Packet classification
• The next 2 slides are slides 19&20 of https://www.mef.net/Assets/Technical_Specifications/PPT/Overview-of-MEF_23_Phase_II-Mar-12-2012.ppt

• Web PPT version has better colouring – lost when slide imported into this presentation
Bandwidth Profile is a characterization of Ethernet frames – e.g., frames from a customer into a UNI.

Bandwidth Profile attributes:
- Frame delivery obligated by the SLA
  - Committed Information Rate (CIR) [bits per sec]
  - Committed Burst Size (CBS) [bytes]
- Frame delivery based on available bandwidth (not subject to SLA)
  - Excess Information Rate (EIR) [bits per sec]
  - Excess Burst Size (EBS) [bytes]
Bandwidth Profile Model

Not really accurate in that CIR is fill rate and token usage is the drain

User

Ingress Traffic

User-Network Interface

Network

Committed Burst Size (CBS)

Overflow

Excess Burst Size (EBS)

Excess Information Rate (EIR)

SLA Guaranteed

Best Effort only

- Note: frames are transmitted immediately if network is uncongested

This slide is not really accurate. See Slides 40-41 for more accurate diagrams. Recommend substituting at least some form of slide 40 for this one.
Context

Following weeks:
Now: “Packet” classification @ input port processors
Then:
1. Buffering
   • Forwarding modes across switch
   • Discard strategies at points of contention, especially output ports
2. Scheduling
   • Scheduling at points of contention, especially output ports
   • Traffic conditioning
   • Traffic management
Resources: textbooks

Assumed background on IP addressing: slides and videos (prefixes, longest prefix matching & prefix aggregation)

Varghese did research in this area => strong coverage, but perhaps too detailed
• Chapter 10: Exact match lookups
• Chapter 11: Prefix match lookups
• Chapter 12: “Packet classification” (matching arbitrary subsets of key bits)

Relevant sections of Keshav:
• Chapter 10: Addressing
• pp. 176-8: classification (“port mappers”)
Resources

D. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comp. Surv.*, vol. 37, pp. 238-75

M. Ruiz-Sanchez and others: “Survey and taxonomy of IP address lookup algorithms”, *IEEE Network*, 15(2):8-23


L. Chisvin and J. J. Duckworth: “Content-Addressable and Associative Memory: Alternatives to the ubiquitous RAM”, *Computer* 22(7):51-64


Cisco Nexus context

“a multistage policy engine that is responsible for manipulating the forwarding results with a combination of parallel searches in memory arrays, hash tables, and ternary content-addressable memory (TCAM).”

-the appropriate forwarding table is looked up for the forwarding decision. For Ethernet, the 32,000-entry station table is used”

“This latency was measured on fully configured interfaces, with access control lists (ACLs), quality of service (QoS), and all other data path features turned on.”

-i.e. incorporating delays from sophisticated packet classification

“Traffic Classification … can be based on CoS or Differentiated Services Code Point (DSCP) bits of the incoming packet or on user-defined QoS ACLs that match Layer 2, 3, and 4 information. ” i.e. non-address fields.

“The lookup and ACL [access control list] receive the extracted packet fields, synthesize the lookup keys, and search a series of data structures”

“Multipath expansion [like inverse multiplexing]… The Cisco Nexus 5500 platform uses multiple polynomial functions to hash a flow to obtain a numerical value that can be used to choose among up to 16 physical interfaces.”
Classification in SDN switches

Packet classification is central to Openflow, which is central to Software Defined Networks: Flows are defined in terms of the fields used to classify packets belonging to a flow. Switch hardware focuses on fast classification.

e.g. Noviflow switches [http://noviflow.com/products/noviswitch/]

- “Optimized TCAM memory supports 125K to 1 million entries in wild card match flow tables, with up to 60 wild card match flow tables
- Supports up to 3 million entries in exact match flow tables (32 bytes wide), with up to 60 exact match flow tables
- Up to 14,000 flow-mods/second”
Where are packet classifiers used?

