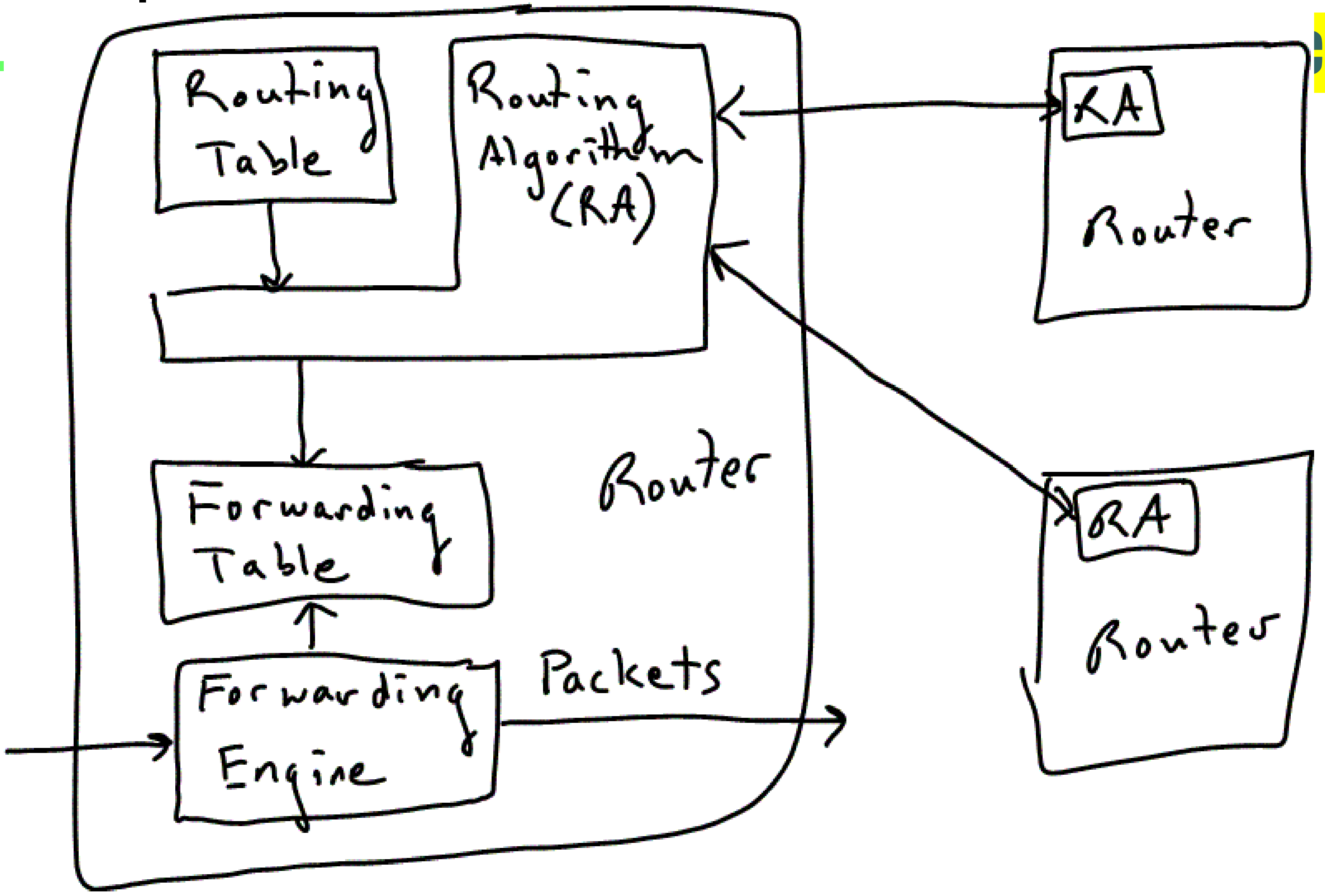
# CS419: Computer Networks

Lecture 6: March 7, 2005

*Fast Address Lookup:*

# Forwarding/Routing Revisited



# Best-match Longest-prefix forwarding table lookup

- We looked at the “semantics” of best-match longest-prefix address lookup
  - As a linear walk through the list of FIB entries, in order of longest-to-shortest prefix
- But we didn’t look at how to do this fast!

# Tree Bitmap

- This is a fast address lookup algorithm from George Varghese (UCSD)
- Used in high-speed routers (Cisco)
  - George has a startup doing this
- This lecture based on this paper:
  - W. Eatherton, Z. Dittia, G. Varghese, “Tree bitmap: hardware/software IP lookups with incremental updates,” ACM SIGCOMM CCR, Volume 34 , Issue 2 (April 2004)

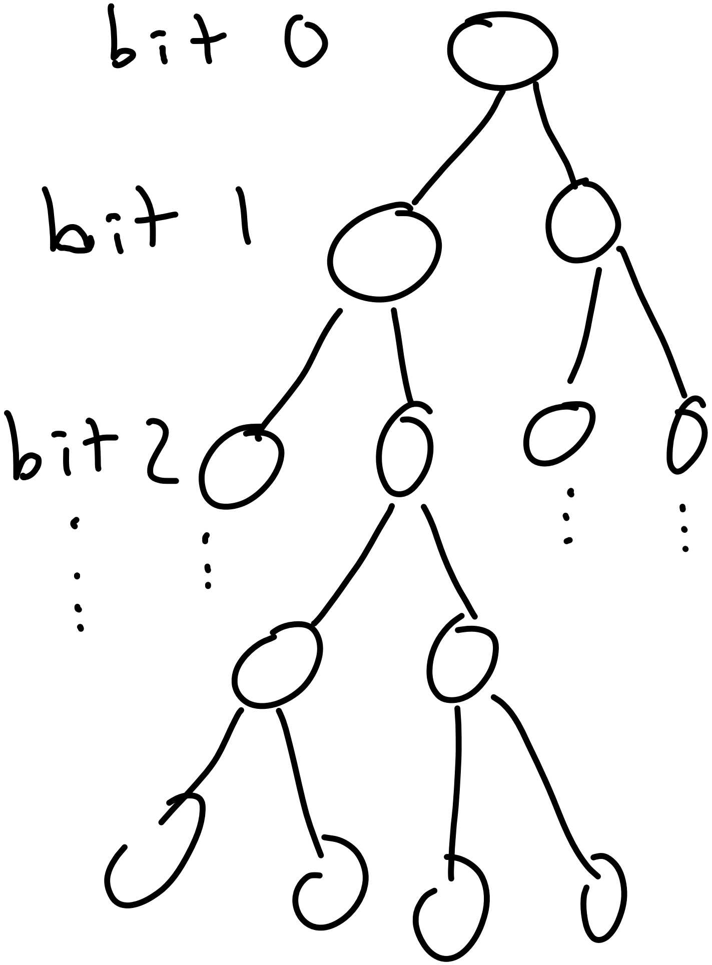
# Main goals:

- Wire-speed forwarding at OC-192 (10 Gbps)
- 24 million packets per second!!!
    - For small packets (TCP acks)
- Minimize memory accesses (4-7 for 41K FIB entries!)
- Performance guarantees
    - Not just for lookup, but for constructing the tree as well

# Other goals

- Operate in software and hardware modes
  - Variations on hardware:  single-chip, off-chip memory, CAMs
- Minimize memory size
- Take advantage of memory characteristics (i.e. cache line associated with a read)
- Tunable across many architectures

