Switching and Routing

Packet classification

Guido Maier

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Ph: +39 022399 3575, Fax +39 022399 3413
guido.maier@polimi.it

(*) – Acknowledgement: the basic material was provided by Jonathan Chao

Introduction

- To meet various QoS requirements, routers need to implement the following features
  - Admission control
  - Resource reservation
  - Per-flow queueing
  - Fair scheduling
- They need to be able to distinguish and classify the incoming traffic into different flows (flow-aware routers)
  - Flow-aware routing and packet classification is the founding principle of Software Defined Networking (SDN) and OpenFlow

Introduction

- Flows are specified by rules
  - Each rule consists of operations comparing packet fields with certain values
- A set of rules is called a classifier
  - Based on the criteria to be applied to classify packets with respect to a given network application

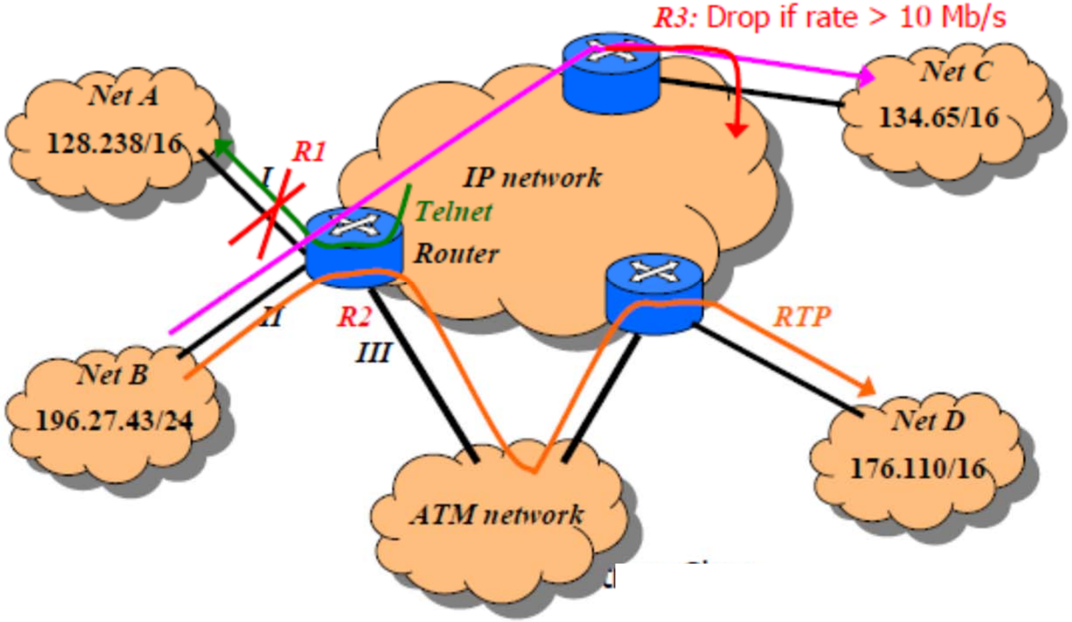
A Packet Classifier Example

Rule	IPd	IPs	Prot.	Port#	Appl	Action
R1	128.238/16	*	TCP	telnet	*	Deny
R2	176.110/16	196.27.43/24	UDP	*	RTP	Send to port III
R3	196.27.43/24	134.65/16	TCP	*	*	Drop if rate > 10 Mb/s

R1: Packet Filtering

R2: Policy Routing

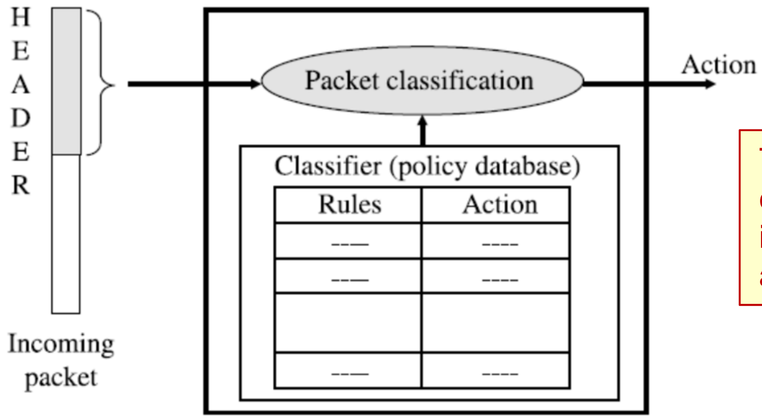
R3: Traffic Policing



Classifier definition

- A classifier C consists of N rules, R_j, $1 \le j \le N$, where R_j is composed of three entities:
  - (a) A regular expression $R_j[i]$, $1 \le i \le d$, on each of the d header fields of a packet.
  - (b) A number, $Pri(R_j)$, indicating the priority of the rule in the classifier.
  - (c) An action, referred to as $Action(R_j)$.
- An incoming packet P with the header considered as a d-tuple $(P_1, P_2, \ldots, P_d)$ is said to match R_j, if and only if, P_i matches $R_j[i]$, where $1 \le i \le d$.
- Given an incoming packet P and thus the d-tuple, the d-dimensional packet classification problem is to find the rule R_m with the highest priority among all the rules R_j matching the d-tuple

A IPv4 packet header, consisting of IP source address (32 bits), destination address (32 bits), source port number (16 bits), destination port number (16 bits), and protocol type (8 bits)



The matching rule in the classifier with the highest priority is chosen and its corresponding action is applied to the packet.

Matching the packet header to the rules in the classifier

Classifier example

TABLE 3.1 Classifier Example

	Network-Layer		Transport-Layer		Application-Layer	
Rule	Destination	Source	Protocol	Destination	Protocol	Action
R_1	128.238/16	*	TCP	= telnet	*	Deny
R_2	176.110/16	196.27.43/24	UDP	*	RTP	Send to port III
R_3	196.27.43/24	134.65/16	TCP	*	*	Drop traffic if rate > 10 Mbps
R_4	*	*	*	*	*	Permit

- Each rule has five regular expressions on five packet-header fields from network layer to application layer
- Each expression could be
  - simple *prefix/length* specification → same definition as in IP lookups
  - *operator/number* specification → could be more general, such as equal 23, range 256–1023, and greater than 1023
- A wildcard is allowed to be inserted to match any value.
- Note *R*4 'all-wildcards' specification → matches with any incoming packet
  - The priorities of rules take effect when a packet matches both *R*4 and the other rules

Classifier example

TABLE 3.2 Example Classifier with Seven Rules in Four Fields

Rule	F_1	F_2	F_3	F_4	Action
R_1	00*	110*	6	(10, 12)	Act_0
R_2	00*	11*	(4, 8)	15	Act_1
R_3	10*	1*	7	9	Act_2
R_4	0*	01*	10	(10, 12)	Act_1
R_5	0*	10*	(4, 8)	15	Act_0
R_6	0*	1*	10	(10, 12)	Act_3
R_7	*	00*	7	15	Act_1

- Rule-set $C = R_j(1 \le j \le N)$ | each rule R_j has d fields → fields are labeled as F_i $(1 \le i \le d)$ and R_j is denoted as $\langle R_{j1}, R_{j2}, \ldots, R_{jd} \rangle$
- Example classifier:
  - 7 rules, 4 fields, last column shows the action
  - The seven rules are listed in the order of descending priorities, that is, R_1 has the highest priority.
- F_1 and F_2 specified in prefixes → handled more efficiently by using tries or TCAM
- F_3 and F_4 specified in ranges → handled more efficiently by projecting the numbers into different ranges and then performing range lookup

Performance Metrics

- Several performance metrics are used to compare and analyze packet classification algorithms
- Search speed
  - In high speed links: 40-byte IP packets @ 10 Gbit/s → 31.25 Mpack/s → classification time < 32 ns
- Storage requirement
  - Small storage memory (cache or SRAM) → low access time, low power consumption
- Scalability in classifier size
  - # (micro)flows in metro/edge routers: 128k – 1 M
- Scalability in the number of header fields
  - More complex services → more header fields
- Update time
  - Some applications (e.g. flow recognition) require fast rule updating time
- Flexibility in specification
  - Wide range of rules (btw. in OpenFlow this range is very constrained)

Packet classification algorithms

- Linear search
  - The simplest algorithm for packet classification
  - Given an incoming packet header, the rules are examined one by one until a match is found
  - For a N-rule classifier, both the storage and query time complexity are O(N), making this scheme infeasible for large rule sets
- Many efficient packet classification schemes have been proposed

Packet classification algorithms

- Trie-based classification
  - Hierarchical trie
  - Set-pruning trie
  - Grid of tries
  - Extending two-dimensional schemes
  - Field-level trie classification
- Geometric algorithms
  - Cross-producting scheme
  - Bitmap intersection
  - Parallel packet classification
  - Area-based quadtree
  - Hierarchical intelligent cuttings
- Heuristic algorithms
  - Recursive flow classification
  - Tuple space search
- TCAM-based algorithms

Packet classification (reduced set)

- Trie-based classification
  - Hierarchical trie
  - Set-pruning trie
  - Grid of tries
- Geometric algorithms
  - Cross-producting scheme
  - Bitmap intersection
- Heuristic algorithms
  - Recursive flow classification
- TCAM-based algorithms

Packet classification (further reduced set)

- Trie-based classification
  - Hierarchical trie
- Geometric algorithms
  - Cross-producting scheme
  - Bitmap intersection
- TCAM-based algorithms

Packet classification

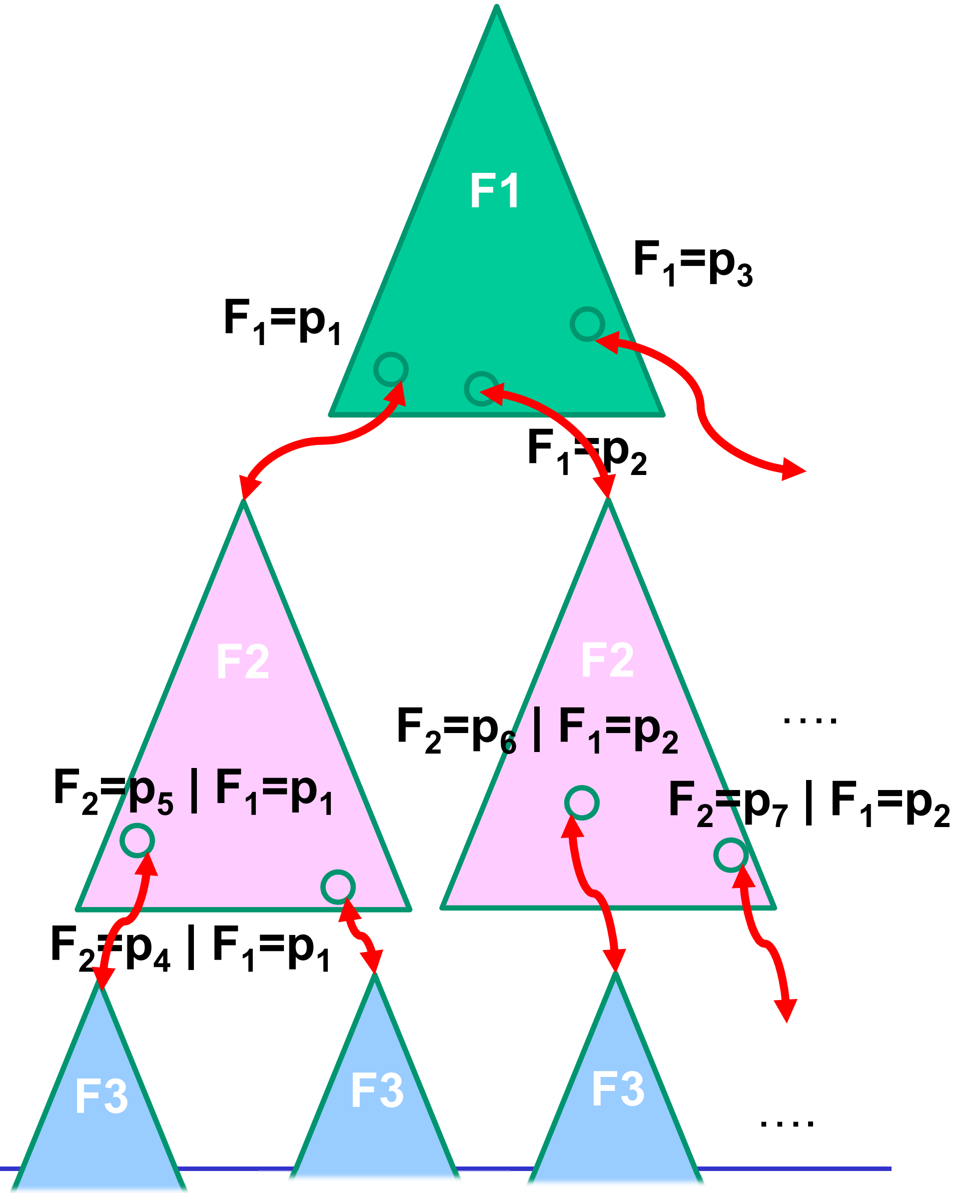
- <u>Trie-based classification</u>
  - Hierarchical trie
- Geometric algorithms
  - Cross-producting scheme
  - Bitmap intersection
- TCAM-based algorithms

The hierarchical trie data structure

- Extension of the one-dimension trie to multiple dimension
  - Each dimension represents a field
  - Also called multi-level tries, backtracking search tries, or trie-of-tries
- Prefix-based rules can be easily processed by using tries

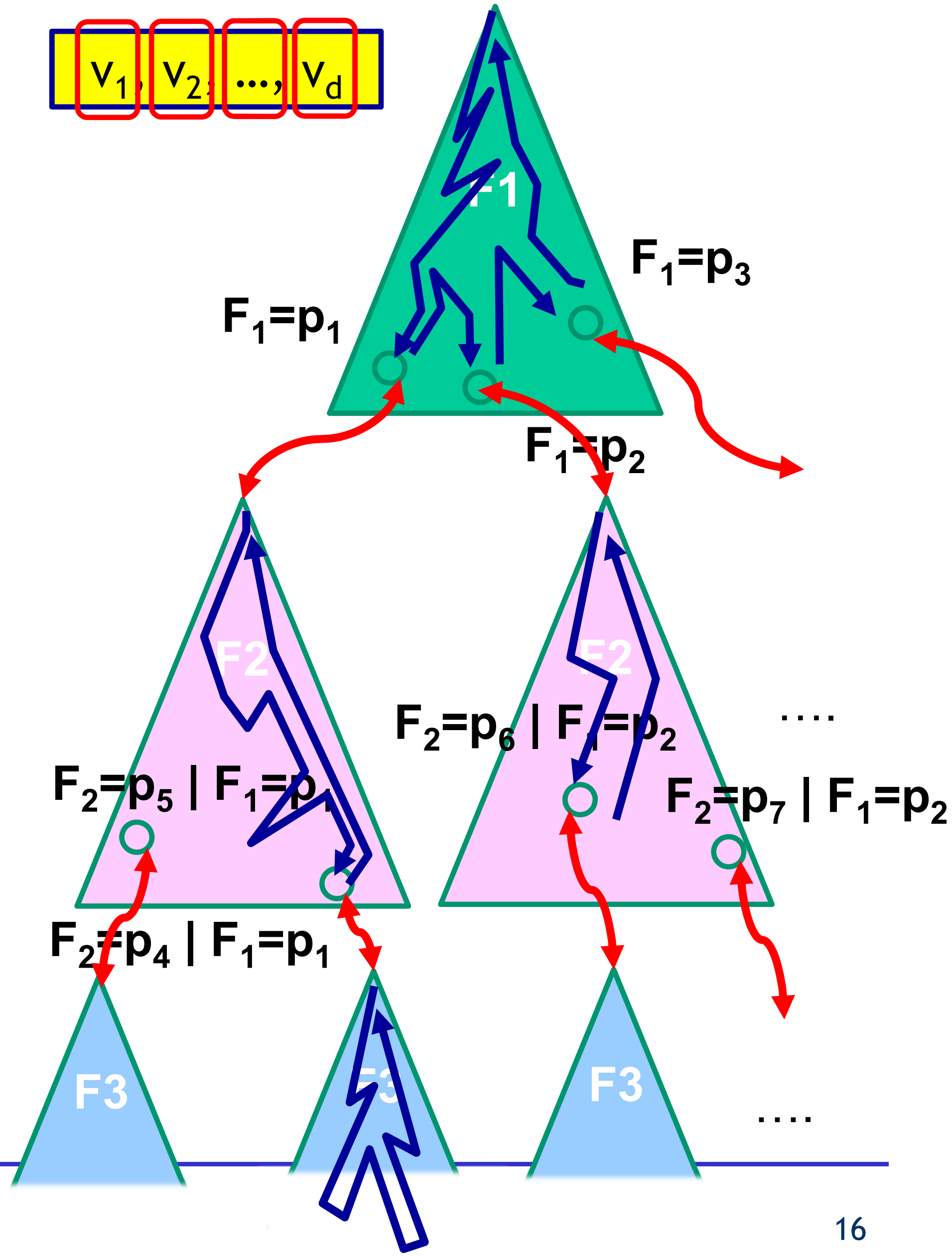
The hierarchical trie data structure

- Formal construction
  - A binary radix trie, called F_1-trie is first built for the set of prefixes $\{R_{j1}\}$ that belong to F_1 of all the rules
  - Secondly, for each prefix p_j in the F_1-trie, a (d − 1)-dimensional hierarchical trie T_p is recursively constructed for those rules that exactly specify p in F_1, that is, the set of rules $\{R_j \mid R_{j1} = p\}$
  - Trie T_p is connected to p by a next-trie pointer stored in node p