- Switches/routers
- Load balancers
- Address translators
- Traffic regulators (shapers, policers, etc) [FM>
- Monitoring
  - Metering
  - Provisioning depends on monitoring trends
  - Anomaly detection → device failures or security attacks
- Firewalls
- Caches [19W>
Outline

The classification problem
Classification approaches
Exact match classification
Partial match classification
Synopsis  
(of the classification problem)

To classify a packet, we  
1. **Extract at least one**† key \( (K) \) from packet (header)  
2. **Query a database** of \( D \) rules to determine which best applies to the key  
3. **Read the entry** (described by \( E \) bits) associated with the rule, **act** according to entry  

e.g. we might  
1. Use the destination address as a key  
2. Have a database that records known destinations and which port leads to them.  
3. Send the packet to the appropriate output port  

† May need multiple lookups [1HN> e.g. Ethernet switch looks up destination and source
Outline

- Entries ($E$ bits of data) associated with a rule
  - applications
- Keys ($K$)
  - Address fields
  - Other header fields
  - Characteristics of keys
- Rules
  - Types: Exact & partial match with(out) prioritisation
  - Number ($D$)
- Evaluating classifiers
Packet classification: applications

When “packets”† arrive at “switch”‡ inputs, need to use packet fields to look up ‘E’ bits of state information:

- **Which output port?**
  - Switch input to the correct output. “Correct”:
    - port leads to destination
    - port provides appropriate service, e.g. voice traffic over terrestrial links & file transfer can go over satellite
  - Filtering (no output port) (e.g. see [XD>]): Prevent flow of traffic
    - From certain addresses, e.g. from users who haven’t paid
    - To certain addresses/services, e.g. firewall

- **What class of service** (within the switch)? [“appropriate service” again]
- **Record usage** (billing, conditioning, etc)

If packet can’t be classified, then switch may take some default action, e.g. discard it, send to a default gateway, flood out in all directions, etc.

† or frames, or segments
‡ or bridge or router
Keys†: Packet address fields

Packet header fields that may be used for classification:

- Source and destination **address fields**
  - Destination address is the most important
  - But source address may be relevant:
    - for filtering
    - when recording usage
    - for “load balancing” (traffic from multiple sources to one destination follow different paths to spread load. Prefer all traffic from one source follow the same path to preserve sequence)
  - for Ethernet switches – learn station locations by observing source addresses
- Link layer: address fields are 48b
- Network layer: address fields are 32b (IPv4) or 128b (IPv6)
  - continued on next slide...

† We use the term “key” to differentiate between addresses in packets and addresses of memories used for classification. Also, the packet address may only be part of the key.
Keys: Other (non-address) header fields

Packet header fields that may be used for classification:

- **Source and destination address fields**
- Virtual Circuit Identifiers: Small in order to simplify classification e.g. 16b for ATM (8 or 12b VPI), 12b VLAN ID – may be nested
- Fields indicating required **Class of Service <PR>:**
  - 8b: Type of Service/Diffserv (IPv4), Traffic Class (IPv6)
  - IPv6: 20b Flow Label
  - 8b: Protocol (IPv4), Next Header (IPv6), e.g. TCP, UDP or ICMP
  - 16b: Transport: source and destination “ports” (not switch ports)
    - e.g. low delay and throughput for VOIP (UDP “port” 5004),
      high delay and high throughput for Dropbox file transfer
- **“Deep Packet Inspection (DPI)”** may classify based on “payload”,
  e.g. does an email contain certain key words?

=> **key width K.** Classification is simple for small $K$ (e.g. $\leq 32b$), harder for larger $K$ (e.g. 48b+)
Characteristics of keys

(Apart from their size, see previous slide <14A>) Recall from week 2:

Values may be correlated
   e.g. LAN with many NICs bought in bulk from one vendor: IEEE link
   layer addresses have same 24 OUI bits (one vendor), and similar
   (perhaps sequential) less-significant bits
   => deal with hash collisions

Timing issues:
   Temporal locality: Keys that have been queried recently will likely be
   queried again soon (e.g. burst of packets <VY> from a source to a
   destination, or subsequent reply) => caching (tries [HH>, small
   CAMs)
   (Varghese argues against this.)
   Some keys will be queried more often than others, e.g. a server
   carrying a high load => optimize for the common case

OUI = Organizationally Unique Identifier

Copyright © 2017 Tim Moors
Rules: Classification variations

- **Exact matching**: All rules care about the same key digits.
  - Key will match at most one rule.
  - e.g. Ethernet switch
  Rare exception: May have multiple identical “rules” (e.g. free CAM words).