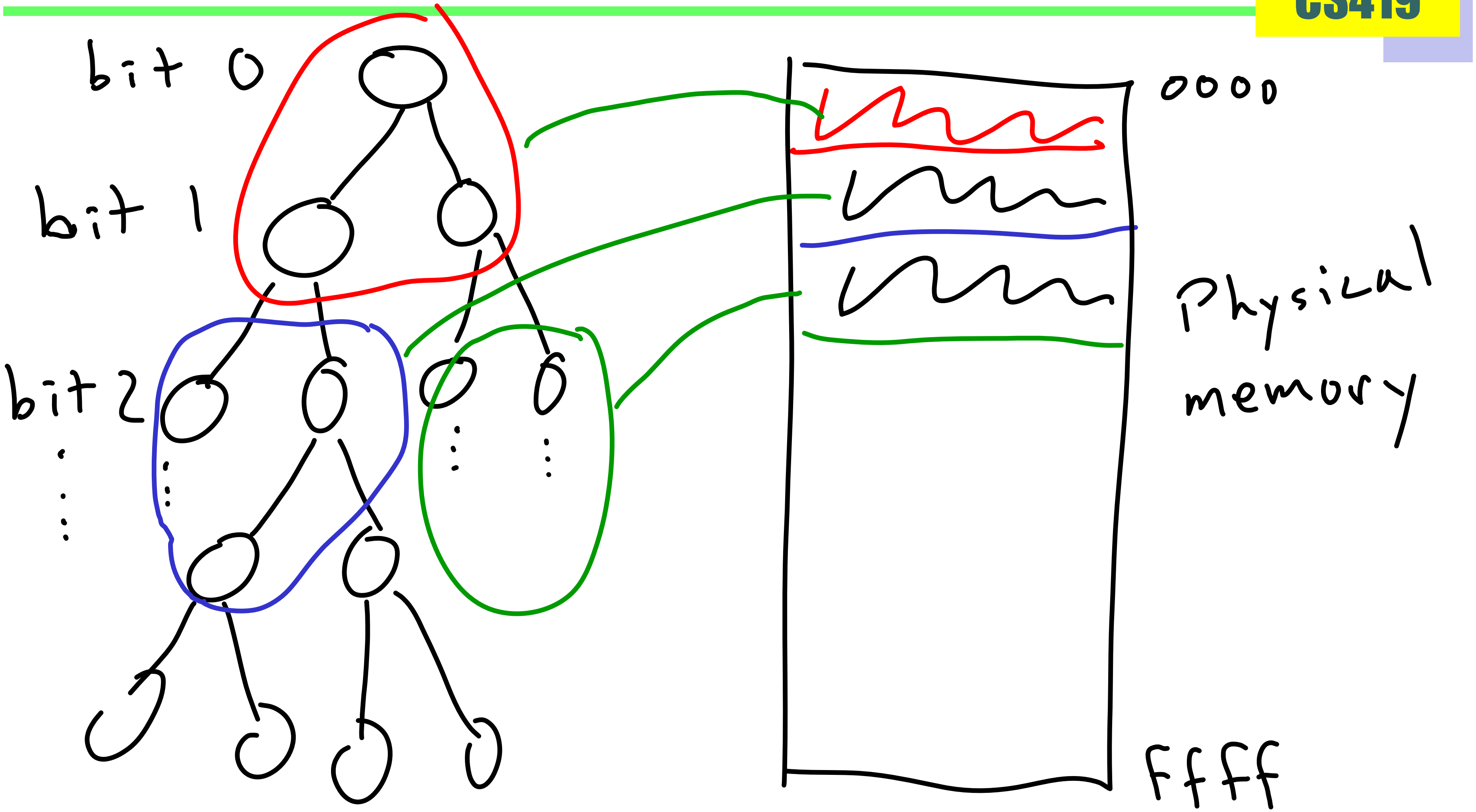
# Tuning to different memories



- All these algorithms involve traversing some kind of tree structure
- The trick to tuning is deciding where in physical memory to stick different parts of the tree…