The hierarchical trie data structure

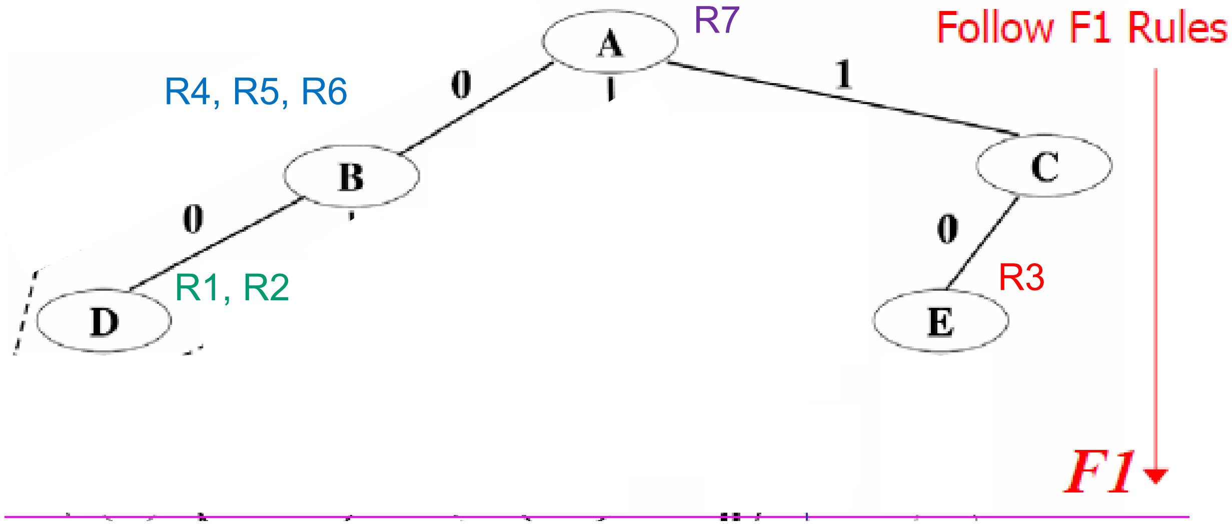
- Classification Scheme
  - Incoming packet with the header $(v_1, v_2, . . . , v_d)$
  - Query algorithm traverses the F_1-trie based on v_1
  - If a next-trie pointer is encountered, the algorithm goes on with the pointer and queries the (d – 1)-dimensional hierarchical trie recursively



The hierarchical trie data structure

C

Rule	F_1	F_2
R_1	00*	11*
R_2	00*	1*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*

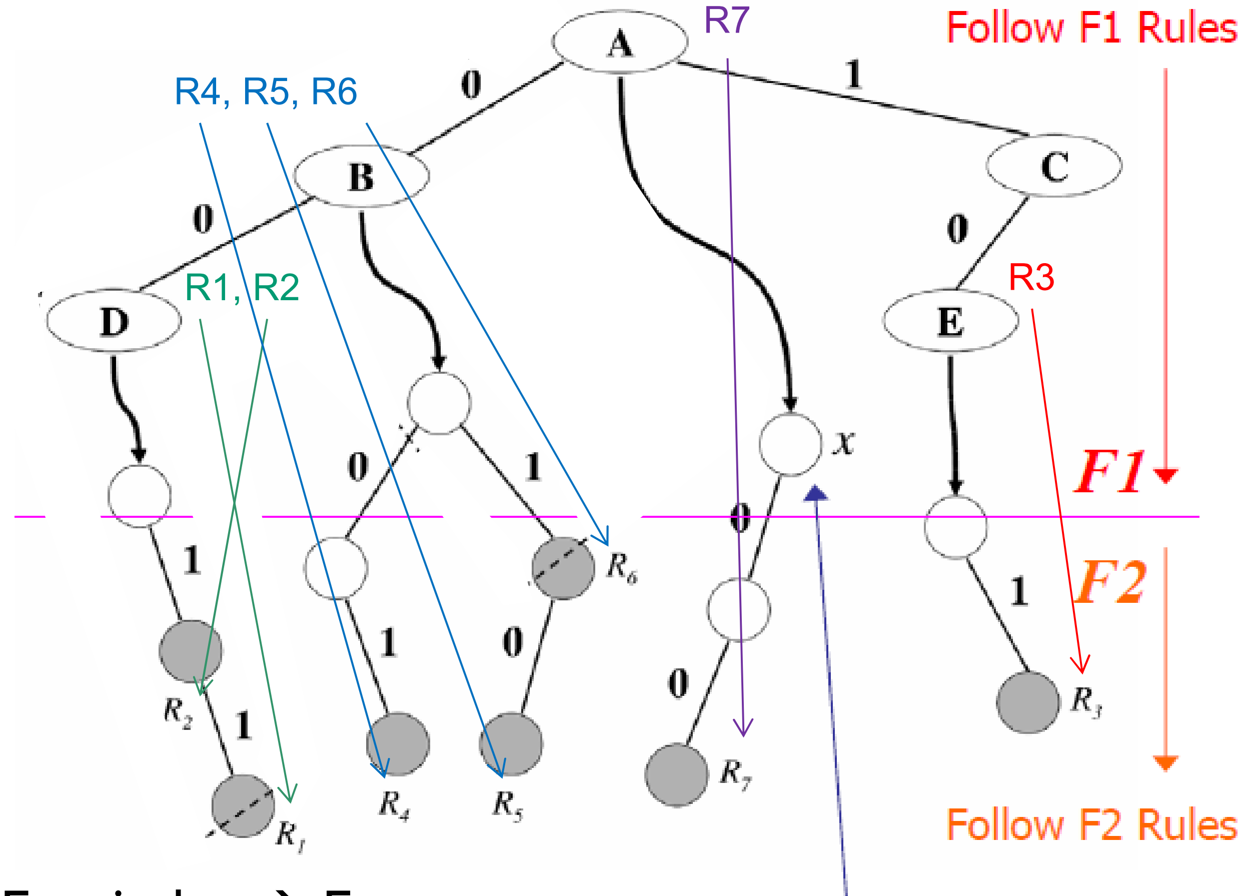


- Nodes: Elipses → F_1
-

The hierarchical trie data structure

C

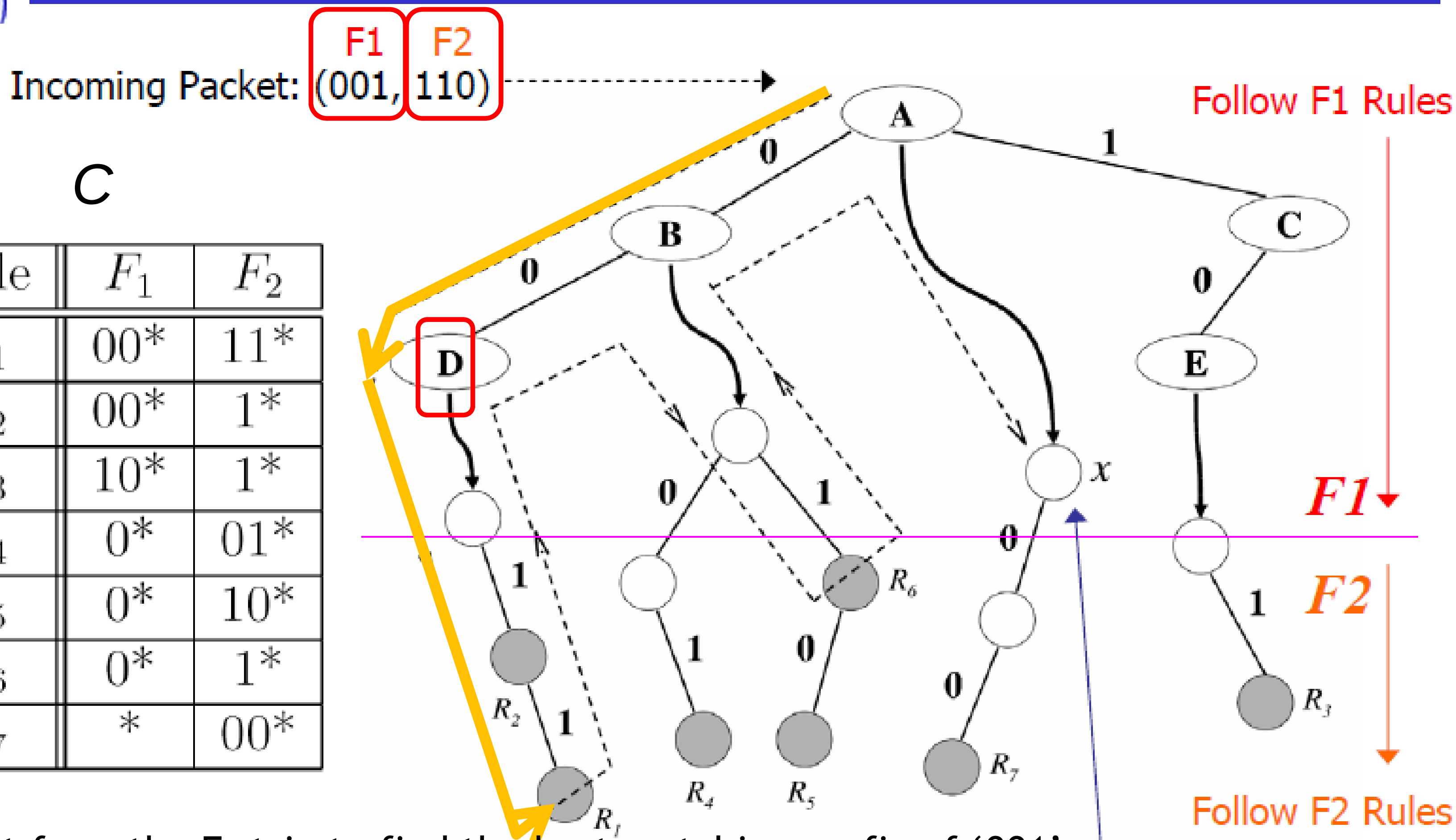
Rule	F_1	F_2
R_1	00*	11*
R_2	00*	1*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*



- Nodes: Elipses → F_1; circles → F_2
- 4 F_2-tries because we have four distinct prefixes in the F_1 field of *C*

The hierarchical trie data structure

Rule	F_1	F_2
R_1	00*	11*
R_2	00*	1*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*



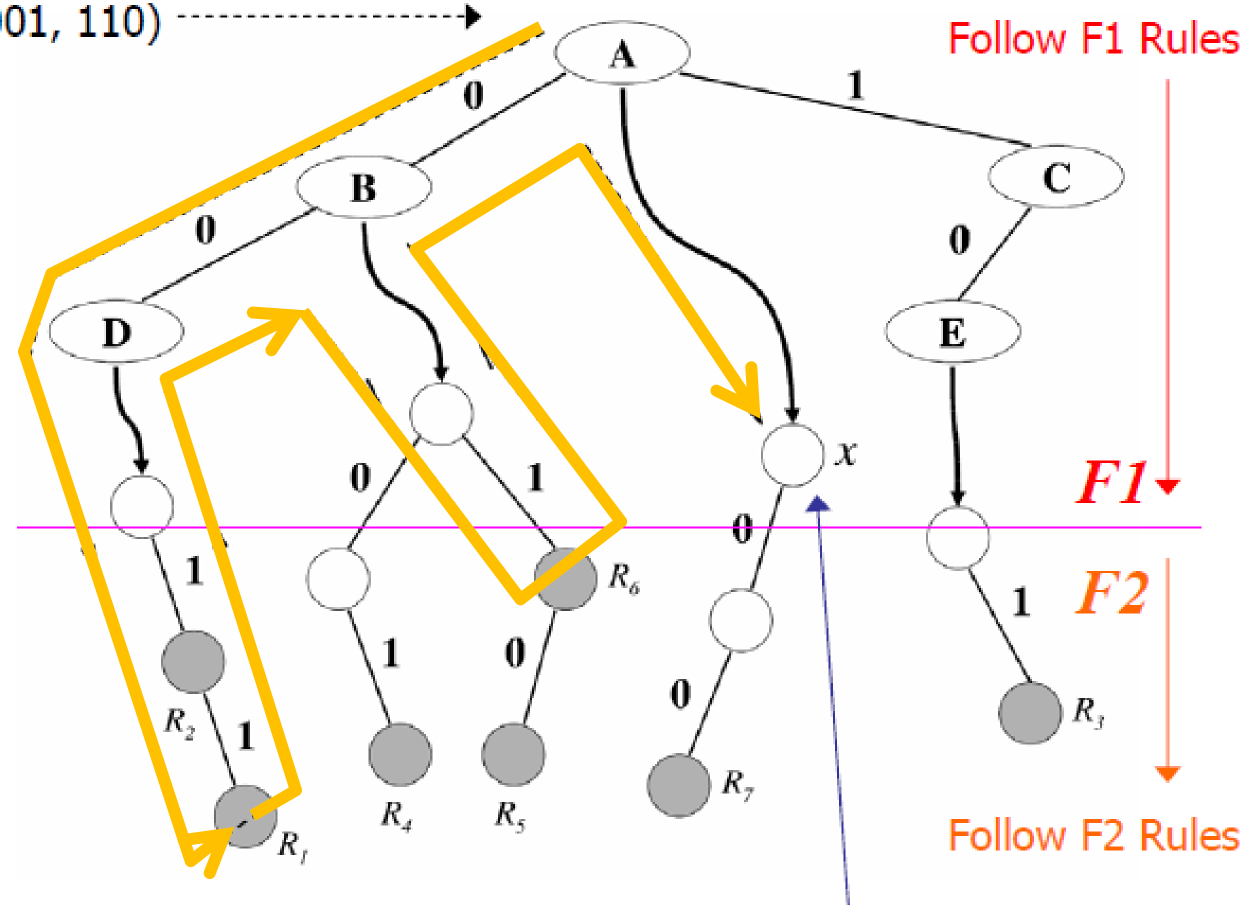
- Start from the F_1-trie to find the best matching prefix of ‘001’
- At node ‘D’, the next-trie pointer is used to guide the search into the F_2-trie to find all matching prefixes of ‘110’
- Both node R_1 and node R_2 are reached
- Only R_1 is recorded due to its higher priority

The hierarchical trie data structure

F1 F2
Incoming Packet: (001, 110)

C

Rule	F_1	F_2
R_1	00*	11*
R_2	00*	1*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*



- Search process backtracks to node ‘B’, which is the lowest ancestor of node ‘D’ in the F_1-trie
- Procedure is repeated until no ancestor node of node ‘D’ is available to be searched