- **Partial matching**: Rules may care about different key digits.
  - Key may match multiple rules. Need to prioritise rules to determine “best” match.
  - Label individual “don’t care” bits with X; series with * character‡

- **Prefix matching**: Don’t care bits are at “end” of key. Are the “don’t care” bits indicated by:
  - keys (e.g. classful IP addresses)?: exact match for multiple possible prefixes
  - rules only (e.g. classless IP): longest prefix matching in routers

- **Arbitrary matching†**: Each rule may (not) care about arbitrary sets of bits
  - e.g. firewall rules

- **Range checking**: Does key lie within some specified numeric range? e.g. firewall checking of UDP/TCP “port” number

† Varghese calls this alone “packet classification”.
‡ * is like the wildcard used in filesystems, and is not to be confused with regular expressions where * indicates any number of the previous character.
Prioritisation of rules

Classification rules may intersect => need to prioritize, e.g.:

- **Router**: Longest-prefix matching: Most specific rule is more appropriate – highest priority.

- **Firewall**: Filtering traffic may have priority over sending certain traffic on matching path, e.g.:
  
  Rule 1: Block traffic from network $N$
  
  Rule 2: Forward traffic to network $M$ through port $P$
  
  Packet from $N$ to $M$ matches both rules, but Rule 1 may have priority => block packet.
One lookup or multiple?

- **In theory**, layers are processed separately, so one packet may need multiple database lookups, for multiple layers.
- **In practice**, a device might combine fields from multiple protocols to create one key, e.g. a switch/router:
  - If Eth.dst is known, Don’t care about IP, switch frame
  - If Eth.dst==mine & ip.dst==known, route packet

- **Combining fields might lead to massive tables:**
  - e.g. Ethernet switch: lookup source to learn, lookup dest to forward; 2 lookups or 1 in table with all source*dest combos.
  - Known as the “table entry explosion” problem
  - OpenFlow v1.3 allows “Multiple Flow Tables” while OFv1.0 only allows one.
Rules: How many?

Databases tend to be sparse (few entries, $D$, relative to keyspace $2^K$)

- Ethernet switch $D=1000$ ($1000/2^{48} = 1$ in 300 billion)
- Backbone router: $D=100k^\dagger$ ($41,578/2^{32} = 1$ in 100,000 $^\dagger$)

<table>
<thead>
<tr>
<th>Site</th>
<th>Entries ($D$) $^\dagger$</th>
<th>Next hops ($\leq 2^E$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mae East</td>
<td>39,819</td>
<td>56</td>
</tr>
<tr>
<td>Mae West</td>
<td>14,618</td>
<td>55</td>
</tr>
<tr>
<td>AADS</td>
<td>20,299</td>
<td>19</td>
</tr>
<tr>
<td>Pac Bell</td>
<td>20,611</td>
<td>3</td>
</tr>
<tr>
<td>FUNET</td>
<td>41,578</td>
<td>20</td>
</tr>
</tbody>
</table>

$^\dagger$ Router entries represent networks = multiple host addresses => the density is not as sparse as claimed here

e.g. 56 possible next hops (output ports) => store in $E=6$ bits

[Nilsson and Karlsson, 98]
Growth of database sizes

Figure from http://bgp.potaroo.net/
Classification problem: Parameters

QOS router example

Use $K$ bit key

$$K=48 \ (32b \ \text{IP DA} + 16b \ \text{server UDP/TCP “port” #})$$

to select one of $D$ entries in database

$$D=10k \ (\text{known addresses})$$

read out $E$ bits of data from entry

$$E=4 \ (2^4=16 \ \text{interfaces on router})$$

Servers use well-known “port” numbers that have lower values than the ephemeral “port” numbers used by clients. Router can inspect source & destination “port” #s and use the smaller as the server “port” #.
Evaluating classifiers

Performance figures:

- **Query time:** Time to *search* the database.
  - e.g. 64B Ethernet frames @ 1Gb/s = 0.5us
  - Shorter is better
  - Want result in a predictable time, e.g. process frames “at wire speed” for worst-case databases
    - Algorithms requiring on average few, but in worst case many, steps may be unsuitable if packet must be classified before next one arrives.
  - Throughput is more important than delay => pipelining
    - e.g. tries [HH> may require multiple consecutive decisions, but these can be concurrently applied to different packets

- **Update time:** Time to *update* the database.
  - Updates don’t need to be as fast as queries (analogous to optical switches being slow to reconfigure but providing fast throughput <6X])
  - Faster is better, but slow may be OK (e.g. in a firewall)
  - May need 2 copies of database: One in use; other being updated

Cost (e.g. RAM size) for given capacity \((K, D, E)\)
Outline

Classification approaches

Analogy: Student classification
Student classification: problem

How can a lecturer rapidly look up student data of $E$ bits?