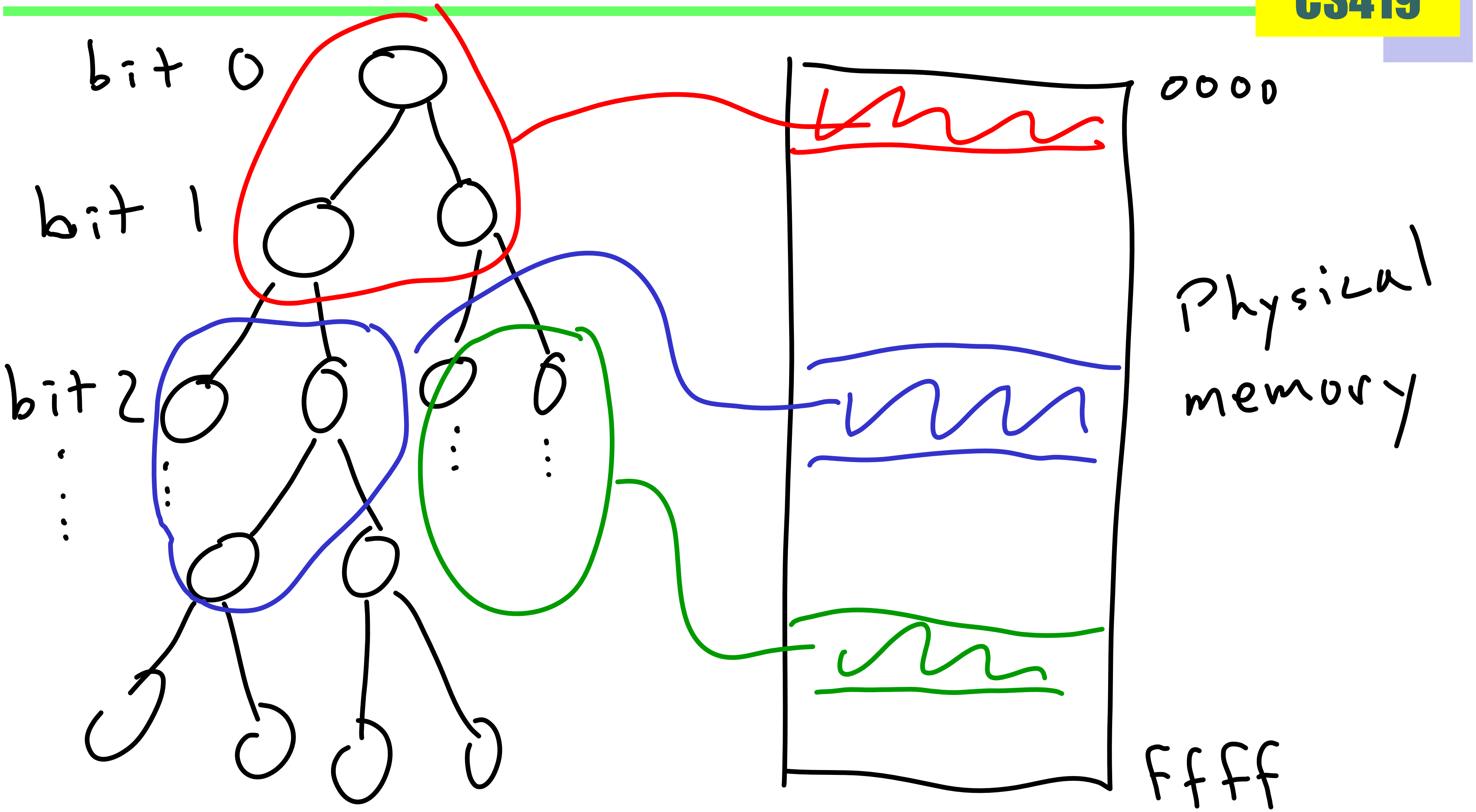
# Tuning to different memories

CS419

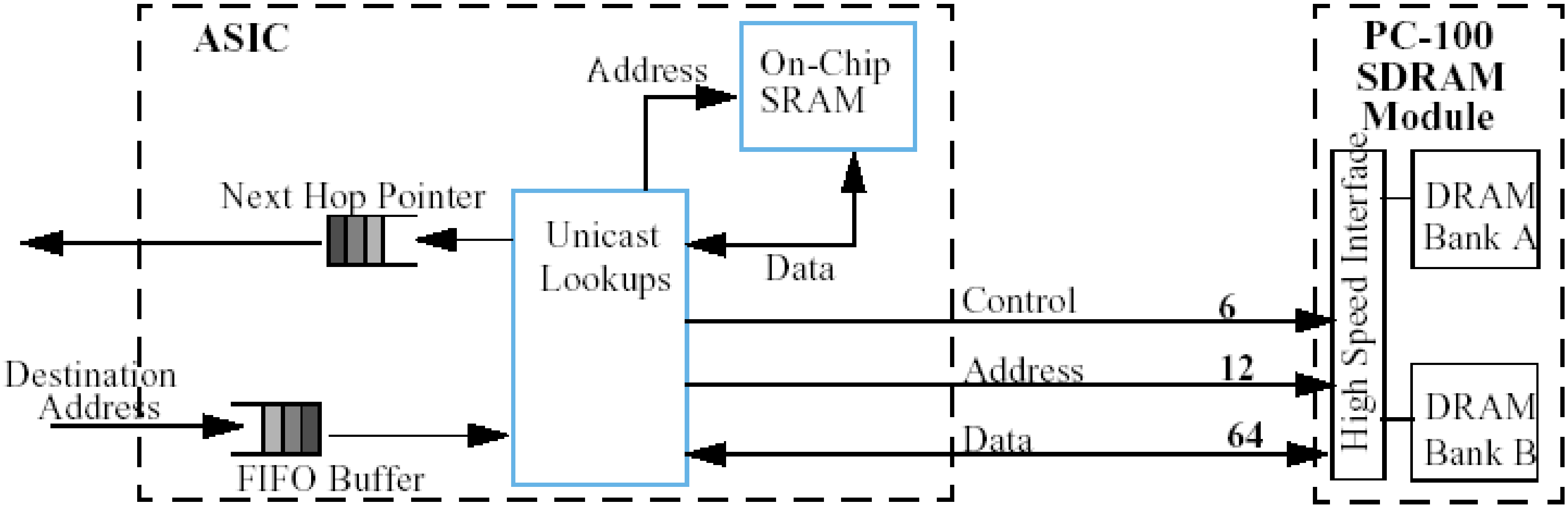


# Tuning to different memories

CS419

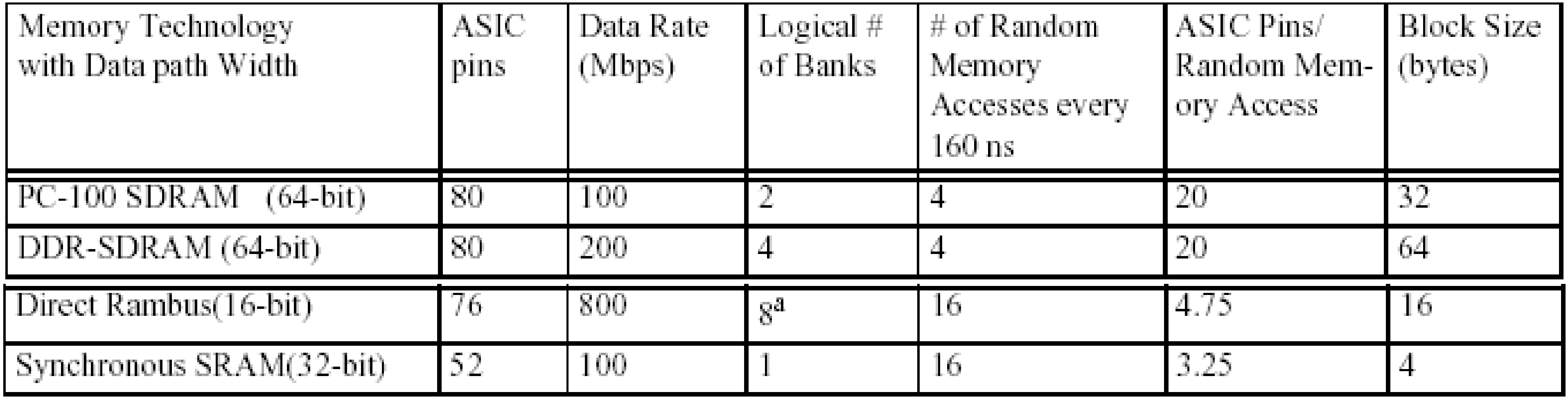


# Example: Multiple memory banks



- Put top of tree in Bank A, bottom tree in Bank B, run two lookups in parallel

# Example: Size of memory “burst”



| Memory Technology with Data path Width | ASIC pins | Data Rate (Mbps) | Logical # of Banks | # of Random Memory Accesses every 160 ns | ASIC Pins/ Random Memory Access | Block Size (bytes) |
|---|---|---|---|---|---|---|
| PC-100 SDRAM (64-bit) | 80 | 100 | 2 | 4 | 20 | 32 |
| DDR-SDRAM (64-bit) | 80 | 200 | 4 | 4 | 20 | 64 |
| Direct Rambus(16-bit) | 76 | 800 | 8[a] | 16 | 4.75 | 16 |
| Synchronous SRAM(32-bit) | 52 | 100 | 1 | 16 | 3.25 | 4 |

Various memory parameters determines the number of bytes that be read in one memory access. This in turn determines how to structure the lookup tree.

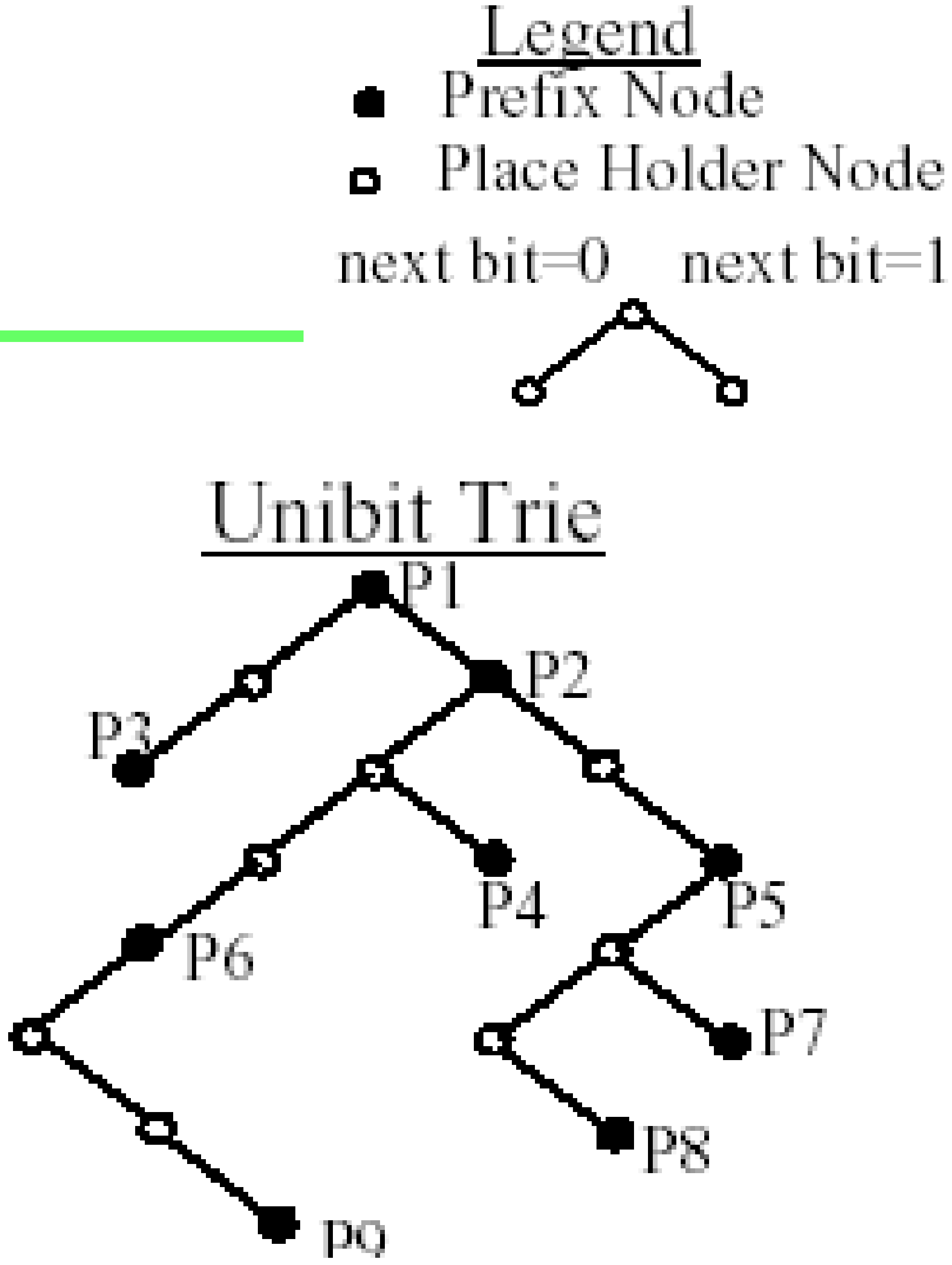
# Some types of trees

- Next we'll look at a number of tree structures, each more advanced (and harder to understand!) than the last
  - Unibit tries
  - Expanded tries
  - Lulea (bitmap)
  - Tree bitmap