The hierarchical trie data structure

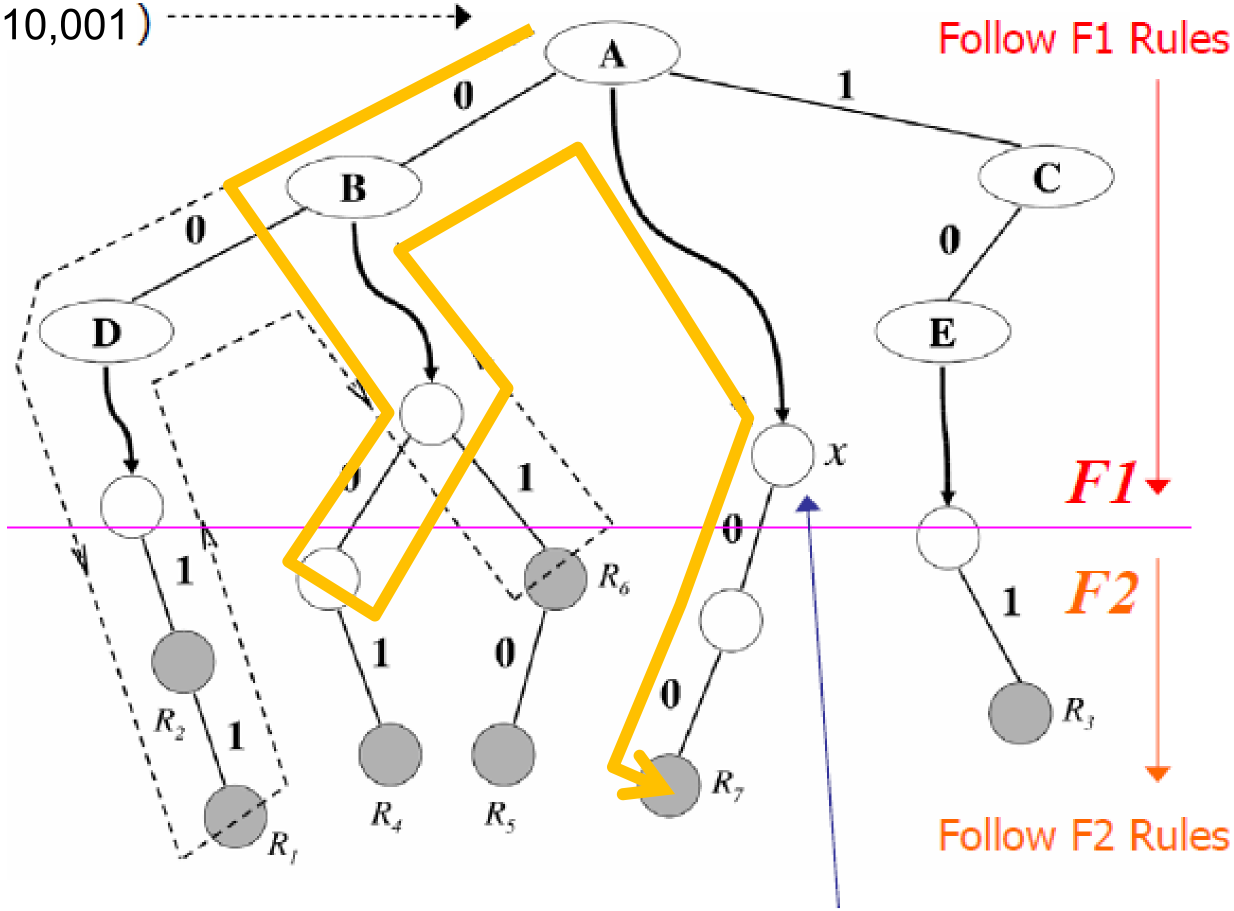
- The backtracking process is necessary, because
  - “001”of the incoming packet may match several prefixes in the first field
  - We have no knowledge in advance which F2-trie contains prefix(es) that match “110”
- During this traversal, three matches are found, R_1, R_2, and R_6. R_1 is returned as the highest priority rule matched

The hierarchical trie data structure

F1 F2
Incoming Packet: (010,001)

C

Rule	F_1	F_2
R_1	00*	11*
R_2	00*	1*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*



- R_7 returned

The hierarchical trie data structure

Performance

- N-rule set, each of which is with d sub-fields and the maximum field length of each field is W
- W: depth of F_d trie

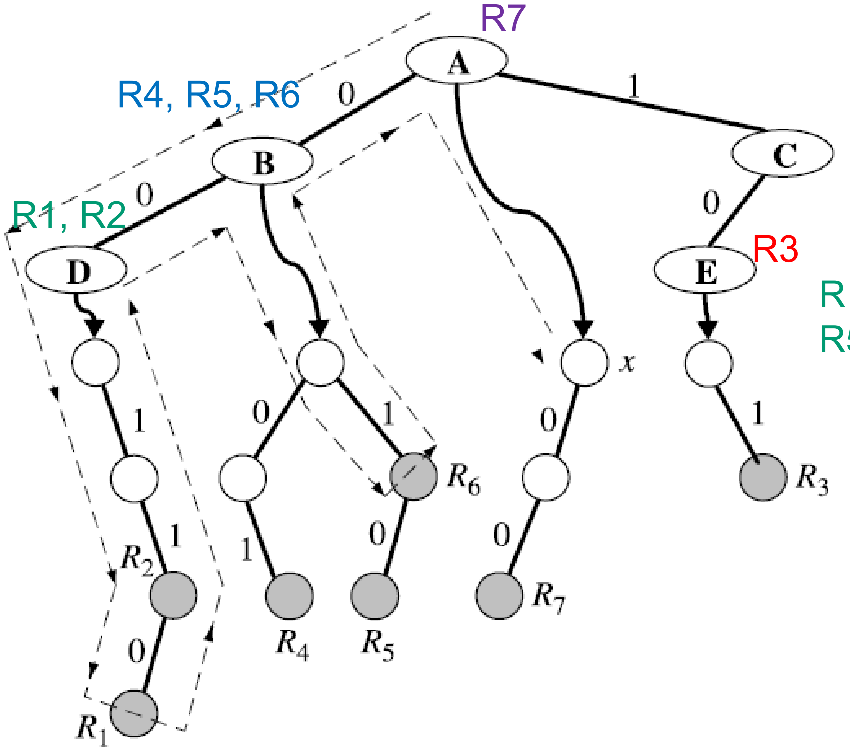
In the book is wrong!

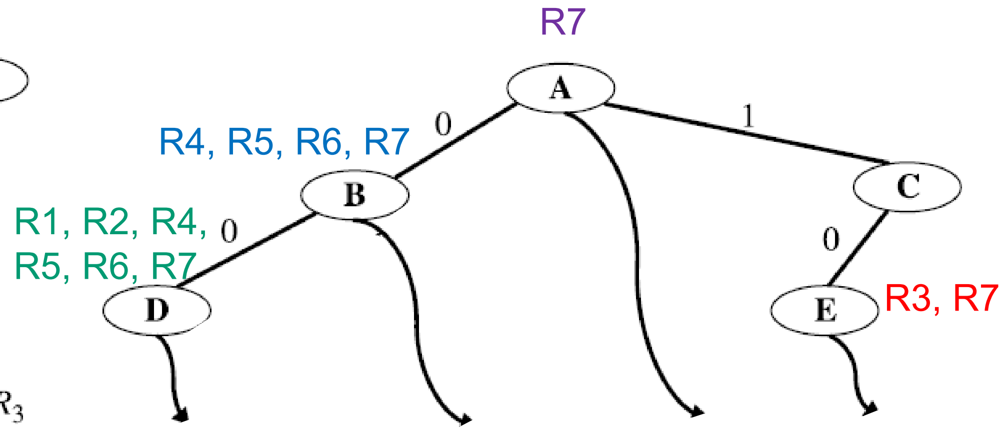
- Storage complexity: O(NdW)
  - Is one of most storage-economic algorithms; the data structure is straightforward and easy to maintain …
  - … at the expenses of a longer searching time
- Search time complexity: $O(W^d)$
  - F_d-trie has a depth of W and thus takes O(W) to search.
  - F_{d-1}-trie also has a depth of W, where each node has a F_d-trie.
  - The worst-case search time for the F_{d-1}-trie is thus $O(W^2)$.
  - With induction, the time complexity becomes $O(W^d)$.
- Update complexity: $O(d^2W)$

In the book is unclear!

  - each field of the updated rule is stored in exactly one location in a d-level tree with maximum depth O(dW)

The set-pruning trie data structure

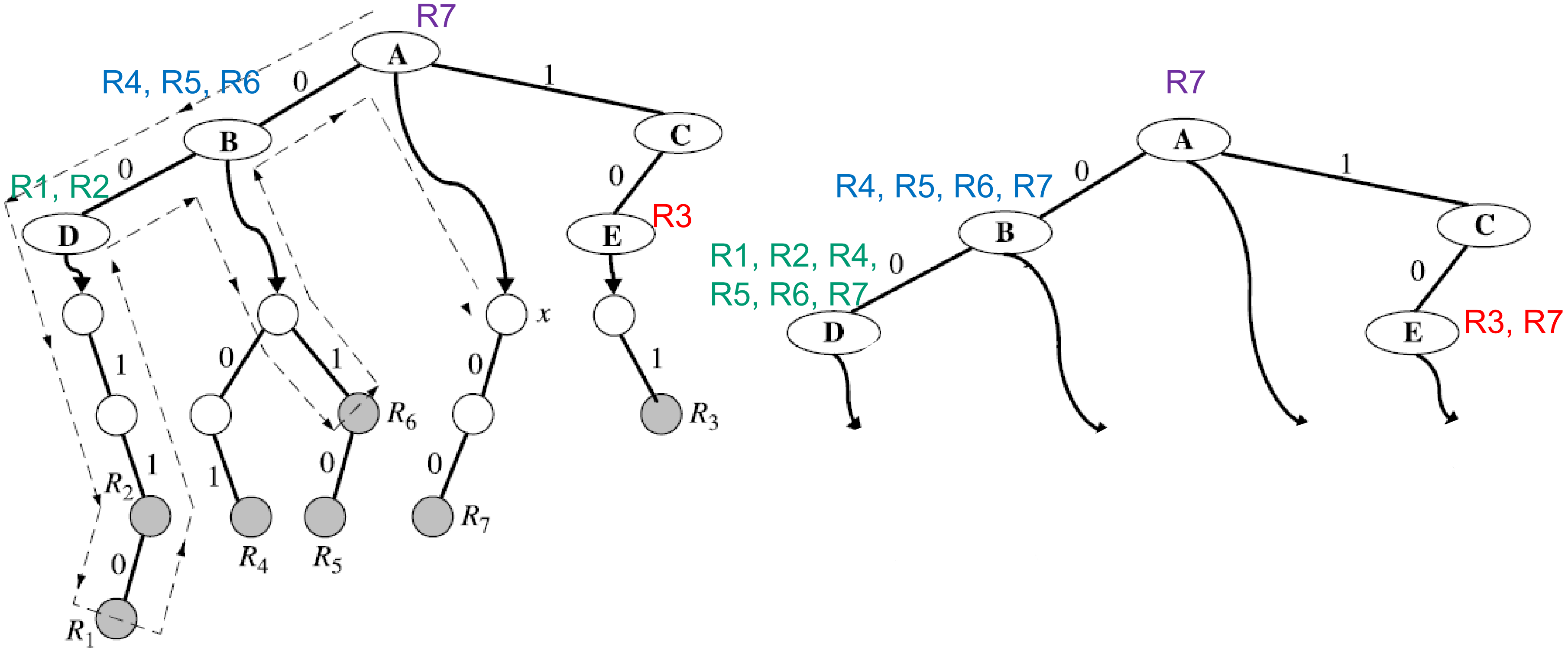




- Each trie node (with a valid prefix) duplicates all rules in the rule sets of its ancestors into its own rule set
-
-

Rule	F_1	F_2
R_1	00*	110*
R_2	00*	11*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*

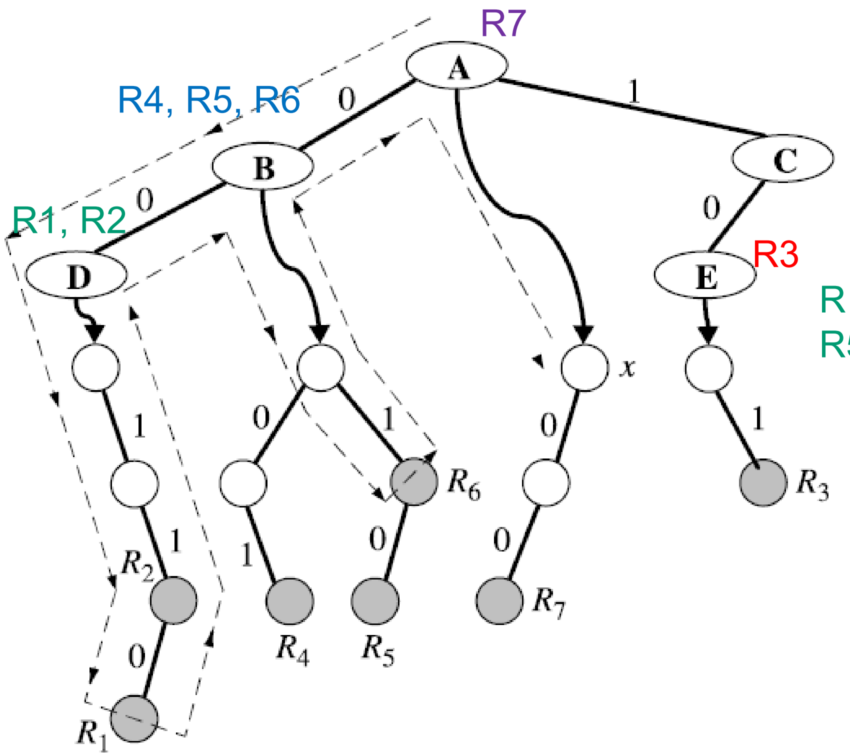
The set-pruning trie data structure

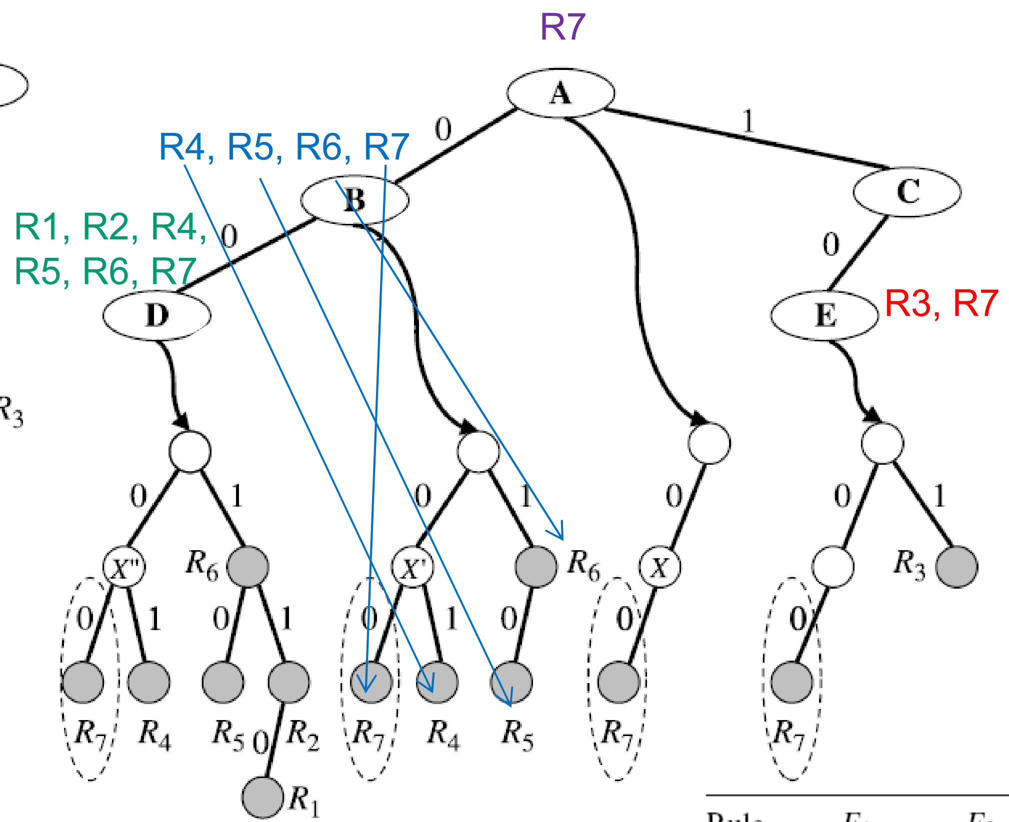


- Each trie node (with a valid prefix) duplicates all rules in the rule sets of its ancestors into its own rule set
- Then constructs the next dimension trie based on the new rule set
-