* e.g. to record result after marking test ($E=3_{10}$)

Each student is identified by a 7 digit student number ($K=7_{10}$)

* $10^7$ possible numbers
* only 78 students in class ($D=78$)
* numbers of students in class are correlated, since they are likely to have enrolled at similar times and numbers are allocated sequentially over time (e.g. many beginning 310...)
Manually classifying students

How might a lecturer do it?

1. Sort list of student IDs into numerical order
   - Sacrifices ease of update in order to expedite search

2. Search through list to find match
   - Sequential search is slow
   - Binary search is possible
   - In practice, initial searches may be binary, until lecturer gets a “feel” for the space (vague memory of experience), then initial guess of location can be more accurate (e.g. 222* => look at top of list)

3. Parallel search: Associative memory (brain) which stores data.
Automatically classifying students

A1: Manual approach: List them in order &:
  • search sequentially
  • binary search

A2: Use student number to index a table in RAM.
  Wastes RAM since $78 \ll 10^K = 10^7$

A3: Convert student number into smaller number to index a table, e.g. choose least significant 3 digits. More efficient with RAM but may have collisions (Hashing)

A4: Special associative memory that can simultaneously match all stored words with supplied value. (CAMs)

A5: Other data structures (linked lists, tries [HH>, etc)
Outline

Exact match classification
  Lookup tables
  Linked lists
  Hashing
  Content Addressable Memories (CAMs)
Naïve solution: Single lookup table

Use the key to index a RAM of $2^K$ words of $E$ bits each

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>3104005</td>
<td>42</td>
</tr>
<tr>
<td>3104004</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td>-1</td>
</tr>
<tr>
<td>3104883</td>
<td>99</td>
</tr>
<tr>
<td>3104884</td>
<td>-1</td>
</tr>
<tr>
<td>3104885</td>
<td>100</td>
</tr>
<tr>
<td>3104886</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td>-1</td>
</tr>
<tr>
<td>3105050</td>
<td>65</td>
</tr>
</tbody>
</table>

e.g. data indicates if student is not member of class (-1), or exam result (if non-negative) for students in class: 3104005, 3104883, 3104885, 3105050

√ Advantage: Simple & fast (one lookup of simple memory).
× Disadvantage: Sparsely populated large RAM ($D<<2^K$)
OK when key is small => virtual circuits & label swapping
Feasibility of a single lookup table

IPv4: 32b destination address lookup =>
Size of RAM is feasible with classfull addresses:
  Class A: 7b network number = 128 word memory
  Class B: 14b network number = 16K word memory
  Class C: 21b network number = 2M word memory
Size becomes infeasible with larger keys (e.g. IPv4 with fields other than DA, e.g. SA and class of service, 128b IPv6 DAs, 48b link layer DAs, etc)
List of data in RAM

√ Saves space: \( D \) words each storing \( K + E \) bits

× Takes time

   × If data is sorted (as shown)
     × Update: To insert/remove an item: average time \( D/2 \)
     √ Query: To perform a match using a Binary search:
       average time \( \log_2(D) \)

× If data is unsorted:

   × Query: To perform a match: average time \( D/2 \)
   × Removal update: After removing an item from the list either:
     × invalidate entry (matching must traverse invalid entries)
     × shift entries to fill space (e.g. fill with highest used address)

√ Insertion update: Easy to add an item to the list
Linked lists

- Efficient use of space: $D$ words each storing $K+E+\log_2 D$ bits
- Timing:
  - Easy to update the list
  - To query:
    - Average time $D/2$
    - But: Easy to relocate item to head of list (remove+add), and expedite future matching if keys exhibit temporal locality
    - Worst case time: $D$

Linked lists appear elsewhere in this course in the context of TSI switching fabrics [RZ] and packet buffers [AR].
Hashing

Definition: Mathematical mapping from large value to smaller one (ideally $K \rightarrow \lceil \log_2 D \rceil$).
Smaller value can then be used to directly index a table.