# Unibit tries

## Prefix Database

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



# Unibit tries

- Traverse the tree one bit at a time
- If terminate at a prefix node, use that as the next hop
- If terminate at a “place holder” (non-prefix) node, use most recently traversed prefix node as the next hop
- One-way branches can be compressed out

# Unibit tries

- Small memory and update times
- Main problem is the number of memory accesses required
  - 32 in the worst case
  - Way beyond our budget of approx 4
    - (OC48 requires 160ns lookup, or 4 accesses)

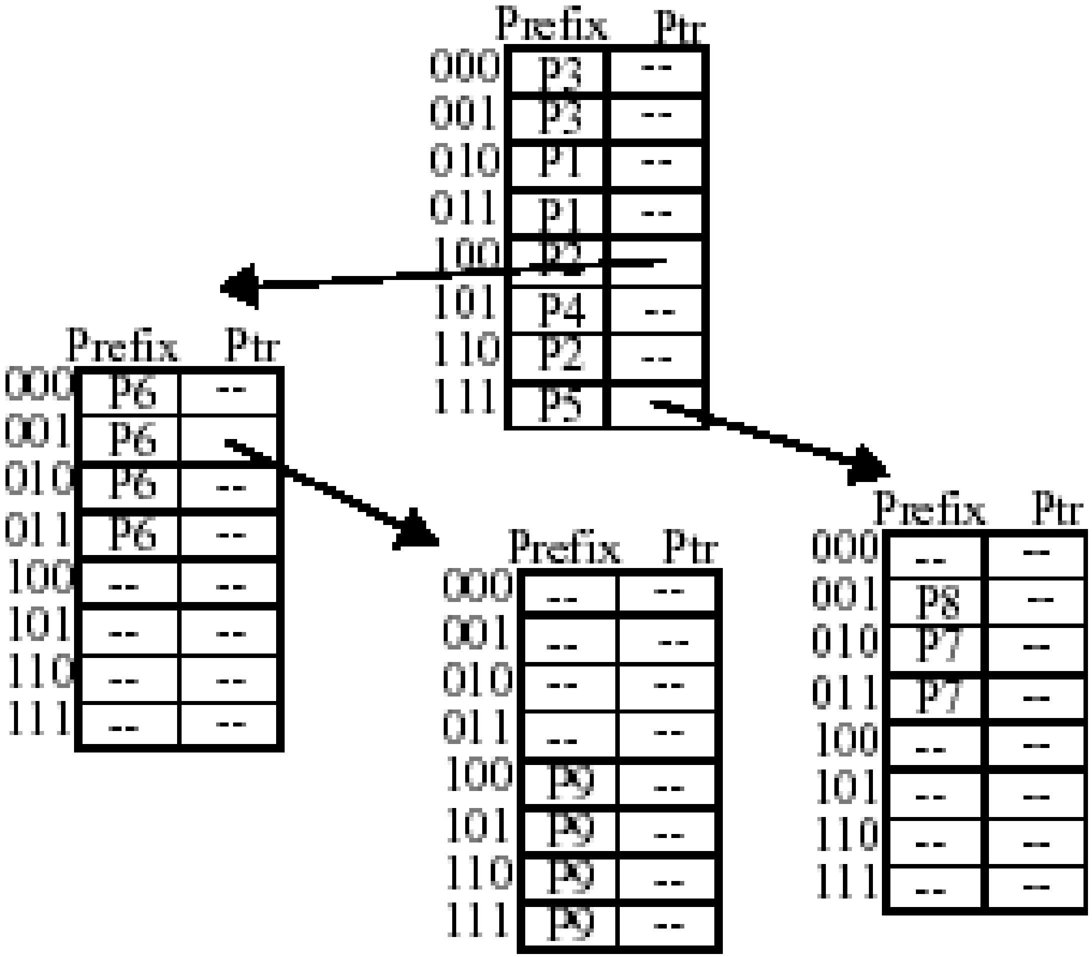
# Expanded tries

- To speed up lookup, branch on multiple bits at each decision instead of just one
  - The number of bits used is the “stride length”
- Otherwise, lookup algorithm similar to unibit
  - i.e. remember most recently traversed prefix in case of non-prefix termination

# Prefix expansion without leaf pushing

## Prefix Database

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



| | Prefix | Ptr |
|---|---|---|
| 000 | P3 | -- |
| 001 | P3 | -- |
| 010 | P1 | -- |
| 011 | P1 | -- |
| 100 | P2 | |
| 101 | P4 | -- |
| 110 | P2 | -- |
| 111 | P5 | |

| | Prefix | Ptr |
|---|---|---|
| 000 | P6 | -- |
| 001 | P6 | |
| 010 | P6 | -- |
| 011 | P6 | -- |
| 100 | -- | -- |
| 101 | -- | -- |
| 110 | -- | -- |
| 111 | -- | -- |

| | Prefix | Ptr |
|---|---|---|
| 000 | -- | -- |
| 001 | -- | -- |
| 010 | -- | -- |
| 011 | -- | -- |
| 100 | P9 | -- |
| 101 | P9 | -- |
| 110 | P9 | -- |
| 111 | P9 | -- |

| | Prefix | Ptr |
|---|---|---|
| 000 | -- | -- |
| 001 | P8 | -- |
| 010 | P7 | -- |
| 011 | P7 | -- |
| 100 | -- | -- |
| 101 | -- | -- |
| 110 | -- | -- |
| 111 | -- | -- |