Rule	F_1	F_2
R_1	00*	110*
R_2	00*	11*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*

The set-pruning trie data structure



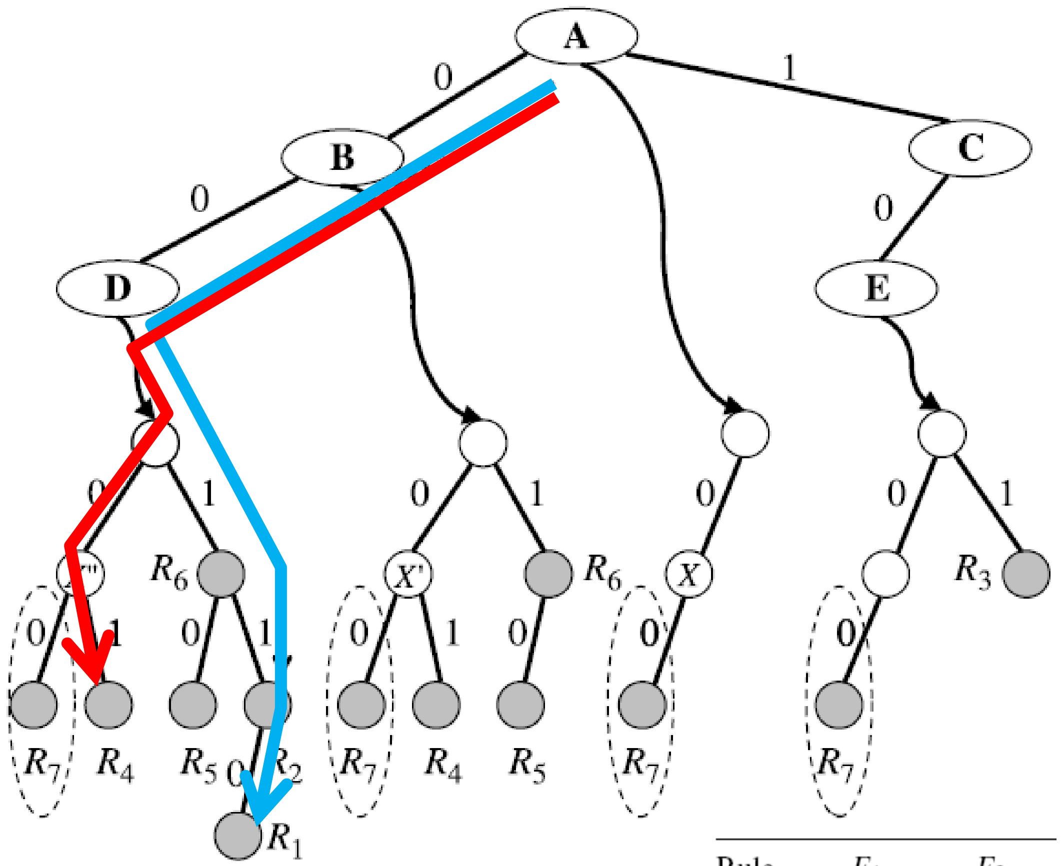


- Each trie node (with a valid prefix) duplicates all rules in the rule sets of its ancestors into its own rule set
- Then constructs the next dimension trie based on the new rule set
- Improvement: avoid backtracking during the query process

Rule	F_1	F_2
R_1	00*	110*
R_2	00*	11*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*

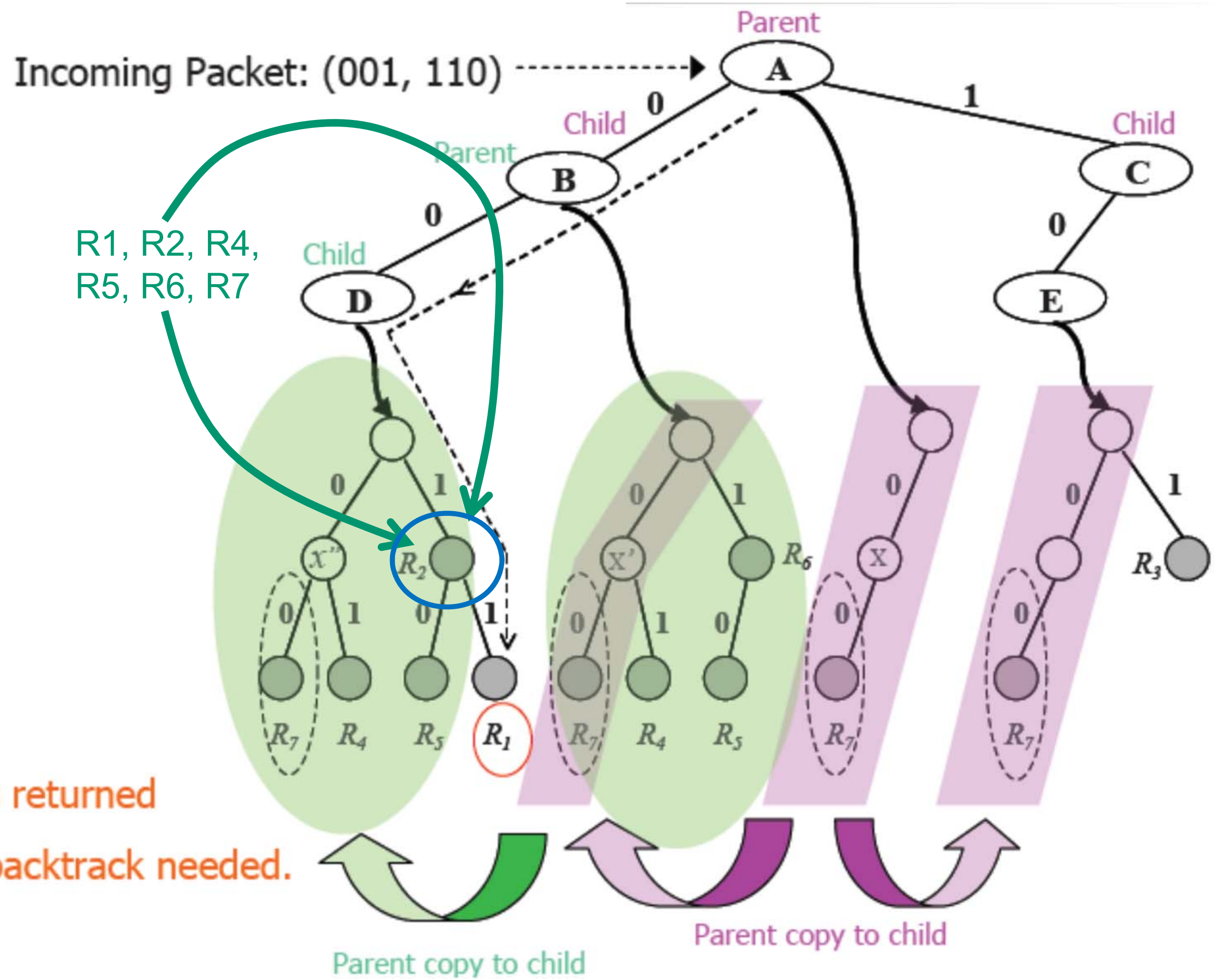
The set-pruning trie data structure

- Search process for a d-tuple consists of d consecutive longest prefix matching on each dimension of the set-pruning trie
- Classification examples:
  - (001, 110) → R1 is returned as the highest priority rule matched
  - Multiple rules may be encountered along the path (R_6, R_2) and the one with the highest priority is recorded
  - (001, 011) → R_4 is returned as the highest priority rule matched



Rule	F_1	F_2
R_1	00*	110*
R_2	00*	11*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*

The set-pruning trie data structure



R_1 is returned

no backtrack needed.

Rule	F_1	F_2
R_1	00*	11*
R_2	00*	1*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*

Note:

When Parent copy to Child, keep the higher priority node and drop the lower one. Here because R_2 has higher priority, drop R_6.

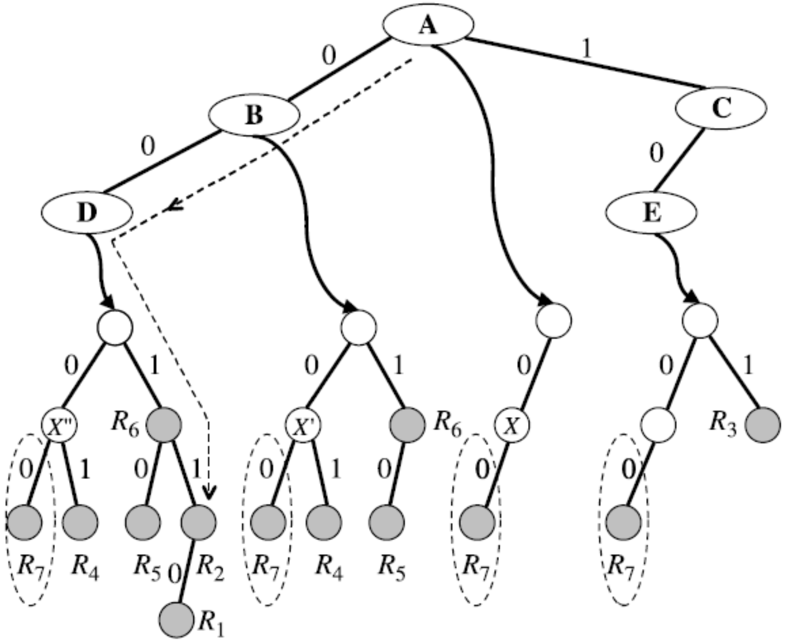
The set-pruning trie data structure

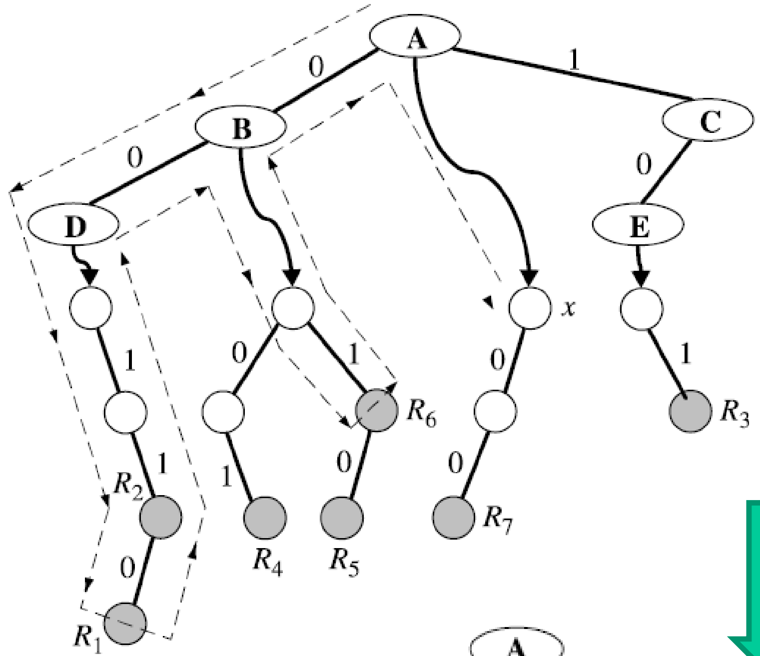
Performance

- N-rule set, each of which is with d sub-fields and the maximum field length of each field is W
- W: depth of F_d trie

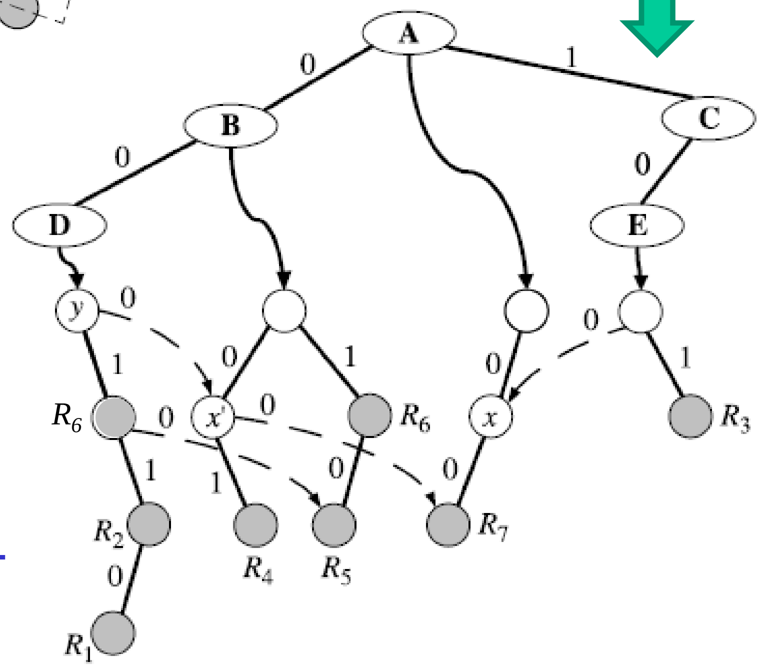
- Storage complexity: $O(N^d dW)$
  - Increased by a factor N^{d-1} due to worst-case duplication of rules (a rule may be replicated N^d times)
- Search time complexity: O(dW)
  - Elimination of backtracking → relevant improvement compared to W^d
- Update complexity: $O(N^d)$
  - Compared to $O(d^2W)$ of hierarchical tries

The grid-of-tries structure (for 2D class.)