But multiple large values may map into same smaller value causing collisions (“hash bash”)

Deal with this by:
1. Choosing mapping functions that minimise collisions e.g.
   $[1NP*>$
2. Check for collision after hashing: If table contains an entry, check if it matches the large value being hashed.
   e.g. if hash function = last 3 digits, $2227203$ might have been added to table, but $4567203$ should not match.
Example hash functions

• Choosing a **subset of the key bits**
  e.g. for student numbers or IEEE link layer addresses:
  • Least significant bits: good (differentiate individuals)
  • Most significant bits: bad
    • students: similar for same year
    • IEEE: LAN may have many devices from one manufacturer

Usually apply a more sophisticated function s.t. hashing process is insensitive to format and it is difficult to find a set of addresses that cause collisions

• Arithmetic operations
  • e.g. add digits in student number
  • more complex operations, e.g. CRC
e.g. Hashing of student numbers

Using 3 least significant digits
1000 entry table:
- 74 entries for unique IDs
- 2 for collisions

<table>
<thead>
<tr>
<th>Address</th>
<th>IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>134</td>
<td>3087134</td>
</tr>
<tr>
<td>188</td>
<td>3108188</td>
</tr>
<tr>
<td>203</td>
<td>2227203, 3114203</td>
</tr>
<tr>
<td>204</td>
<td></td>
</tr>
<tr>
<td>205</td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>3050206</td>
</tr>
<tr>
<td>207</td>
<td></td>
</tr>
<tr>
<td>208</td>
<td></td>
</tr>
<tr>
<td>209</td>
<td>3081209</td>
</tr>
<tr>
<td>217</td>
<td>3120217, 3125217</td>
</tr>
<tr>
<td>231</td>
<td>3116231</td>
</tr>
</tbody>
</table>

Least significant 2 digits of sum of pairs of digits
100 entry table:
- 36 unique entries (about half)
- 18 for collisions (worst: 4)

<table>
<thead>
<tr>
<th>Address</th>
<th>IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3 11 03 87 (3+11+3+87=104)</td>
</tr>
<tr>
<td>6</td>
<td>3 06 89 08, 3 11 97 95</td>
</tr>
<tr>
<td>7</td>
<td>3 08 05 91, 3 11 62 31</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3 08 02 96, 3 10 95 01</td>
</tr>
<tr>
<td>10</td>
<td>3 06 73 28, 3 12 17 78</td>
</tr>
<tr>
<td>11</td>
<td>3 03 93 12, 3 03 94 11, 3 07 00 01</td>
</tr>
</tbody>
</table>

Compare 7.4% or 36% utilisation with 0.78E-3% for single lookup
CRCs as hashing functions

CRCs make decent hashing functions for Ethernet switches!:

- **√** No extra computation: Must calculate CRC anyhow to detect errors. Error detection computation covers whole frame, but can take intermediate result (after processing first 6 or 12B of header) for hash value.
- **√** Frame *starts* with important keys (DA/SA), so CRC won’t vary because of preceding fields (c.f. 802.11 – hash depends on Duration)
- **√** Scramble output well

**Ethernet frame format:**

+------------------------------------------+--------------------------------+-------+-----------------+-------+
| Destination Address | Source Address | Type | Data           | CRC   |
| 6B                  | 6B             | 2B   | 46–1500B       | 4B    |

**c.f. WiFi frame header:**

+-------------------+------------------------------------------+-------------------+
| Ver | Typ | Subtype | Dir | F | R | P | D | W | O | Duration |
+-------------------+------------------------------------------+-------------------+
|                  | +------------------------------------------+-------------------+
|                  | Destination Address                       |                   |
+-------------------+------------------------------------------+-------------------+
Updating hash tables

To add an entry to the hash table:

1. Check whether the hash of the new entry collides with an existing value.

2a. If not, add new entry directly to table.

2b. If there is a clash, update the existing entry s.t. it points to classifier used to resolve collisions (e.g. linked list or CAM). Add new entry to that classifier.