# Expansion

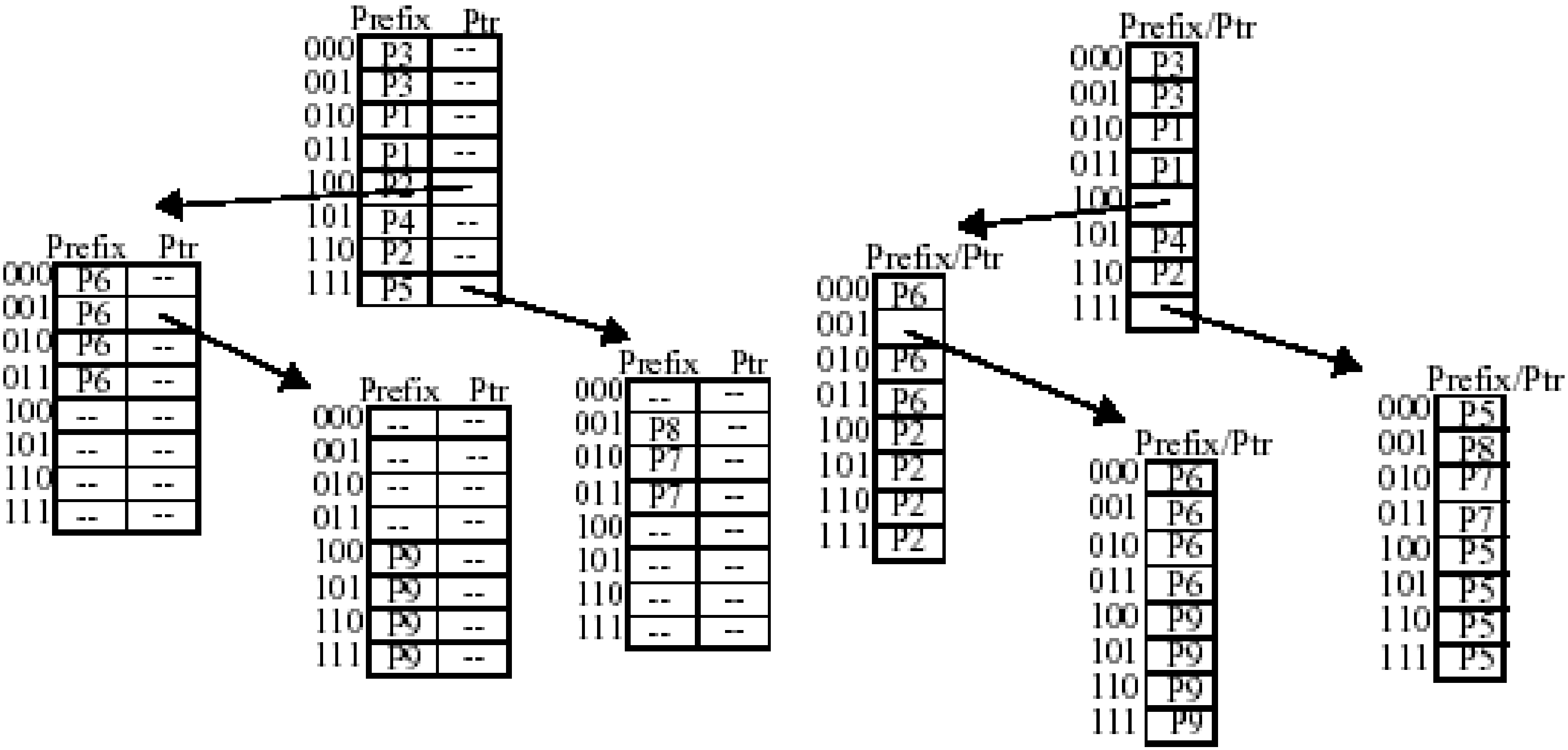
- Prefixes that don’t fall on stride boundaries must be “expanded” to fill all slots
- Eg P6 expanded to four slots
- Or, P2 expanded initially to four slots, but then P4 and P5 take precedence over P2

# Expanded trie inefficiencies

- Expansion uses up more space
- Also, each entry requires two fields
  - A pointer to the next node in the tree
  - A prefix
- This is because some entries require both a pointer and a prefix
  - i.e. P2, P5, and P6
- Update speed versus memory size tradeoff

# We can combine pointer and prefix…(leaf pushing)

CS419



| | Prefix | Ptr |
|---|---|---|
| 000 | P3 | -- |
| 001 | P3 | -- |
| 010 | P1 | -- |
| 011 | P1 | -- |
| 100 | P2 | |
| 101 | P4 | -- |
| 110 | P2 | -- |
| 111 | P5 | |

| | Prefix | Ptr |
|---|---|---|
| 000 | P6 | -- |
| 001 | P6 | |
| 010 | P6 | -- |
| 011 | P6 | -- |
| 100 | -- | -- |
| 101 | -- | -- |
| 110 | -- | -- |
| 111 | -- | -- |

| | Prefix | Ptr |
|---|---|---|
| 000 | -- | -- |
| 001 | -- | -- |
| 010 | -- | -- |
| 011 | -- | -- |
| 100 | P9 | -- |
| 101 | P9 | -- |
| 110 | P9 | -- |
| 111 | P9 | -- |

| | Prefix | Ptr |
|---|---|---|
| 000 | -- | -- |
| 001 | P8 | -- |
| 010 | P7 | -- |
| 011 | P7 | -- |
| 100 | -- | -- |
| 101 | -- | -- |
| 110 | -- | -- |
| 111 | -- | -- |

| | Prefix/Ptr |
|---|---|
| 000 | P3 |
| 001 | P3 |
| 010 | P1 |
| 011 | P1 |
| 100 | |
| 101 | P4 |
| 110 | P2 |
| 111 | |

| | Prefix/Ptr |
|---|---|
| 000 | P6 |
| 001 | |
| 010 | P6 |
| 011 | P6 |
| 100 | P2 |
| 101 | P2 |
| 110 | P2 |
| 111 | P2 |

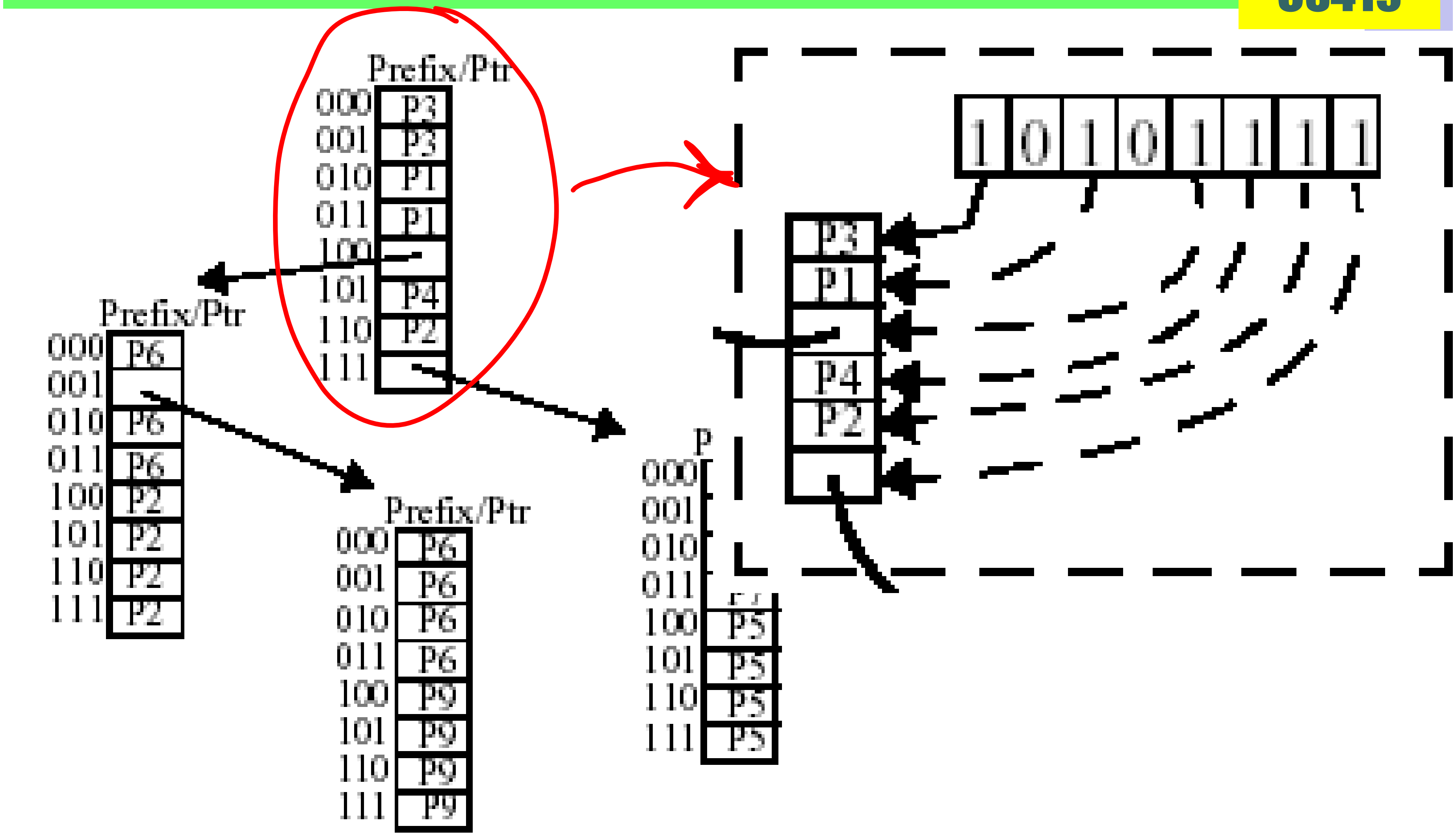
| | Prefix/Ptr |
|---|---|
| 000 | P6 |
| 001 | P6 |
| 010 | P6 |
| 011 | P6 |
| 100 | P9 |
| 101 | P9 |
| 110 | P9 |
| 111 | P9 |

| | Prefix/Ptr |
|---|---|
| 000 | P5 |
| 001 | P8 |
| 010 | P7 |
| 011 | P7 |
| 100 | P5 |
| 101 | P5 |
| 110 | P5 |
| 111 | P5 |