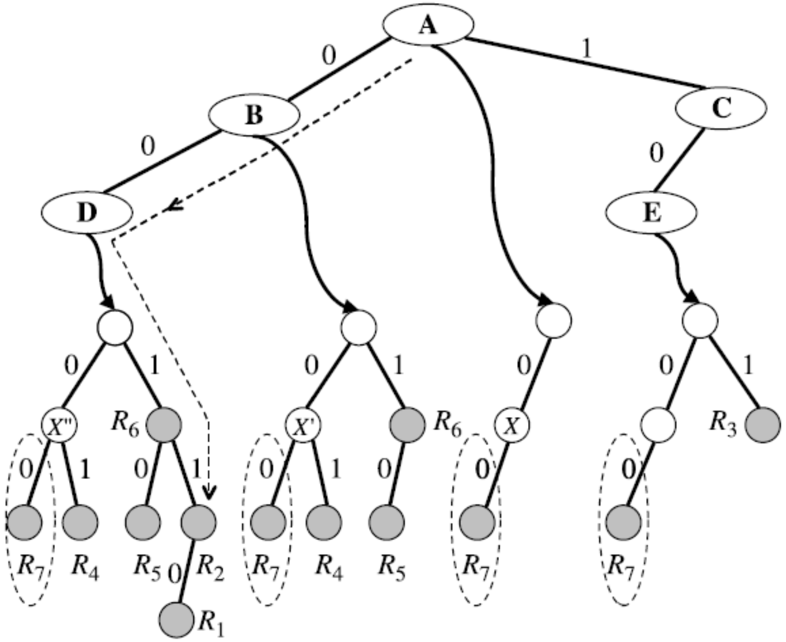


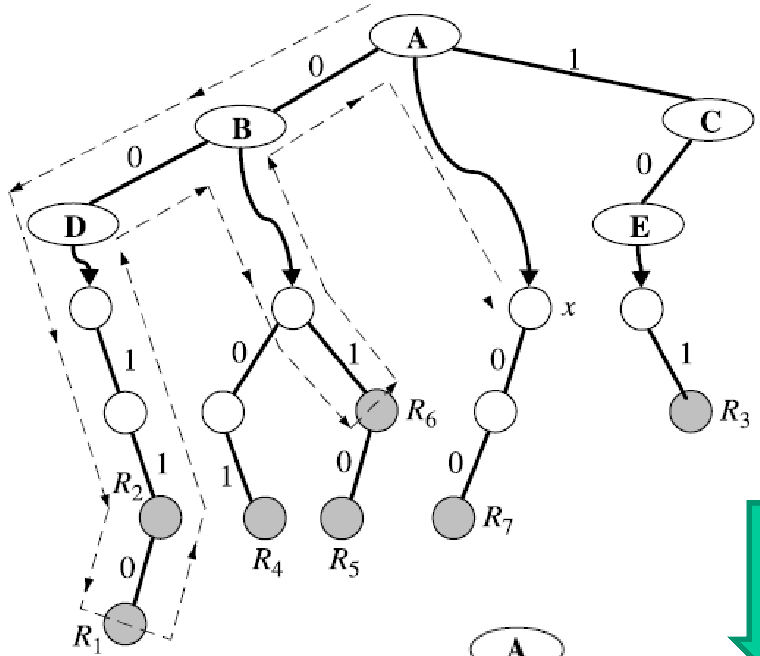


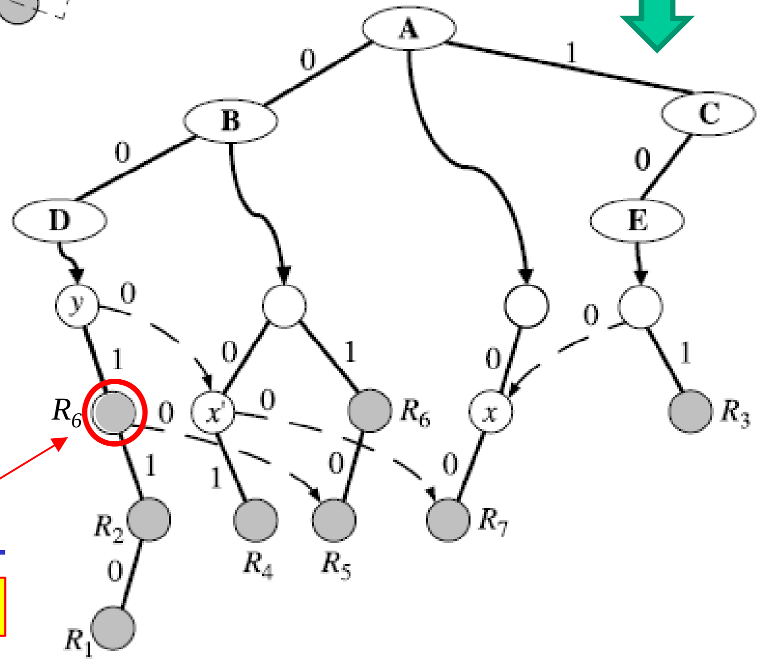
- The F_1-trie node of the set-pruning trie duplicates rules belonging to its ancestors
- This procedure could also be interpreted that the F_1-trie node merges the F_2-tries of its ancestors into its own F_2-trie
- If the hierarchical trie and set-pruning trie have been built for a classifier *C*, the grid-of-tries structure of *C* could be constructed by adding switching pointers to the F_2-tries of the hierarchical trie with comparison to that of the set-pruning trie



The grid-of-tries structure (for 2D class.)



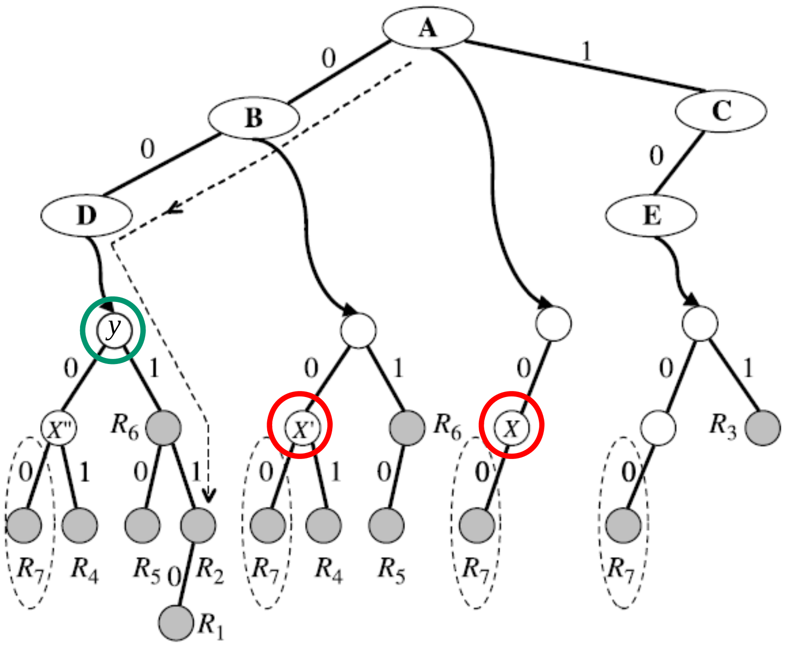


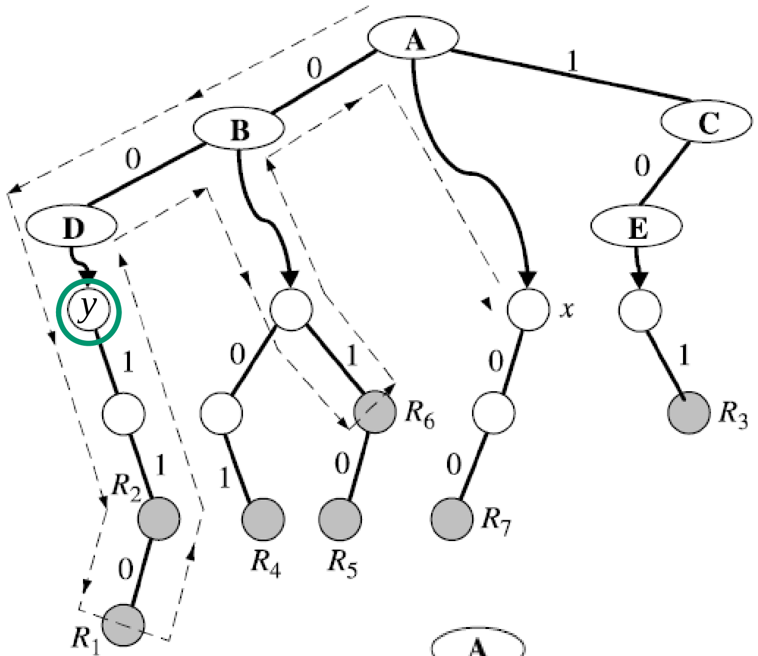


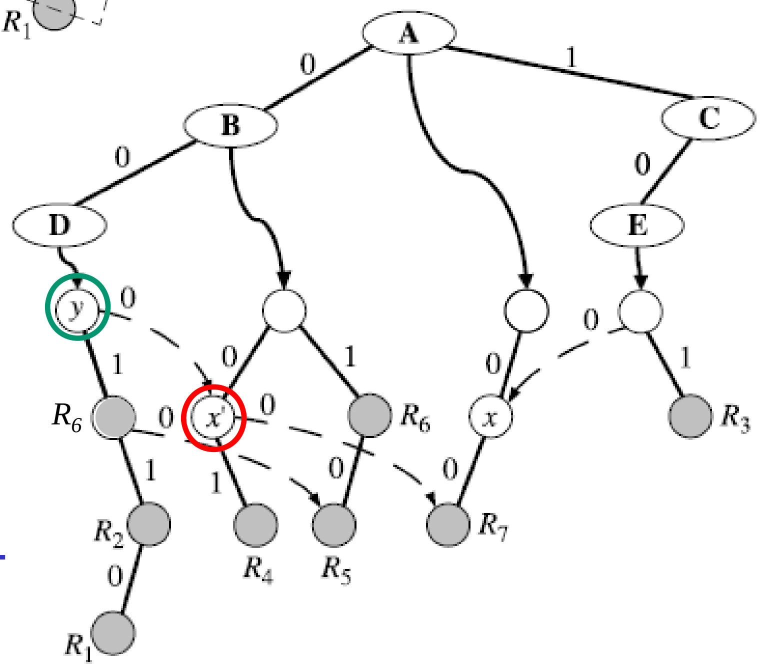
- Note that R_6 has to be reported on the F_2 trie of D, so that the F_2-D trie in the grid-of-tries is exactly equivalent to the F_2-D trie in the set-pruning structure
- Otherwise, R_6 would not be found along the search path without backtracking

In the book is wrong!

The grid-of-tries structure (for 2D class.)



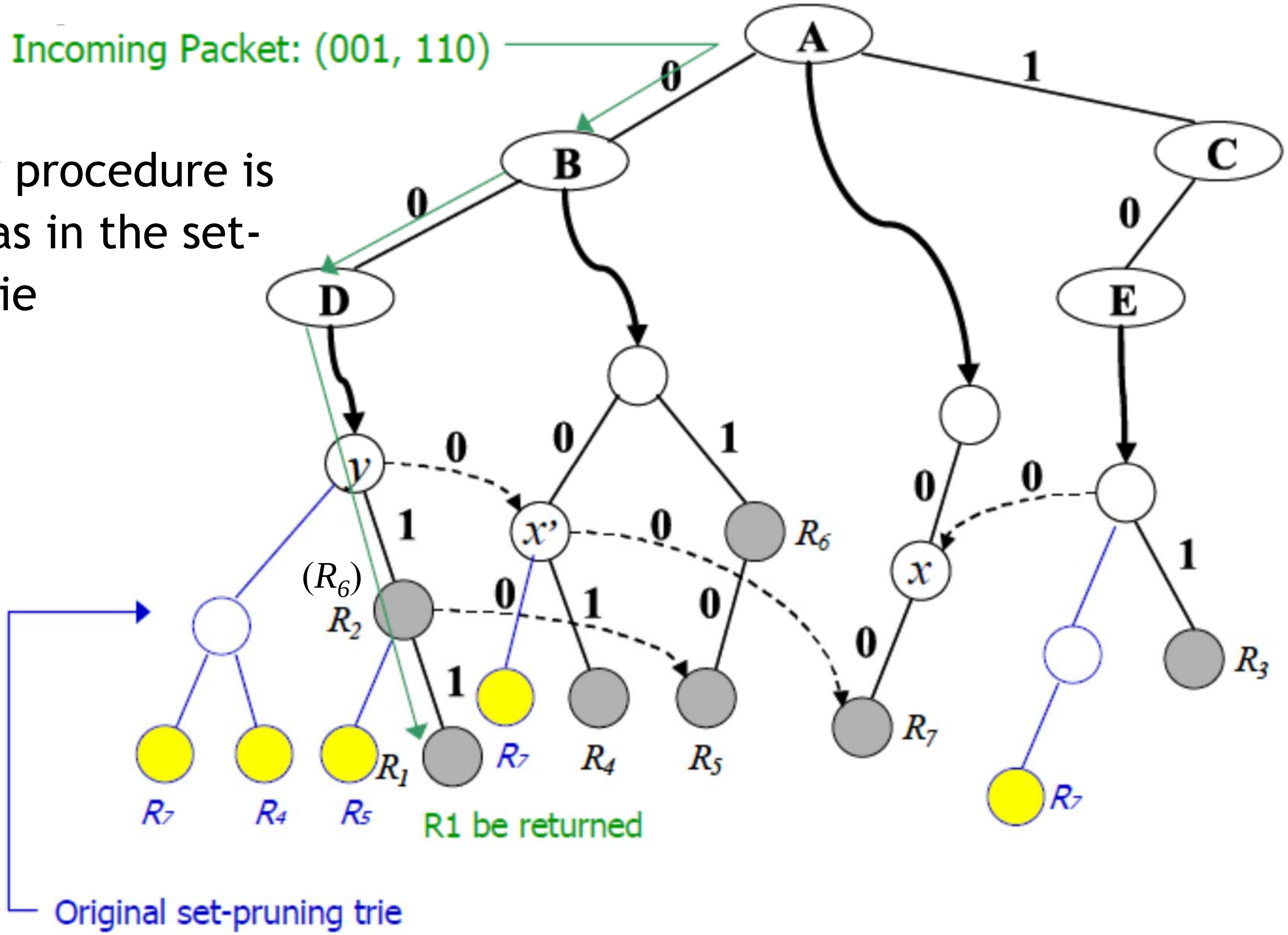




- A switching pointer, p_s, labeled with 0/1 is inserted at node y whenever its counterpart in the set-pruning trie contains a 0/1-pointer to another node z (e.g. x”) while y does not
- Node z may have several counterparts in the hierarchical trie, but p_s points to the one contained in the F_2-trie that is ‘closest’ to the F_2-trie containing node y
  - Node x and node x' are both counterparts of node x". However, the switching pointer at node y points to node x' since node B is closer to node D than node A

The grid-of-tries structure (for 2D class.)

- The query procedure is identical as in the set-pruning trie



The grid-of-tries structure (for 2D class.)

Performance

  - W: depth of F_d trie
  - N-rule set
- Storage complexity: O(NdW) → O(2NW)
  - As the hierarchical trie
- Search time complexity: O(dW) → O(2W)
  - As the set-pruning trie
- Update complexity:
  - Incremental updates are complex → the entire data structure has to be rebuild, with complexity O(NdW), at each update

- The grid-of-tries structure performs well on both query time and storage complexity
- But incremental updates are complex since several pointers may point to a single node
- If the node is to be removed, a new node needs to be created and the pointers need to be updated to point to the new node

Packet classification

- Trie-based classification
  - Hierarchical trie
- Geometric algorithms
  - Cross-producting scheme
  - Bitmap intersection
- TCAM-based algorithms

Geometric Algorithms

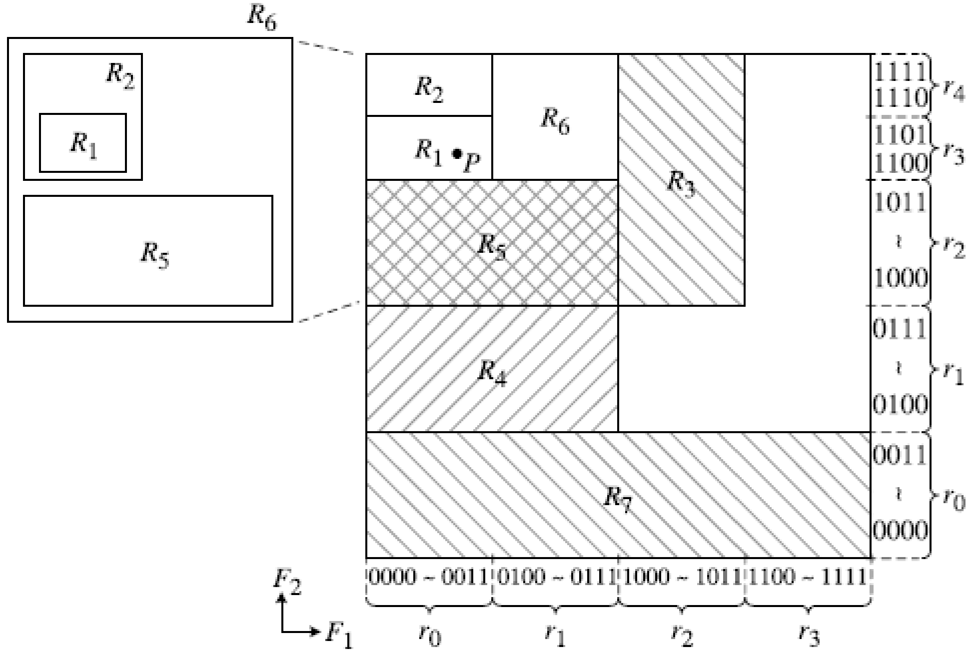
- Each field of a classifier can be specified in either a prefix/length pair or an operator/number form
- From a geometric point of view, both specifications could be interpreted by a range (or interval) on a number line
- A rule with two fields represents a <u>rectangle</u> in the 2D Euclidean space and a rule with d fields represents a d-dimensional hyper-rectangle
- The classifier is a set of such hyper-rectangles with priorities associated
- A packet header (d-tuple), it represents a <u>point</u> P in the d-dimensional space
- Packet classification problem is equivalent to <u>finding the highest priority hyper-rectangle that encloses P</u>

Geometric Algorithms

- Standard problems in the field of computational geometry that resemble packet classification
  - Point location problem: finding the enclosing region of a point, given a set of non-overlapping regions →Theoretical bounds in N (hyper-)rectangular regions and d > 3 dimensions are O(log N) time with $O(N^d)$ space [= memory], or $O((\log N)^{d-1})$ time with O(N) space
  - Packet classification is at least as hard (regions can overlap) → implies that the packet classification is extremely complex in the worst case
    - A solution is either impractically large (with 100 rules and 4 fields, $O(N^d)$ space is about 100MBytes) or too slow ($O((\log N)^{d-1})$ is about 290 memory accesses).