Recall also, the RTL 8308 chip, used in the D-link DES-1008D shared memory switch, has a 128-entry CAM to accommodate hash bashing.
Outline

Exact match classification
  Lookup tables
  Linked lists
  Hashing
  CAMs
Random-Access Memory vs Content-Addressable Memory

Random Access Memory
Words store data associated with an address.
Address decoder selects one of $2^a$ words from memory

- $a=3=K$, $D=2K \oplus$, $E=2$

Content Addressable Memory
Words store labels, include comparison logic, and may store data
Key is compared to all words in parallel, selecting matching word (if ∃)

- $K=3$, $D=2$, $E=2$

Packet class’n scale:
- $K=48b$?
- $D=1k$?
- $E=4b$?
Reading from a CAM

Each word of memory contains:
- $K$ bits of **label storage** and **comparison** logic

Figure shows the essence of a CAM with
- 4 words with 3b labels
- $D \leq 4$, $K=3$

Matching process:
1. Key is distributed to all words simultaneously.
2. Comparison operations are done in parallel.
3. Words with matching labels generate **match signal**.
   e.g. match line like an OR gate: float (by default) high (1) and if any bit mismatches then that comparator pulls the match line low (0) => the word doesn't match
CAM details

Data needn’t be stored in any particular order

When **reading**, key value connects to comparators

When **writing**, key value connects to storage

Key in

Key in

Key in

1

0

0

1

0

0

1

1

1

1

1
Writing to a CAM

Don’t care *where* the key is stored\(^\dagger\). Usually mark free words with some key value that is distinct from valid keys.

Distinction may take the form of:
- 1b per word indicating free/used. Must specify whether matching on that bit or remaining bits – e.g. by using a mask\(^\ddagger\).
- Invalid key value indicating free word, e.g. all 0s or broadcast address

Removal:
- Search for key to remove.
- Latch match line & use to enable writing of distinct key to that word.

Insertion:
1. Search for free word.
2. Choose one of the free words (e.g. priority encoder\(^\ddagger\))
3. Latch match line & use to enable writing of distinct key to that word.

\(^\dagger\) Position may matter when multiple matches are possible and rules need to be prioritised [14U>]
\(^\ddagger\) masks and priority encoders are also useful for ternary searching [14U>]

\[=?\]
Associating data with keys in a CAM

1. Store in CAM words
2. Encode ID of matching CAM word and use to index a RAM
Example CAM: MUAA8K80

50MHz clock speed;
32b I/O => multiple cycles
80-bit width with programmable CAM/RAM partition

For Ethernet switches: Auto-learn internal aging with 9-bit time stamp

Synchronous port† for high speed packet processing
Asynchronous port for table maintenance

Package: 160 PQFP

† port of the chip, not of the switch
Figure from Music Semiconductors datasheet
SRAM circuits for RAM and CAM

1. A six-transistor CMOS SRAM cell, like the one shown in this diagram, is the basic building block of a memory array.

RAM

From http://www.elecdesign.com/Articles/Print.cfm?AD=1&ArticleID=4992
CAM seems to add THREE transistors below RAM cell, not “four or more”

CAM

4. A CAM cell is a standard SRAM cell with four or more transistors designed to implement the exclusive-or (EOR) function.
Binary and ternary CAM cells

While on the topic of circuits, but before we define ternary functionality

(a) 6-transistor SRAM cell.  (b) Binary CAM cell.  (c) Ternary CAM cell

From [http://pagiamtzis.com/cam/camintro.html](http://pagiamtzis.com/cam/camintro.html)

wl = word line,  ml=match line,  bl=bit line,  sl=search (key) line
Are CAMs the ideal classifier?

Sample specs:

<table>
<thead>
<tr>
<th>Device</th>
<th>Capacity</th>
<th>Clock Speed (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Music Semiconductors MUAA8K80</td>
<td>8K × 80b</td>
<td>50</td>
</tr>
<tr>
<td>IDT 75K62100</td>
<td>128K × 72b</td>
<td>133</td>
</tr>
</tbody>
</table>

- High speed: 64B Ethernet frames @ 1Gb/s = 2M frames/s << 50MHz (Although multiple clock cycles required to feed data in/out, e.g. 2×32b bus)
- Low storage capacity/density
  - Comparison logic for each cell
  - Relatively low density (bits/unit area)
    - e.g. 4-6 transistors for SRAM cell, “11-15”† for CAM => more space/bit for CAMs
  - High power consumption & consequently power supply/RF noise
    - e.g. 2W for 8Mb SRAM @ 200MHz, 7W for 2Mb CAM @ 50MHz
- Niche memory => not improved through competitive pressures (ala DRAM)

† Slide 1QM showed CAM with 9 transistors
Hybrid classifiers

Try using one fast selective classifier (like a cache) & if that fails, revert to another slower but comprehensive classifier, e.g.