# Some observations

- Leaf pushing increases update time
  - Prefix can appear in many nodes (i.e. P5)
- Because of memory “burst” reads, the entire node can be read with one memory access
  - Try to make node size match burst size

# Lulea uses a bitmap to compress out repeated entries



| | Prefix/Ptr |
|---|---|
| 000 | P3 |
| 001 | P3 |
| 010 | P1 |
| 011 | P1 |
| 100 | |
| 101 | P4 |
| 110 | P2 |
| 111 | |

| | Prefix/Ptr |
|---|---|
| 000 | P6 |
| 001 | |
| 010 | P6 |
| 011 | P6 |
| 100 | P2 |
| 101 | P2 |
| 110 | P2 |
| 111 | P2 |

| | Prefix/Ptr |
|---|---|
| 000 | P6 |
| 001 | P6 |
| 010 | P6 |
| 011 | P6 |
| 100 | P9 |
| 101 | P9 |
| 110 | P9 |
| 111 | P9 |

| | P |
|---|---|
| 000 | |
| 001 | |
| 010 | |
| 011 | |
| 100 | P5 |
| 101 | P5 |
| 110 | P5 |
| 111 | P5 |

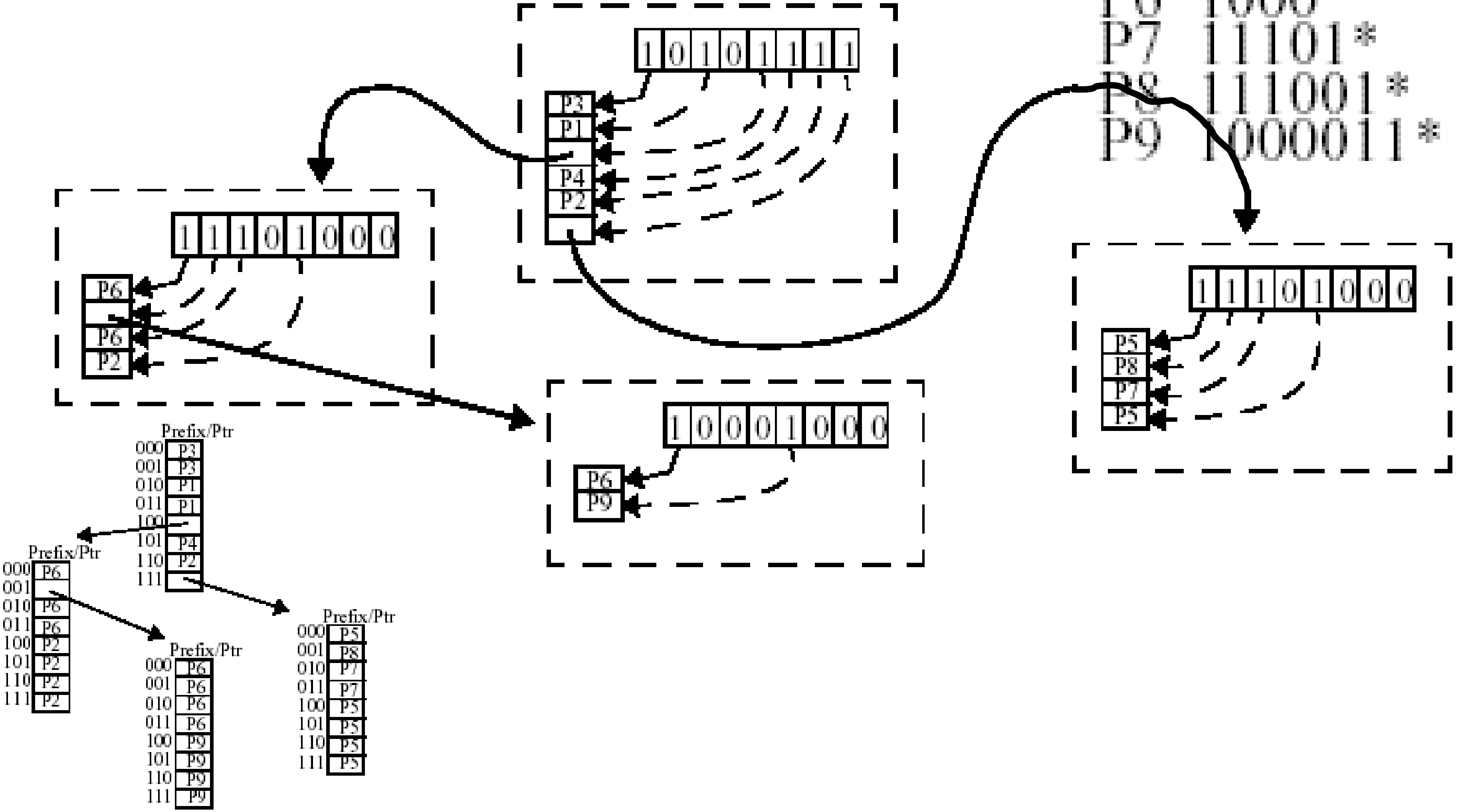
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

| |
|---|
| P3 |
| P1 |
| |
| P4 |
| P2 |
| |

# Lulea bitmap

## Prefix Database

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



| Prefix | Ptr |
|---|---|
| 000 | P3 |
| 001 | P3 |
| 010 | P1 |
| 011 | P1 |
| 100 | |
| 101 | P4 |
| 110 | P2 |
| 111 | |

| Prefix | Ptr |
|---|---|
| 000 | P6 |
| 001 | |
| 010 | P6 |
| 011 | P6 |
| 100 | P2 |
| 101 | P2 |
| 110 | P2 |
| 111 | P2 |

| Prefix | Ptr |
|---|---|
| 000 | P6 |
| 001 | P6 |
| 010 | P6 |
| 011 | P6 |
| 100 | P9 |
| 101 | P9 |
| 110 | P9 |
| 111 | P9 |

| Prefix | Ptr |
|---|---|
| 000 | P5 |
| 001 | P8 |
| 010 | P7 |
| 011 | P7 |
| 100 | P5 |
| 101 | P5 |
| 110 | P5 |
| 111 | P5 |

# Lulea bitmap processing

- Doesn’t this just increase processing?
  - Have to shift through the bitmap…
- Yes, but memory access is by far the bottleneck
  - Hardware easily process the bitmaps
  - Even software can execute many instructions in one memory access