The geometric representation of the classifier

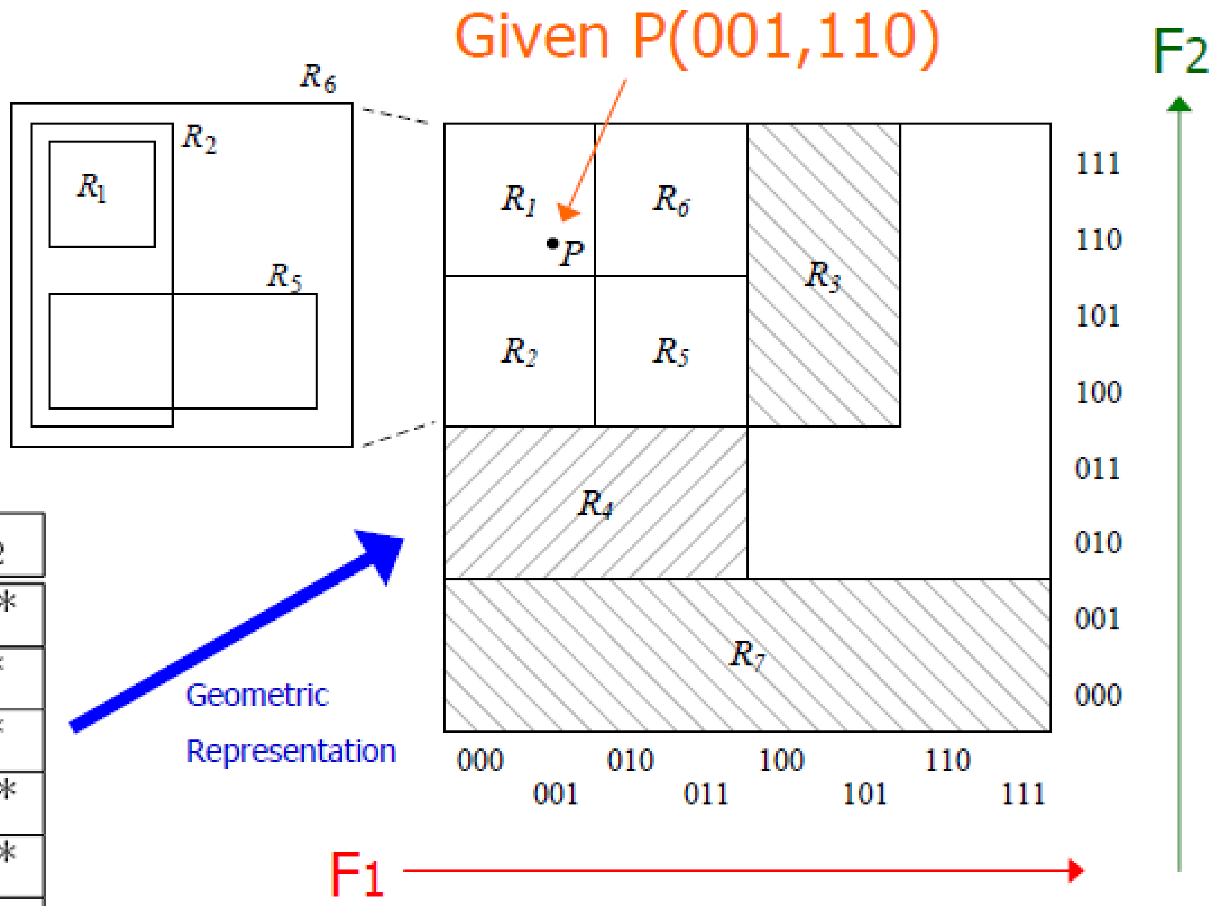
Rule	F_1	F_2
R_1	00*	110*
R_2	00*	11*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*



- If the prefixes or ranges in one field are projected on the number line $[0, 2^W - 1]$, a set of disjoint elementary ranges (or intervals) is obtained
  - The concatenation of these elementary ranges forms the whole number line
- Given a number Z on the number line, the range lookup problem is defined as locating the elementary range (or interval) containing Z
- A prefix represents a range on the number line

The geometric representation of the classifier (example)

Rule	F_1	F_2
R_1	00*	11*
R_2	00*	1*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*

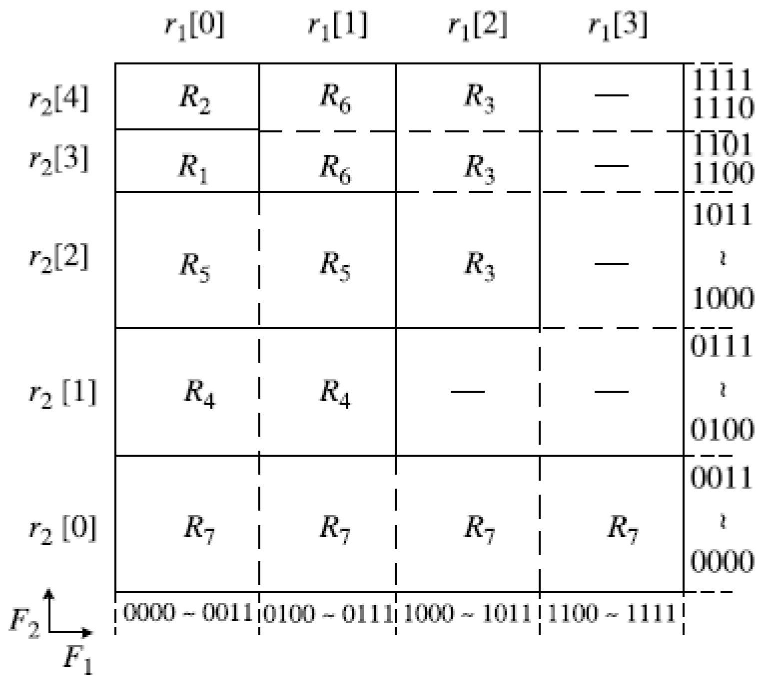


The geometric representation of the cross-producting algorithm

- The cross-producting scheme works by performing d range lookup operations, one on each field
- Compose these d results to index a pre-computed table that returns the highest priority rule matched
- For instance, given a 2-tuple (p_1; p_2), two range lookups are performed on each range set and the two matching ranges returned are composed to index the pre-computed table

The geometric representation of the cross-producting algorithm

Rule	F_1	F_2
R_1	00*	110*
R_2	00*	11*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*

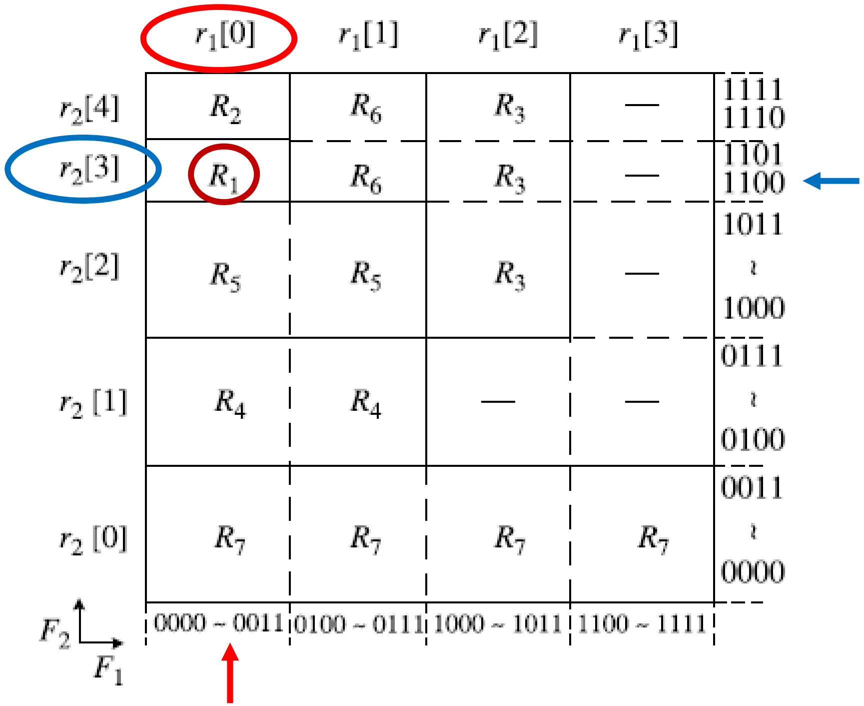


- For the first step, the rule specifications in the F_1 and F_2 fields are projected on two number lines
- Two sets of ranges $\{r_1[0], \ldots, r_1[3]\}$ and $\{r_2[0], \ldots, r_2[4]\}$, are obtained
- Each pair of ranges $(r_1[i], r_2[j])$, corresponds to a small rectangle with a pre-computed best matching rule written inside

An array with as many entries as the number of rectangles is needed in memory

The geometric representation of the cross-producting algorithm

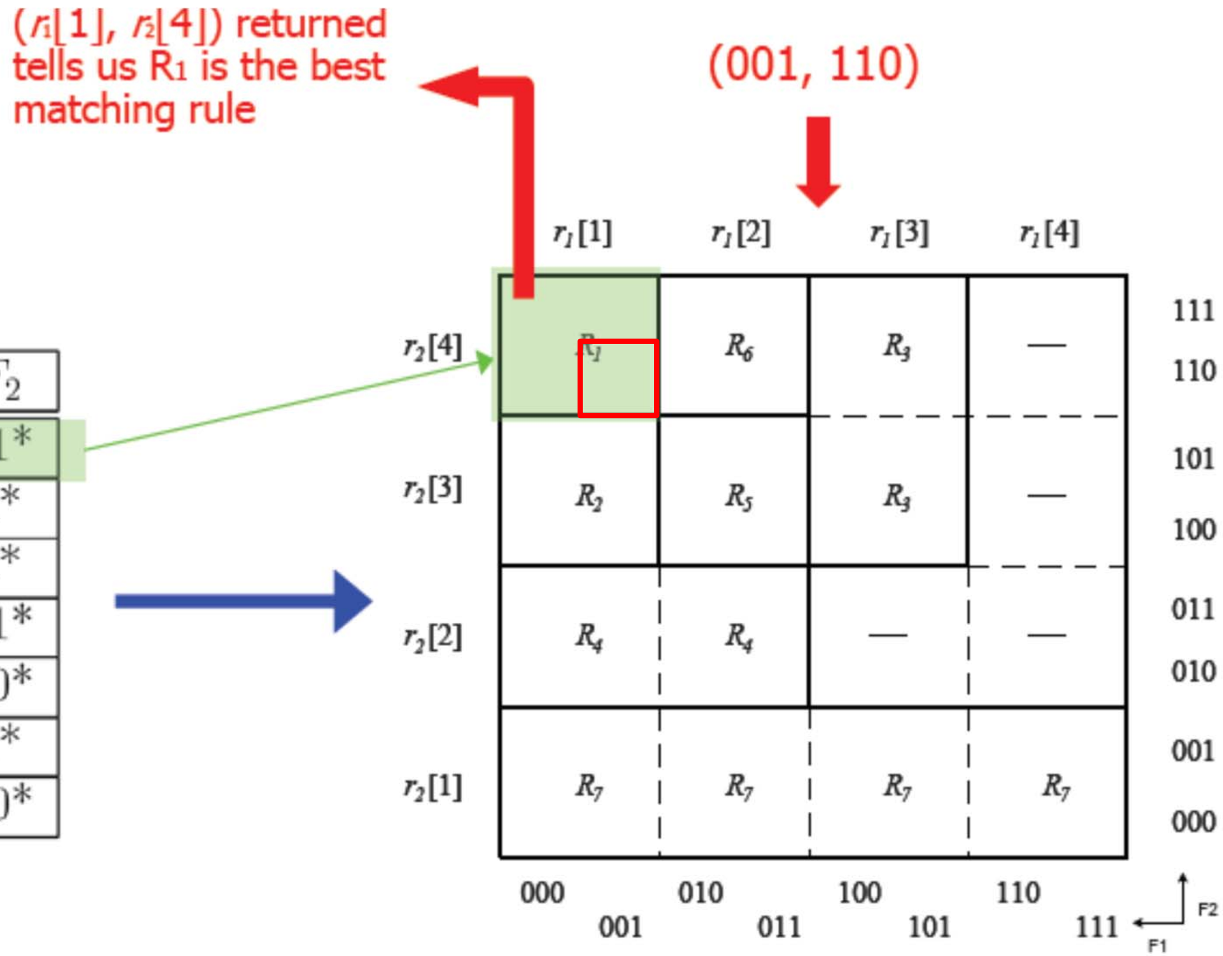
Rule	F_1	F_2
R_1	00*	110*
R_2	00*	11*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*



- Given a 2-tuple (p_1, p_2), two range lookups are performed on each range set and the two matching ranges returned are composed to index the pre-computed table
  - If p_1 = 0010 and p_2 = 1100, the two returned ranges (r_1[0], r_2[3]), tell us that R_1 is the best matching rule

The geometric representation of the cross-producting algorithm

Rule	F_1	F_2
R_1	00*	11*
R_2	00*	1*
R_3	10*	1*
R_4	0*	01*
R_5	0*	10*
R_6	0*	1*
R_7	*	00*

