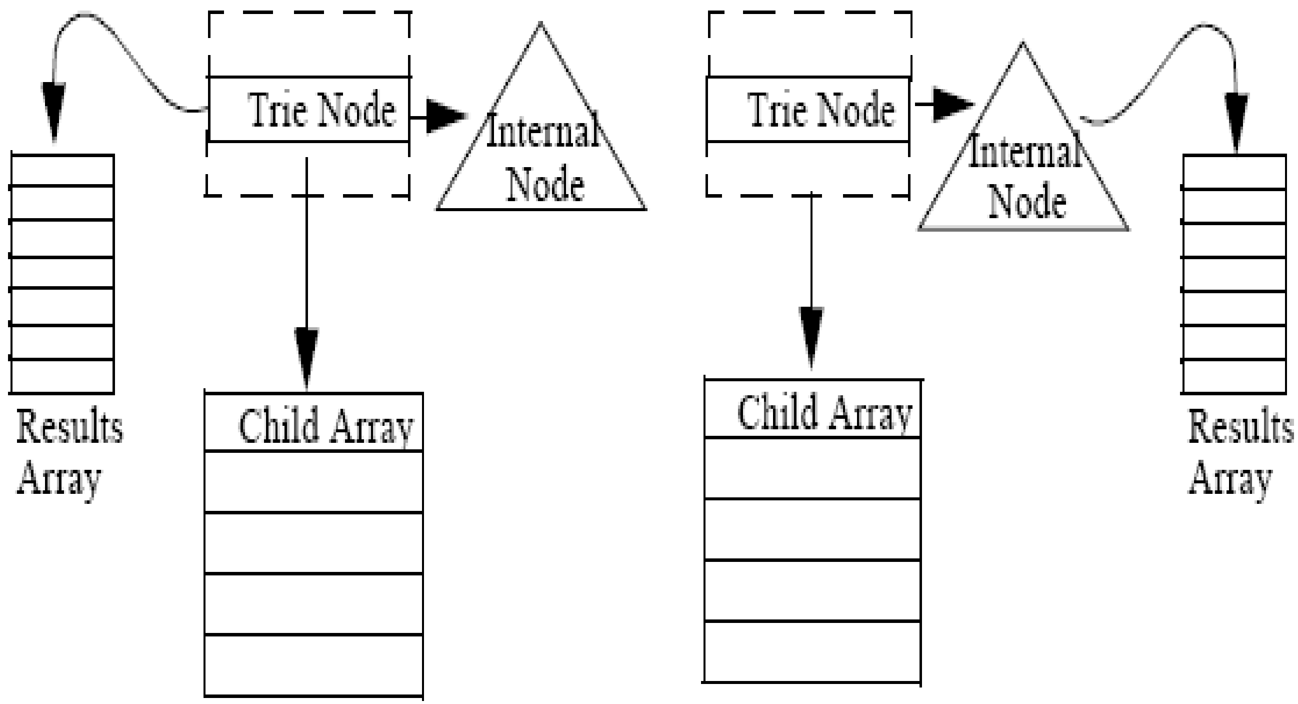


Figure 8: Split Tree Bitmap Optimization

Segmented Tree Bitmap

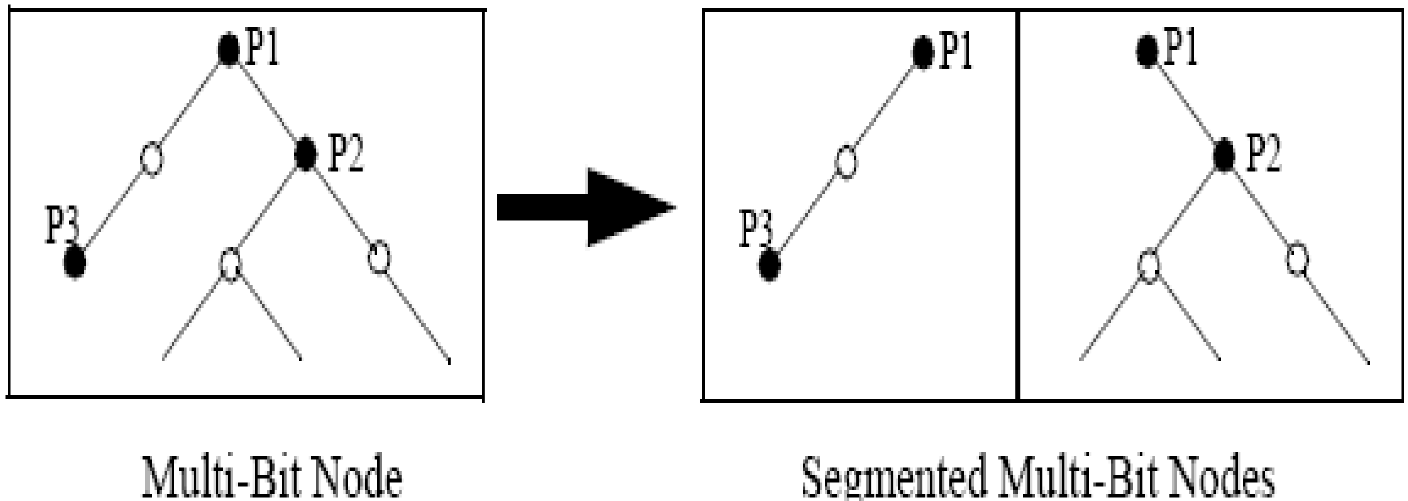