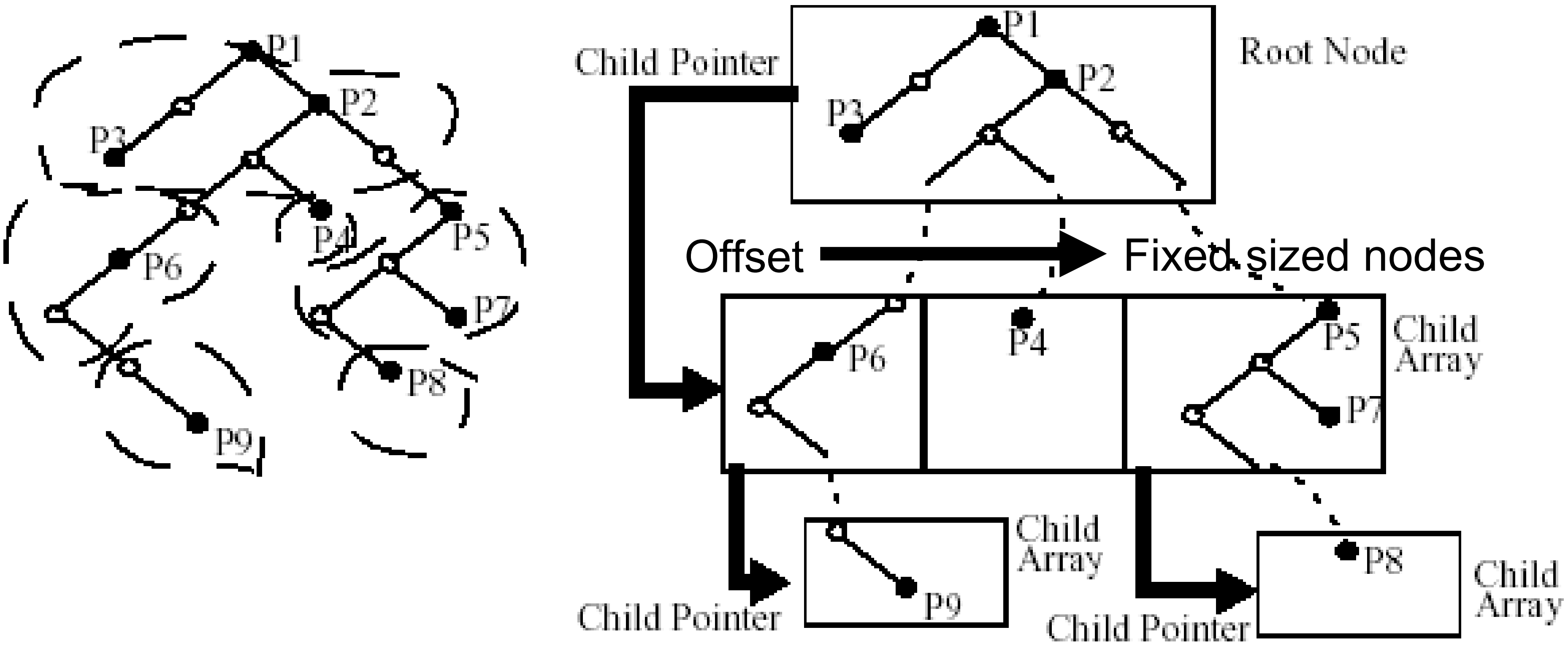
# Lulea trie performance

- Very compact storage
- Very fast lookup
- But, updating the Lulea trie can be very expensive
- For instance, adding a short prefix can result in a lot of leaf pushing…many entries must be modified

# Tree Bitmap: first insight

- Avoid the problems of expansion and leaf pushing by going back (conceptually) to the basic Unibit tree
- BUT:  Avoid the problem of many pointers by storing child nodes in contiguous memory areas as an array
  - Instead of many pointers, calculate offset into child array

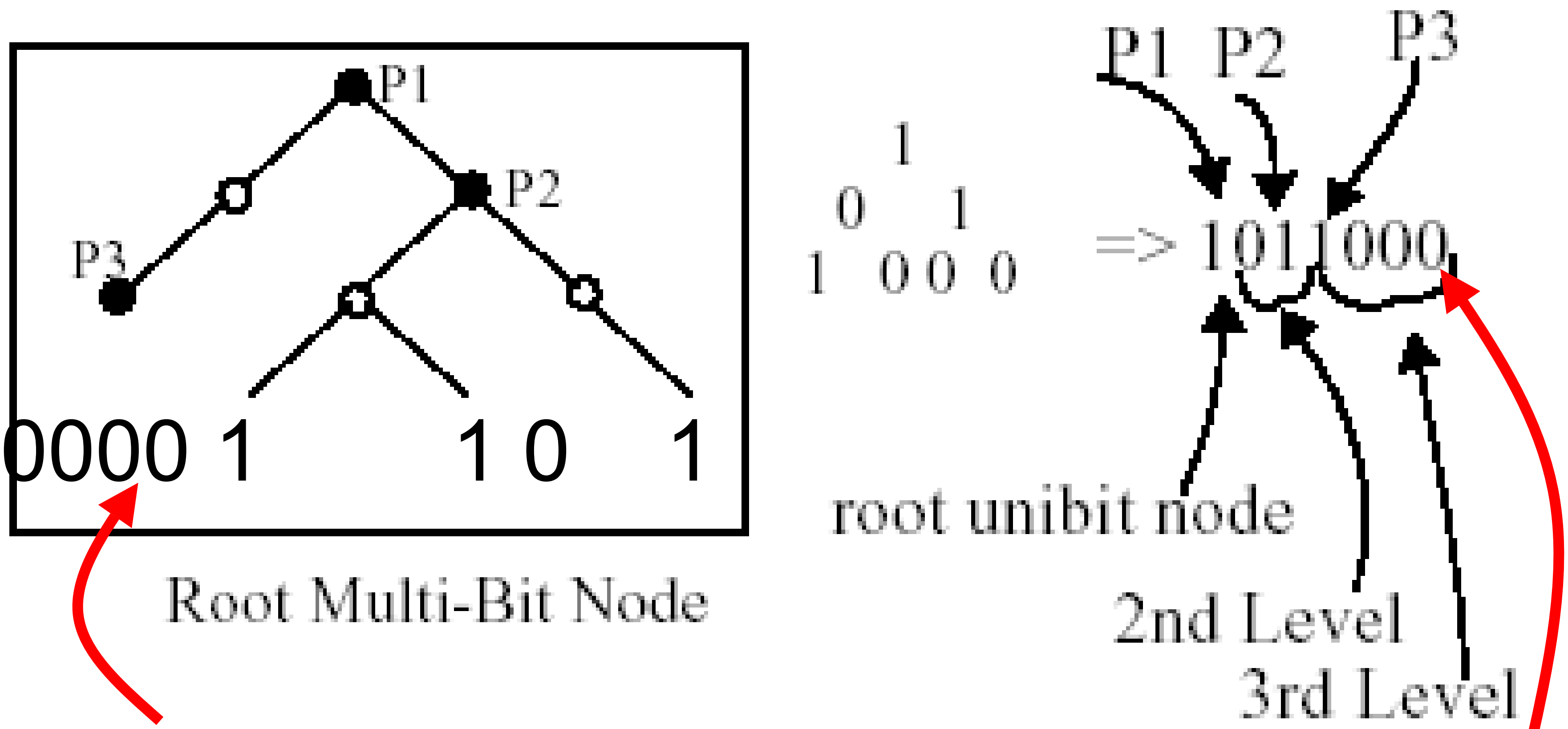
# Tree Bitmap with three-bit strides



# Tree Bitmap: second insight

- To compress, use two bitmaps instead of just one
  - Internal prefix bitmap
  - External pointers bitmap
- This avoids leaf pushing
  - (which is what gives Lulea potential large update times)