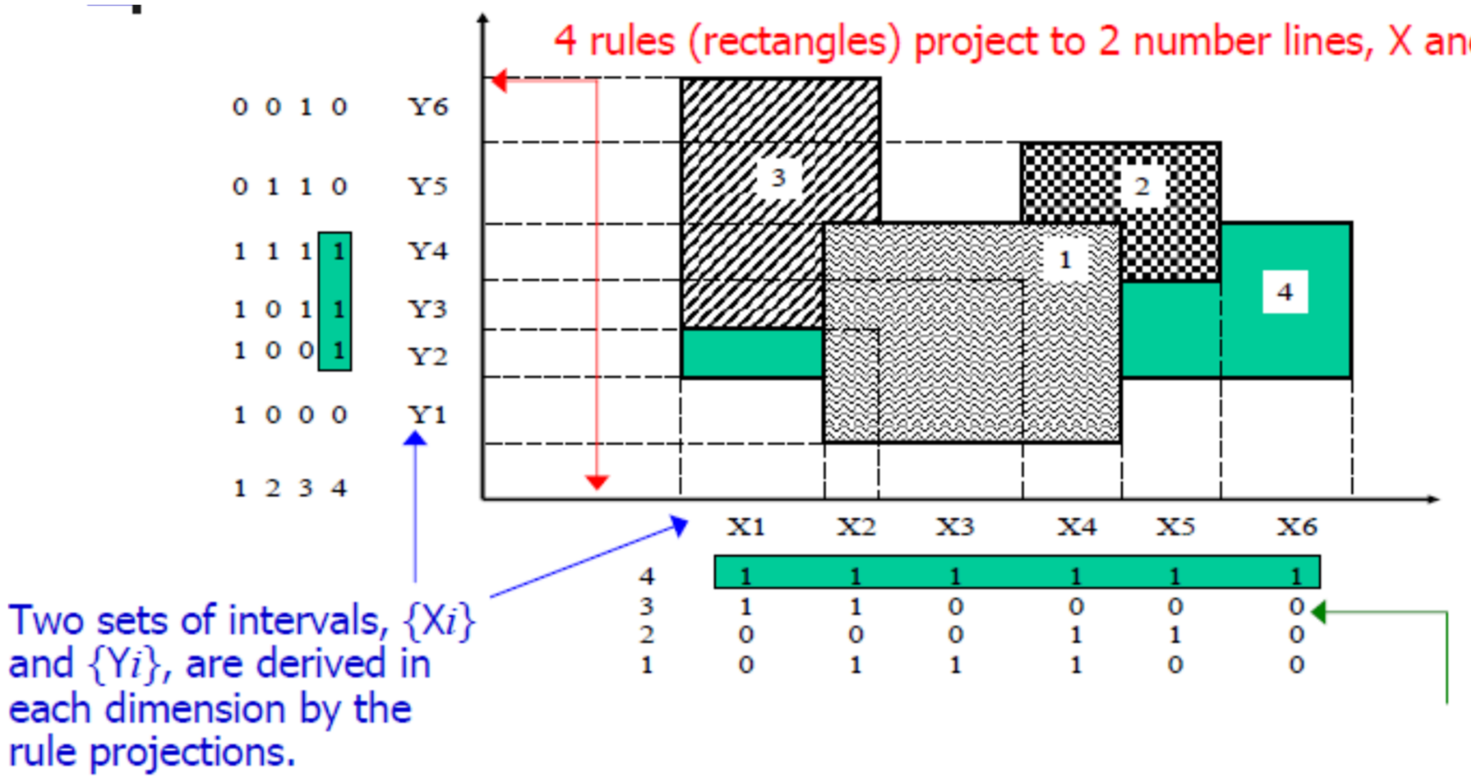


Cross-producting algorithm performance

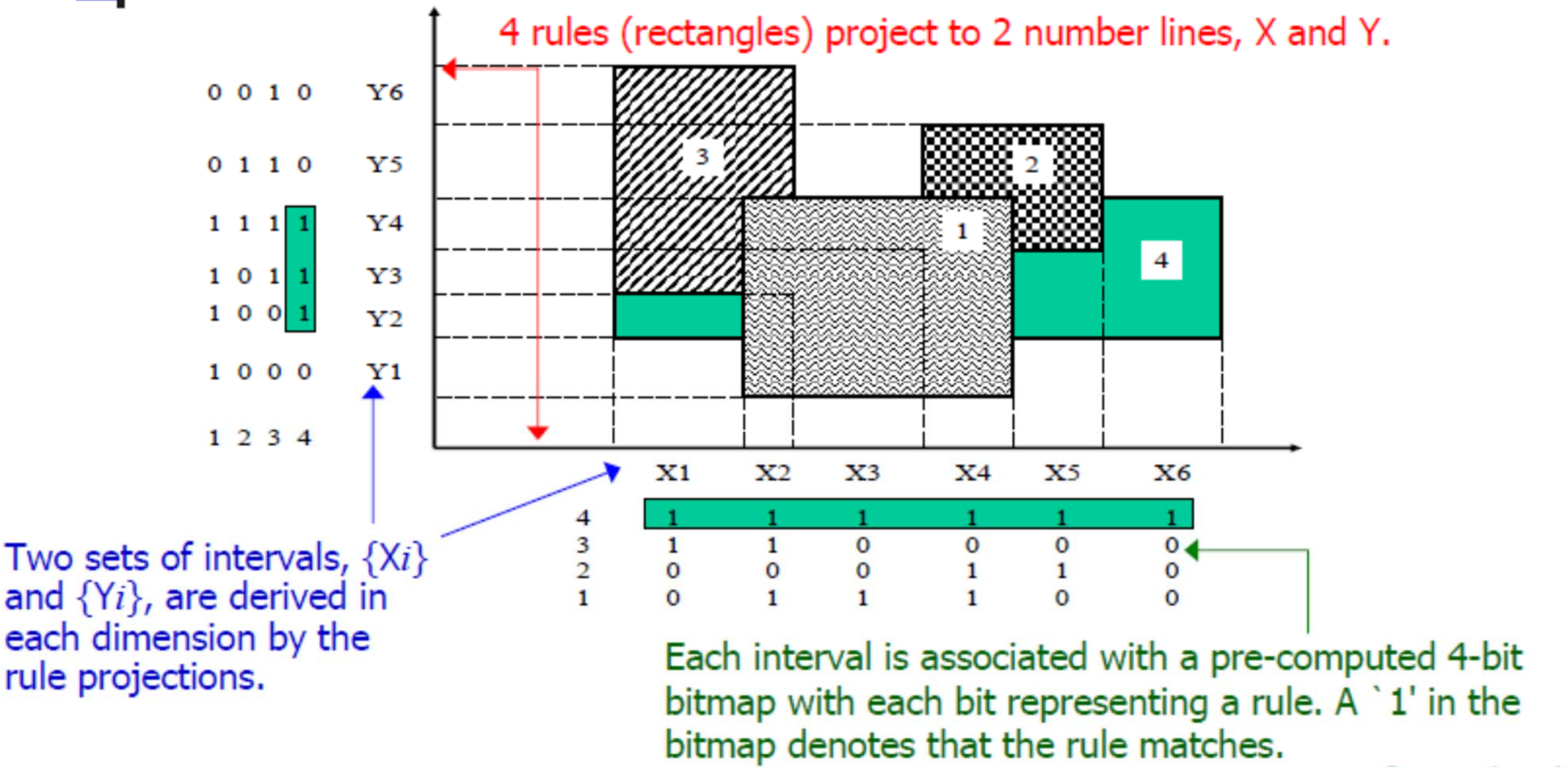
- The storage complexity
  - Suffers from a memory explosion problem in the worst case, when the cross-product table can have $O(N^d)$ entries
    - It can be proven that N prefixes leads to at most 2N – 2 ranges on each dimension
- The search time complexity: $O(d\ t_{RL})$
  - t_{RL} is the time complexity of finding a range in one dimension.
- The update complexity
  - Incremental updates require reconstruction of the cross-product table, so it cannot support dynamic classifiers well

The geometric representation of the bitmap-insertion scheme for a 2D classifier

- The set of rules, S, that matches a packet is the intersection of d sets, S_i, where S_i is the set of rules that matches the packet in the ith dimension alone
- Four rules of a 2D classifier are depicted as four rectangles and projected on the two number lines

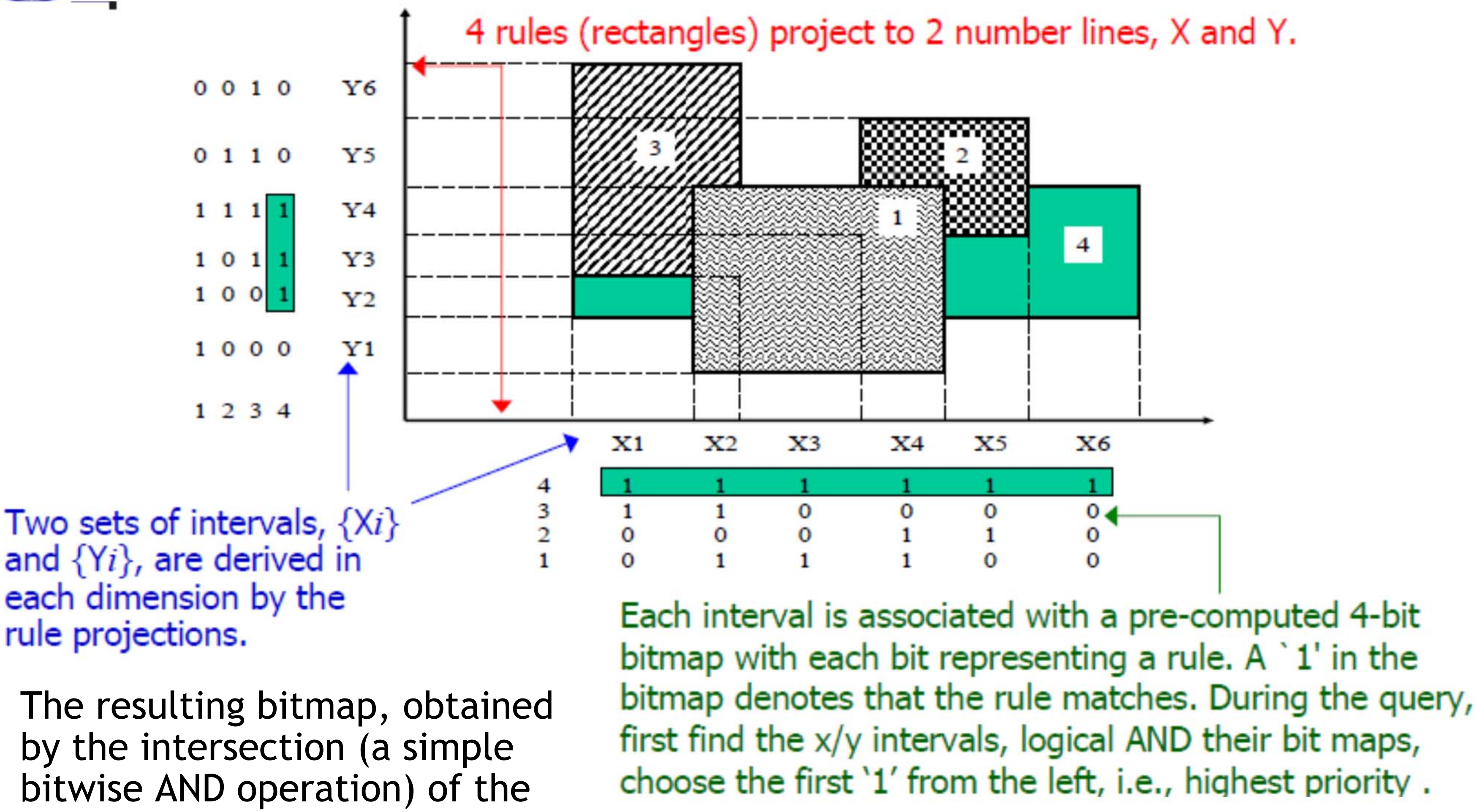


The geometric representation of the bitmap-insertion scheme for a 2D classifier



- Given a packet P(p_1, p_2), two range lookups (e.g., using a multiway search tree) are performed in each interval set and two intervals, X_i and Y_j, which contain p_1 and p_2, are determined

The geometric representation of the bitmap-insertion scheme for a 2D classifier



- The resulting bitmap, obtained by the intersection (a simple bitwise AND operation) of the bitmaps of X_i and Y_j, shows all matching rules for P
- If the rules are ordered in decreasing order of priority, the first '1' in the bitmap denotes the highest priority rule

The geometric representation of the bitmap-insertion scheme for a 2D classifier

- Applies to multi-dimensional packet classification with either type of specification in each field
- Based on the observation that the set of rules, S, that match a packet is the intersection of d sets, S_i, where S_i is the set of rules that match the packet in the *i*th dimension alone

The bitmap-insertion scheme
Performance

- The storage complexity: O (dN^2)
  - since each bitmap is N bits wide, and there are O(N) ranges in each of the d dimensions
- The search time complexity: O $(dt_{RL}+dN/w)$
  - t_{RL} is the time to perform one range lookup
  - *w* is the memory width
  - Time complexity can be reduced by a factor of *d* by looking up each dimension independently in parallel
- The update complexity
  - Incremental updates are not well-supported
- Up to 512 rules with a 33-MHz FPGA and five 1-Mbyte SRAMs, classifying 1 mpps
  - works well for a small number of rules in multiple dimensions, but suffers from a quadratic increase in storage and linear increase in classification time with the size of the classifier

Packet classification

SKIP

- Trie-based classification
  - Hierarchical trie
  - Set-pruning trie
  - Grid of tries
- Geometric algorithms
  - Cross-producting scheme
  - Bitmap intersection
- <u>Heuristic algorithms</u>
  - Recursive flow classification
- TCAM-based algorithms

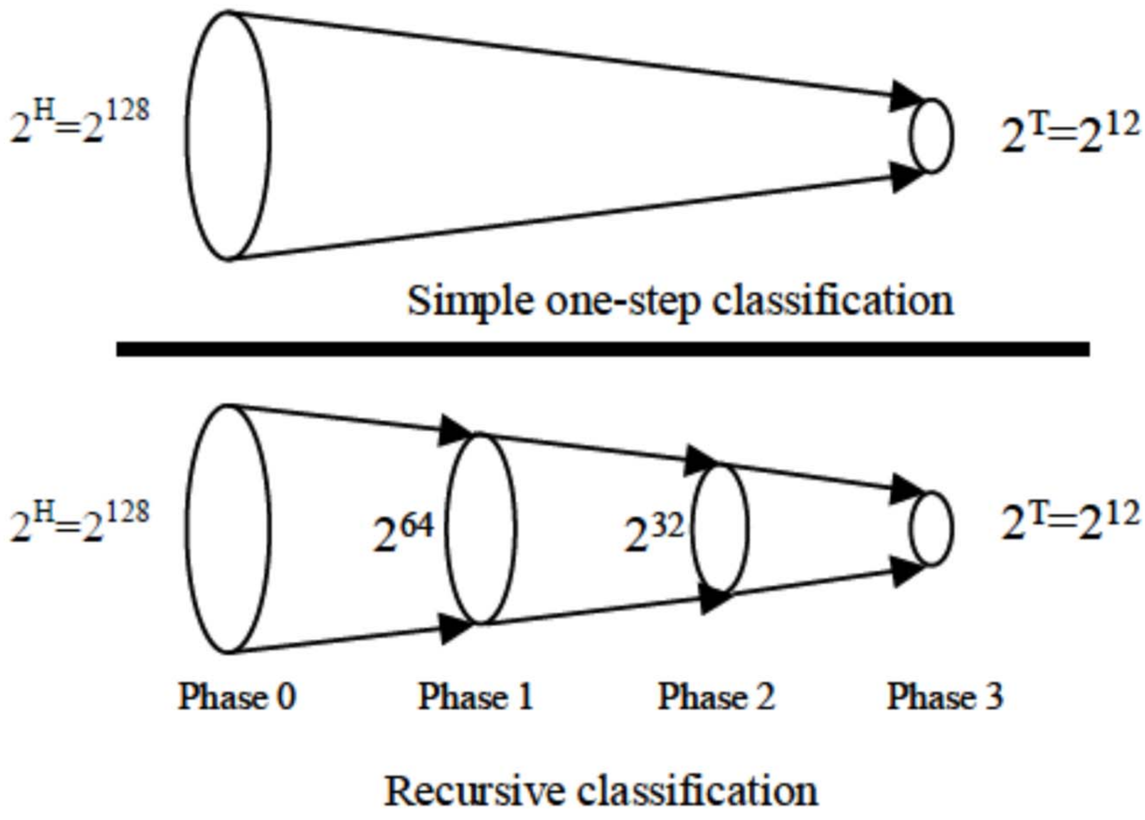
Heuristic algorithms

- Given the complexity of the packet classification problem, a simplification is desirable by slicing the search spaces in subspaces
- The optimum is found in most of the cases, but it is not guaranteed

Recursive Flow Classification (RFC)

- Basic idea of the Recursive Flow Classification (RFC)

Classifying a packet involves mapping the *H*-bit packet header to a *T*-bit action identifier (where $T = logN$, $T << H$) for a N-rule classifier.