Figure 9: Segmented Tree Bitmap

IP Lookup Engine

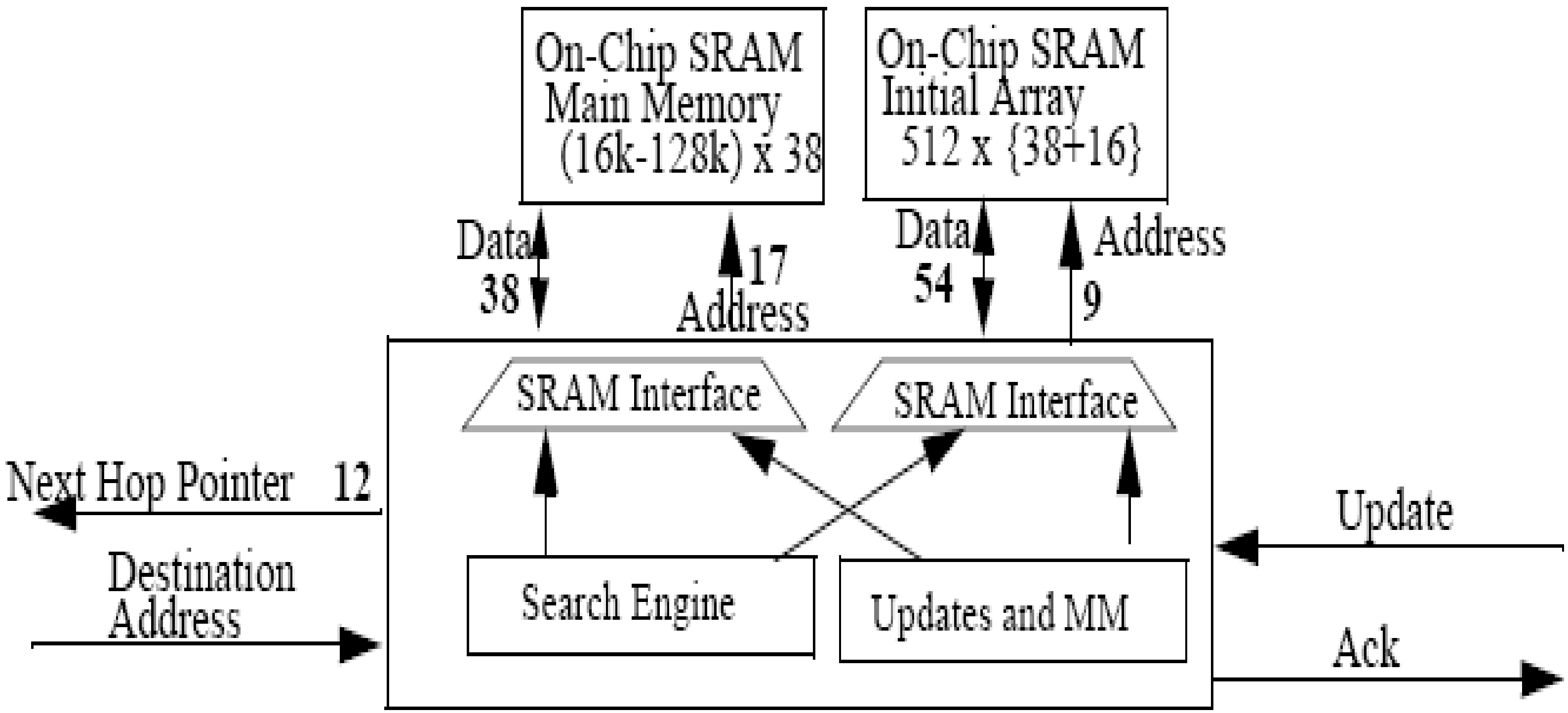


Figure 10: Block Diagram of IP Lookup Engine Core