# Tree Bitmap's two bitmaps



Extended Paths Bitmap

Internal Tree Bitmap

# Compact “nodes”

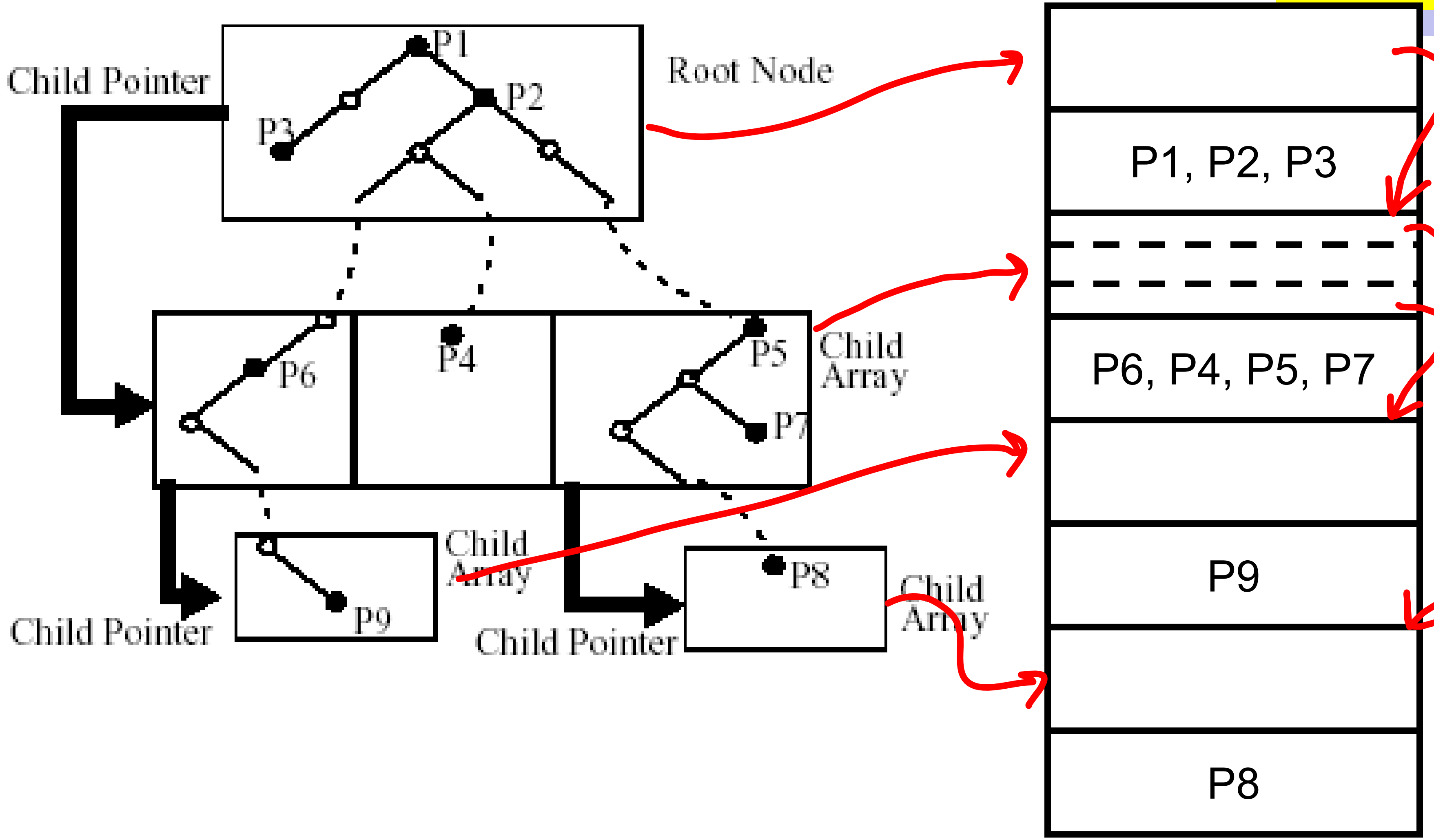
- Each child node contains only:
  - Internal Tree Bitmap
  - Extended Paths Bitmap
  - One pointer to child array
- But what about the next hop info for stored prefixes???
  - This is what was pushed to the leaves in Lulea…

# Stored next hop info for prefixes

- Store prefixes in a separate array *adjacent in memory* to the node
- Internal tree bitmap tells us where in that array to find the pointer
- Furthermore, don’t actually retrieve the next hop info until the very end of the search
    - Adds one extra memory access at the very end

# Next hop pointer array in adjacent memory location

CS419



# Lookup algorithm (basic idea anyway)

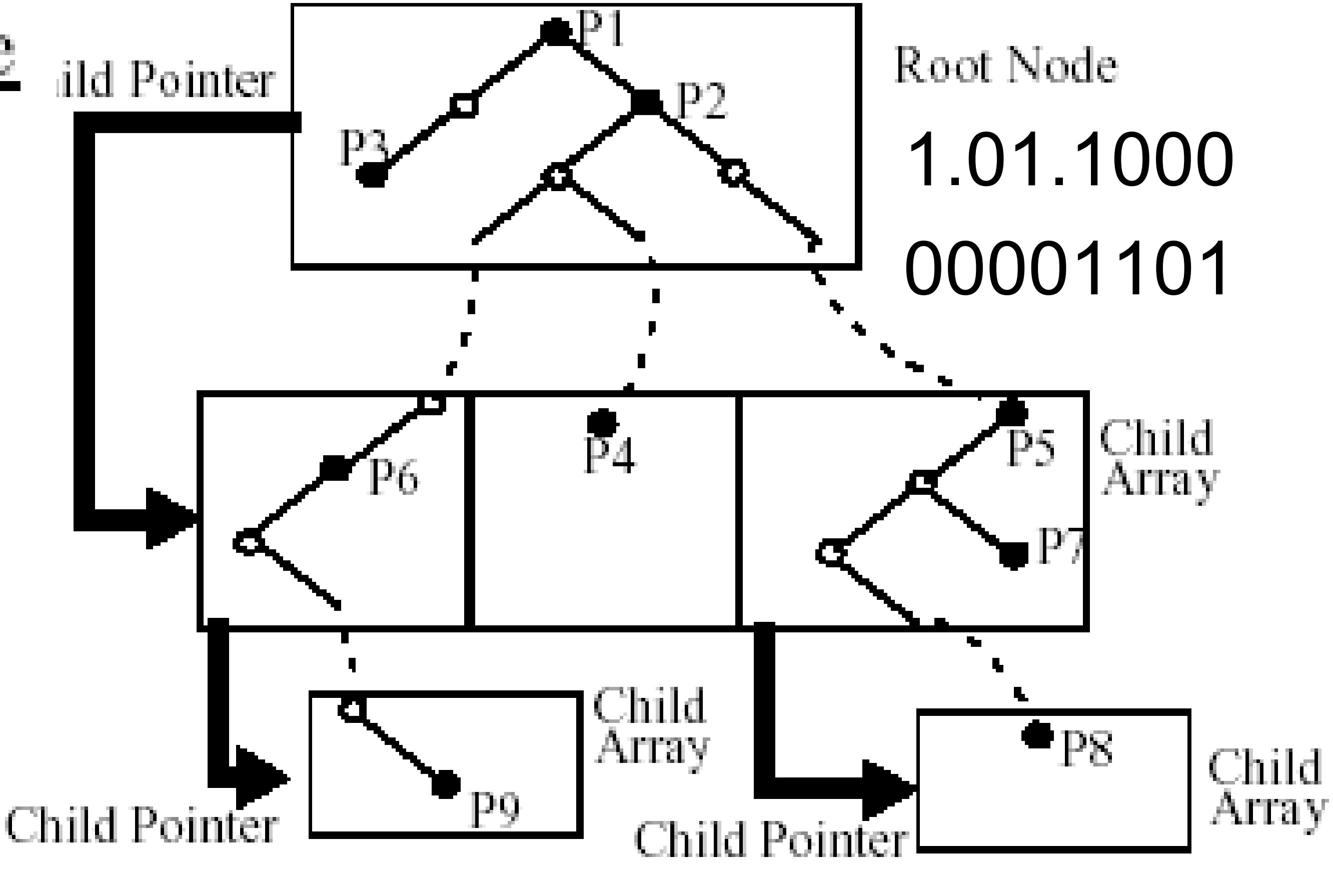
- Conceptually, the two bitmaps allow you to “reconstruct” the Unibit tree for a given stride (i.e. 3 bits)
- The child pointer plus Extended Paths Bitmap tell you where to find the child node
- The Internal Tree Bitmap tells you which Unibit tree nodes have prefixes

# Lookup algorithm (basic idea anyway)

CS419

Prefix Database

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



1.01.1000

00001101