- **Hash then CAM** (e.g. as in RTL8308 chip used in D-link DES-1008D):
  1. Try hashing first.
  2. Resolve hash bashes by storing colliding labels in a CAM. Small CAM resolves rare hash bashes without requiring variable time to search through a list.

- **CAM then trie** [HH>:
  Store most recently used labels in a CAM.
  1. Try the CAM first (fast)
  2. If CAM is inconclusive, then use a trie that stores all labels. Common cases are processed fast. Trie (large but slow) deals with exceptional cases.
Outline

Partial match classification
  Exact match techniques revisited
  Ternary CAMs
  Tries, PARTICIA trees
Can earlier classifiers do partial matching?

**Tables**
- Hard to update – e.g. to install rule with mask that doesn’t care about \( b \) bits may have to update \( 2^b \) table entries. (See [P5>: trie with \( b \) bit strides)
- No benefit from aggregation

**Lists**
- Functions OK: Prioritise by placing longest prefix first
  But performance will suffer: can’t sort to expedite search
- Search time still doesn’t scale well to many entries.

**Hashing**
- More suitable for exact matches than longest prefix (but see Varghese S 11.10, pp. 259-61)

**CAMs** – add ternary functionality …
Ternary CAMs

Ternary comparison: 0, 1, X (“Don’t care” = 0 or 1)

Don’t care bits generally specified as a mask (e.g. 1=“Do care”, 0=“Don’t”)

**Word mask**
- word-specific, stored with label

**Key mask**
- apply to all words
- reiterate search with shrinking prefix
- e.g. mask =>0Xs, then 1X, then 2Xs...

Key in
- 0/1/X
- 0/1/X
- 0/1/X
- 0/1/X

192.16.001xxxxx.X  port P
(192.16.32/19)

192.16.0010001x.X  port R
(192.16.34/23)

and prioritise match lines
s.t. lowest rule has priority

CAMs are good when excluded bits needn’t form a suffix, e.g. firewall

In electronics, tri-state outputs provide effectively ternary values: 0, 1 or high-impedance (no output).

Tr-state outputs are used when multiple outputs share a line; but only want one to write to that line at a time.

Disable comparison
Tries

Trie (from retrieval):
“a tree where each node corresponds to a string that is defined by the path to that node from the root.”

Rule Prefix

<table>
<thead>
<tr>
<th>Rule</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0*</td>
</tr>
<tr>
<td>b</td>
<td>01000*</td>
</tr>
<tr>
<td>c</td>
<td>011*</td>
</tr>
<tr>
<td>d</td>
<td>1*</td>
</tr>
<tr>
<td>e</td>
<td>100*</td>
</tr>
<tr>
<td>f</td>
<td>1100*</td>
</tr>
<tr>
<td>g</td>
<td>1101*</td>
</tr>
<tr>
<td>h</td>
<td>1110*</td>
</tr>
<tr>
<td>i</td>
<td>1111*</td>
</tr>
</tbody>
</table>

Use in partial matching: OK for longest prefix matching (X = “don’t care” bits at end of key) but won’t work when X bits in midst of string (e.g. firewall).
Implementing tries in RAM

Each node stored in **one word**, containing at most 3 ptrs:
- left: next rule if bit is 0
- centre: next rule if bit is 1
- right: rule if this is last matched bit
  (Only need right pointer if other 2 aren’t leaves. e.g. not needed for nodes 8 and 9.)

Start search at node 1, continue until either:
- match leaf node, e.g. 011 matches c
- can’t progress past a node, then use previous best match, e.g. 0101 matches a en route to blockage at node 7.

---

<table>
<thead>
<tr>
<th>RAM Addr.</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>=0, =1, rule</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2, 3, 0</td>
</tr>
<tr>
<td>2</td>
<td>0, 4, a</td>
</tr>
<tr>
<td>3</td>
<td>5, 6, d</td>
</tr>
<tr>
<td>4</td>
<td>7, c, 0</td>
</tr>
<tr>
<td>5</td>
<td>e, 0, 0</td>
</tr>
<tr>
<td>6</td>
<td>8, 9, 0</td>
</tr>
<tr>
<td>7</td>
<td>10, 0, 0</td>
</tr>
<tr>
<td>8</td>
<td>f, g, 0</td>
</tr>
<tr>
<td>9</td>
<td>h, i, 0</td>
</tr>
<tr>
<td>10</td>
<td>b, 0, 0</td>
</tr>
</tbody>
</table>

a => to port X
b => to port Y
c => to port X
...
Optimising tries

Even if trie uses lots of RAM, most recently used nodes will be stored in (processor’s) cache => Faster for recently used queries.

- Lengthening the “stride” with multi-bit tries →
- Collapsing nonbranching branches →
Multi-bit strides

Rather than binary movement at each node (at most 2 child nodes), we could make bigger “strides” at each node, e.g.:
- decimal: up to 10 children
- more likely: an integer-sized group of bits (e.g. 2 bits = up to 4 children, 3b = up to 8 children, ...)

Terminology: “stride size” = number of bits used to branch out from each node, e.g. stride of 1 = binary, stride of 3 = octal

Stride size can vary with depth in trie (see example)

Larger stride:
- √ reduces number of nodes traversed, increasing speed
- × requires larger tables that may be sparsely occupied
  (with memories increasing in size faster than increasing in speed, wasting some space for number of accesses makes sense)

A multi-bit trie is similar to hashing that directly uses a subset of the bits of the key, and then resolves hash bashes using further bits.
e.g. Multi-bit variable-stride trie

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Next-hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>35</td>
</tr>
<tr>
<td>10*</td>
<td>7</td>
</tr>
<tr>
<td>01*</td>
<td>21</td>
</tr>
<tr>
<td>110*</td>
<td>9</td>
</tr>
<tr>
<td>1011*</td>
<td>1</td>
</tr>
<tr>
<td>0001*</td>
<td>68</td>
</tr>
<tr>
<td>01011*</td>
<td>51</td>
</tr>
<tr>
<td>00110*</td>
<td>3</td>
</tr>
<tr>
<td>10001*</td>
<td>6</td>
</tr>
<tr>
<td>100001*</td>
<td>33</td>
</tr>
<tr>
<td>10000000*</td>
<td>54</td>
</tr>
<tr>
<td>100000000*</td>
<td>12</td>
</tr>
<tr>
<td>1000000000*</td>
<td>14</td>
</tr>
</tbody>
</table>

Slide by V. Sivaraman based on example from D. Taylor et al: 'Scalable IP lookup for programmable routers', Proc. Infocom, pp. 562-71
Path-compressed tries

Observation: When keys are sparse, many trie nodes have only one descendent
× Slow: Many steps lead to one result.
× Large: Many steps in memory.
Solution: Rather than use every key bit to determine subtree, record in node which bits should be used. At leaves, compare (in parallel) all bits of leaf (requiring additional storage) with key. If mis-match, then use previous best match.

Provides “path compression” by recording only genuine branches.

PATRICIA (a similar path-compression scheme) is the classification technique used with BSD Unix

When prefix searching, may have to backtrack up tree => can require 2 tree traversals => slow. [Varghese p. 245]
Summary

Task is to use $K$ (e.g. 48) bits of packet header fields to search database of $D$ (e.g. 1k) entries to determine what to do (e.g. $E=4b$ to identify output port)

Longest Prefix Matching reduces database size at expense of complexity (rules have varying lengths)

Approaches:

- **Lookup tables**: Good/viable for small (10-20b) keys => VCIs
- **Linked lists**: Slow either to search or update. Place popular/recent addresses at head of list for speed.
- **CAMs**: Perfect functionality, but high power & low density, specialist memory
- **Hashing**: Need to consider hashing collisions, e.g. using CAM or linked lists.
- **Tries**: Good for longest prefix matching, multi-bit tries save time at expense of space
Things to think about

- **Critical thinking**: Should optional protocol fields (e.g. IP and TCP options) be shifted from headers to trailers s.t. higher layer fields used for classification always occur in the same position?

- **Engineering methods**: Packet classification is a good example of optimising for the common case (and sacrificing uncommon cases). e.g. fast search at the expense of slow updates.

- **Links to other areas**: Could you apply (variants of) these classification techniques to search for patterns in other data, e.g. genes in strands of DNA, or words in audio streams? Why might differ? (e.g. handle insertions/deletions rather than matching full strings)

- **Independent learning**: Read about the variety of flow tables in modern SDN switches
  