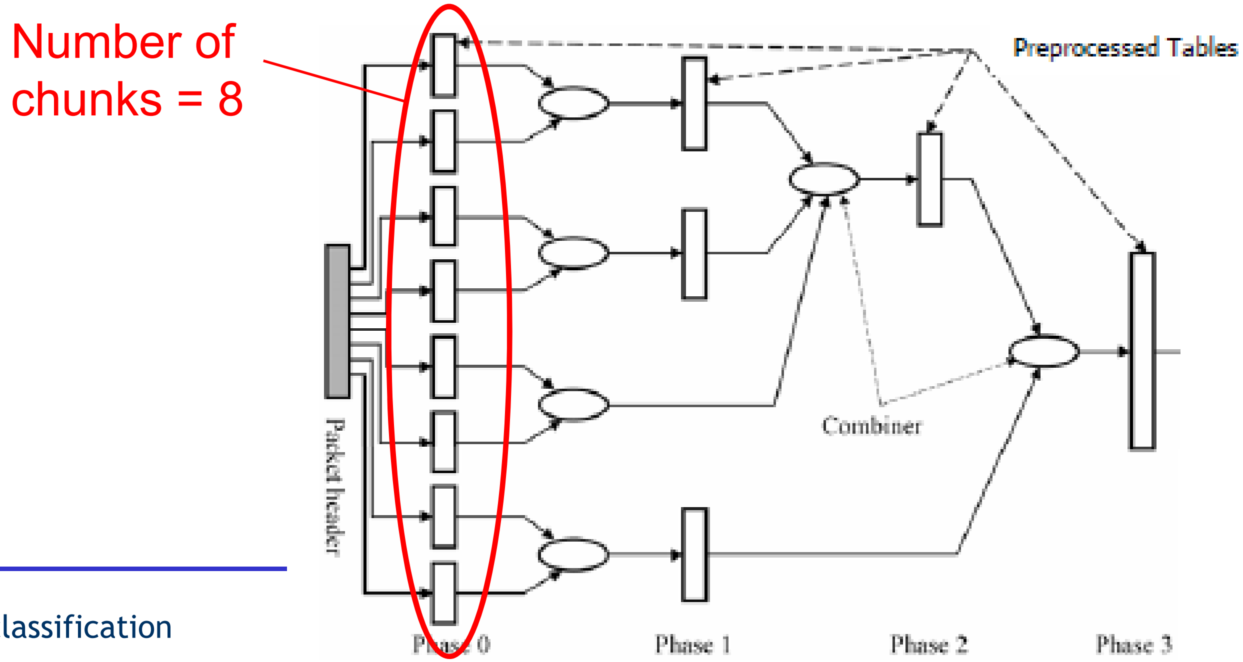


Recursive Flow Classification (RFC)

- A simple but impractical method is to pre-compute the action for each of the 2^H different packet headers and storing the results in an array $2^H \times 2^T$
  - Only one memory access is needed to yield the corresponding action. But this would require too much memory
- Main objective
  - Perform the same mapping but over several stages (phases)

Recursive Flow Classification (RFC)

- In the first step the d fields of the packet header are split into multiple chunks that are used to index multiple memories in parallel
- The mapping is performed recursively; at each stage, the algorithm performs a reduction, mapping one set of values to a smaller set
- In each phase, a set of memories returns a value shorter expressed in fewer bits, than the index of the memory access
- Results of a phase are combined to be used as input to the subsequent phase



Recursive Flow Classification (RFC)

- Structure and Packet flow in the recursive flow classification

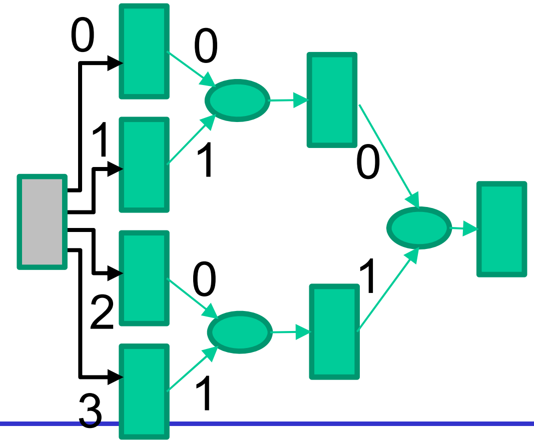
Destination IP (addr/mask)	Source IP (addr/mask)	Port Number	Protocol
152.163.190.69/0.0.0.0	152.163.80.11/0.0.0.0	*	*
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0	eq www	UDP
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0	range 20-21	UDP
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0	eq www	TCP
152.163.198.4/0.0.0.0	152.163.160.0/0.0.3.255	gt 1023	TCP
152.163.198.4/0.0.0.0	152.163.36.0/0.0.0.255	gt 1023	TCP

The construction of the preprocessed tables is explained with a sample rule set *C*.

Preprocessed Tables

gt = greater than = >

- **The first step of the preprocessed table construction**
  - Split the d fields of the packet header into multiple chunks that are used to index multiple memories in parallel



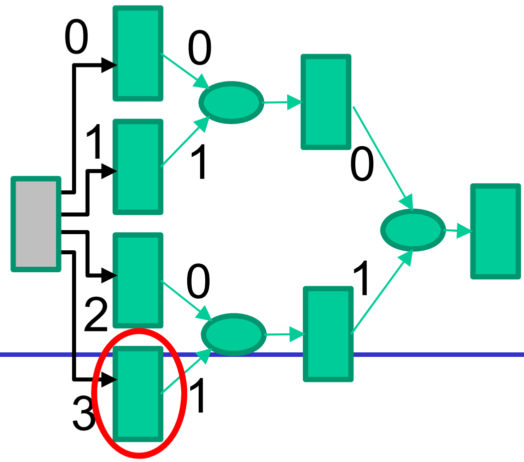
Chopping of packet header into chunks for rule set *R* in the first phase of RFC:

	Source IP	Destination IP	Port	Protocol
Size (bit)	32	32	16	8
Chunk #	0	1	2	3

Recursive Flow Classification (RFC)

- Each of the parallel lookups will map the chunk to an *eqID* according to the rules
  - Consider a fixed chunk of size *b* bits. Its mapping table is of 2^b entries and each entry contains an *eqID* for that chunk value
  - The eqIDs are determined by those component(s) of the rules in the classifier corresponding to this chunk
  - The term "Chunk Equivalence Set" (CES) is used to denote a set of chunk values that have the same *eqID*
    - e.g. for chunk #3 (protocol, 8 bits), the 'CES' will be:
      - (a) {TCP}
      - (b) {UDP}
      - (c) {all remaining numbers in the range 0–255}

Protocol
*
UDP
UDP
TCP
TCP
TCP



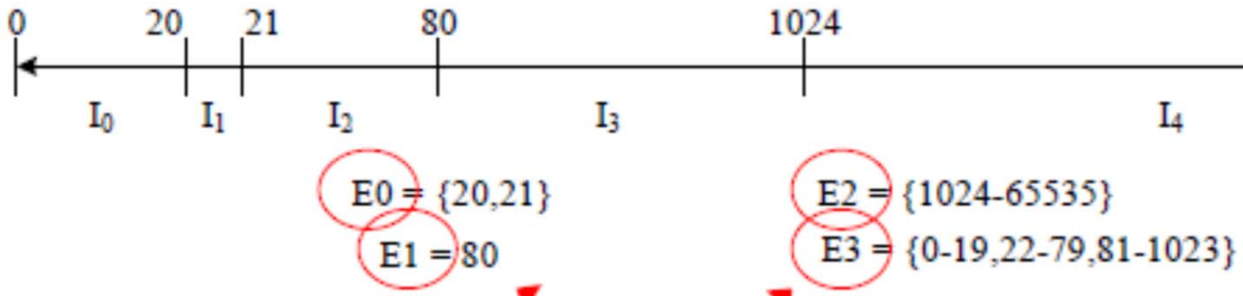
Recursive Flow Classification (RFC)

- Each CES can be constructed in the following manner:
  - For a *b*-bit chunk, project the rules in the classifier on to the number line $[0, 2^b\text{-}1]$
  - Each component projects to a set of intervals on the number line
  - The end points of all the intervals projected by these components form a set of non-overlapping intervals
    - Two points in the same interval always belong to the same equivalence set
    - Two intervals are also in the same equivalence set if exactly the same rules project onto them

Recursive Flow Classification (RFC)

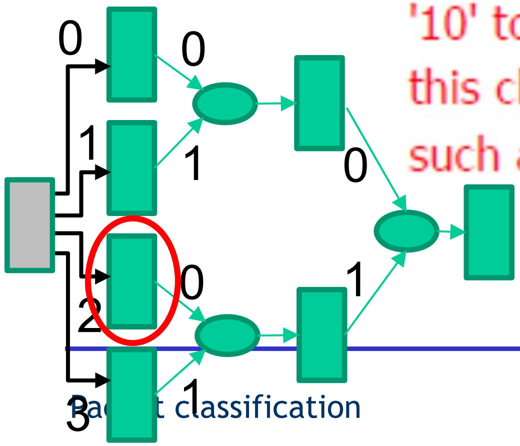
Example:

- **Computing the four equivalence classes E0…E3 for chunk #2 (corresponding to the 16-bit transport-layer destination port number) in the rule set**



Port Number
*
eq www
range 20-21
eq www
gt 1023
gt 1023

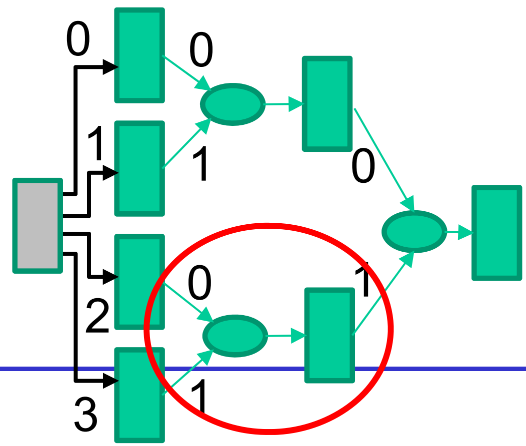
The four CESs can be decoded using two bits. For example we can assign '00' to E1, '01' to E0, '10' to E2 and '11' to E3. Then the RFC table for this chunk is filled with the corresponding *eqIDs*, such as *table*(20) =01', *table*(23) ='11', etc.



Well-known port number for http (www): 80

Recursive Flow Classification (RFC)

- A chunk is formed by a combination of two (or more) chunks obtained from memory lookups in previous steps, with a corresponding CES
  - If the resulting chunk is of width *b* bits, we again create equivalence sets
    - Two *b*-bit numbers that are not distinguished by the rules of the classifier belonging to the same CES
  - Compute all possible intersections of the equivalence sets from the previous steps being combined
    - Each distinct intersection is an equivalence set for the newly created chunk

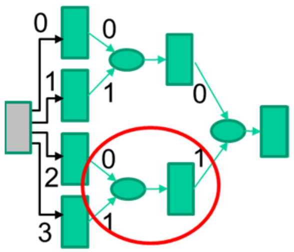


Recursive Flow Classification (RFC)

- Example
  - If we combine chunk #2 (port number) and #3 (protocol), then five CES scan be get:
    - {({80}, {UDP})
    - {({20-21}, {UDP})}
    - {({80}, {TCP})}
    - {({gt1023}, {TCP})}
    - {all the remaining crossproducts}

Destination IP (addr/mask)	Source IP (addr/mask)	Port Number	Protocol
152.163.190.69/0.0.0.0	152.163.80.11/0.0.0.0	*	*
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0	eq www	UDP
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0	range 20-21	UDP
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0	eq www	TCP
152.163.198.4/0.0.0.0	152.163.160.0/0.0.3.255	gt 1023	TCP
152.163.198.4/0.0.0.0	152.163.36.0/0.0.0.255	gt 1023	TCP

  - These can be expressed in a three-bit eqID, as shown in following Figure



CES Port	CES Prot	Port Number and Protocol	Class Number	eqID (only 3 bits required)
01	01	eq www & udp	1	000
00	01	Range 20-21 & udp	2	001
01	00	eq www & tcp	3	010
10	00	gt 1023 & tcp	4	011
....	....	all remaining crossproducts	5	100

  - We can see that, during step two the number of bits has been reduced from four (two bits for chunk #2 and #3 respectively after step one) to three
  - For the combination of the two steps, this number has dropped from 24 to 3

Recursive Flow Classification (RFC)

Port Number	Class Number	eqID (only 2 bits required)
Range 20-21	1	00
eq www	2	01
gt 1023	3	10
0-19,22-79,81-1023	4	11

(c) Port number field made into chunks and eqIDs

Protocol	Class Number	eqID (only 2 bits required)
tcp	1	00
udp	2	01
all remaining protocols	3	10

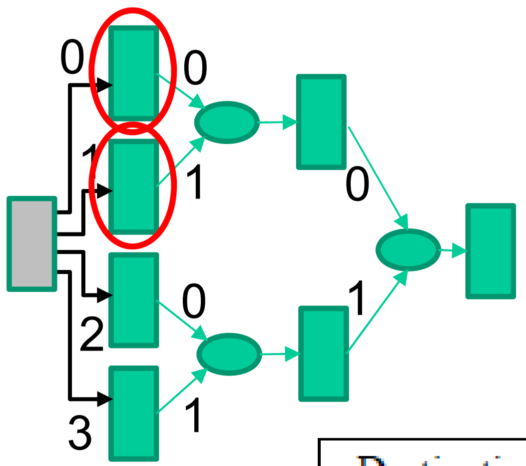
(d) Protocol field made into chunks and eqIDs

CES Port	CES Prot	Port Number and Protocol	Class Number	eqID (only 3 bits required)
01	01	eq www & udp	1	000
00	01	Range 20-21 & udp	2	001
01	00	eq www & tcp	3	010
10	00	gt 1023 & tcp	4	011
….	….	all remaining crossproducts	5	100

(e) Port number and protocol fields combined and made into chunks and eqIDs

Recursive Flow Classification (RFC)

- Rule storing organization for RFC for the rule set in Table



Destination IP (addr/mask)	Source IP (addr/mask)
152.163.190.69/0.0.0.0	152.163.80.11/0.0.0.0
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0
152.163.198.4/0.0.0.0	152.163.160.0/0.0.3.255
152.163.198.4/0.0.0.0	152.163.36.0/0.0.0.255

Note that the entries are not prefixes

Destination IP (addr/mask)	Class Number	eqID (only 2 bits required)
152.163.190.69/0.0.0.0	1	00
152.168.3.0/0.0.0.255	2	01
152.163.198.4/0.0.0.0	3	10

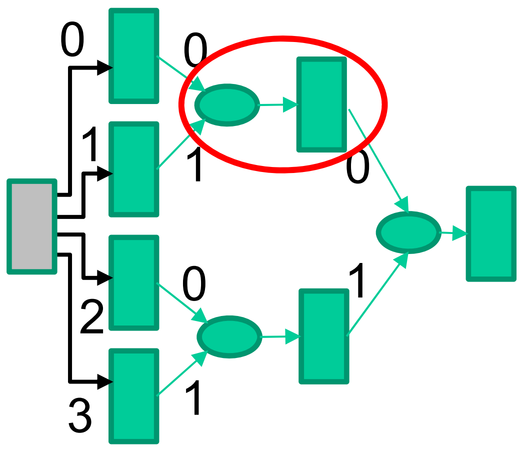
(a) Destination IP field made into chunks and eqIDs

Source IP (addr/mask)	Class Number	eqID (only 2 bits required)
152.163.80.11/0.0.0.0	1	00
152.163.200.157/0.0.0.0	2	01
152.163.160.0/0.0.3.255	3	10
152.163.36.0/0.0.0.255	4	11

(b) Source IP field made into chunks and eqIDs

Recursive Flow Classification (RFC)

- Rule storing organization for RFC for the rule set in Table



Destination IP (addr/mask)	Source IP (addr/mask)
152.163.190.69/0.0.0.0	152.163.80.11/0.0.0.0
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0
152.163.198.4/0.0.0.0	152.163.160.0/0.0.3.255
152.163.198.4/0.0.0.0	152.163.36.0/0.0.0.255

CES D-IP	CES S-IP	CES S-IP + D-IP (3 bit)
00	00	000
01	01	001
10	10	010
10	11	011
….	….	100

Recursive Flow Classification (RFC)

Classification scheme

- First split into several chunks to be used as an index
- Then the required *eqIDs* are combined into chunks of the second phase
- This procedure goes on until the final phase is reached
  - When all the remaining *eqIDs* have been combined into only one chunk
- The corresponding table will hold the actions for that packet

Recursive Flow Classification (RFC)

Performance

- The storage complexity
  - Different combination of the chunks can yield different storage requirement
  - With real-life 4D classifiers of up to 1700 rules
    - RFC appears practical for 10-Gbps line rates in hardware and 2.5-Gbps rates in software
    - The storage space and preprocessing time grow rapidly for classifiers larger than 6000 rules
  - An optimization reduces the storage requirement of a 15,000 four-field classifier to below 4 Mbytes

Packet classification

- Trie-based classification
  - Hierarchical trie
- Geometric algorithms
  - Cross-producting scheme
  - Bitmap intersection
- <u>TCAM-based algorithms</u>

TCAM-Based Algorithms

- Ternary content addressable memory (TCAM) → increasing popularity for fast packet classification
- TCAM coprocessor works as a lookaside processor on behalf of a network processor
  - The processor generates a search key based on the information from the packet header
  - Passes it to the TCAM coprocessor for classification via a NPU/TCAM coprocessor interface
  - Each entry of the TCAM stores a rule by concatenating the prefixes of all fields
  - TCAM coprocessor finds a matched rule in O(1) clock cycles
    - Highest possible lookup/matching performance
- A local CPU is in charge of rule table update through a separate CPU-TCAM coprocessor interface

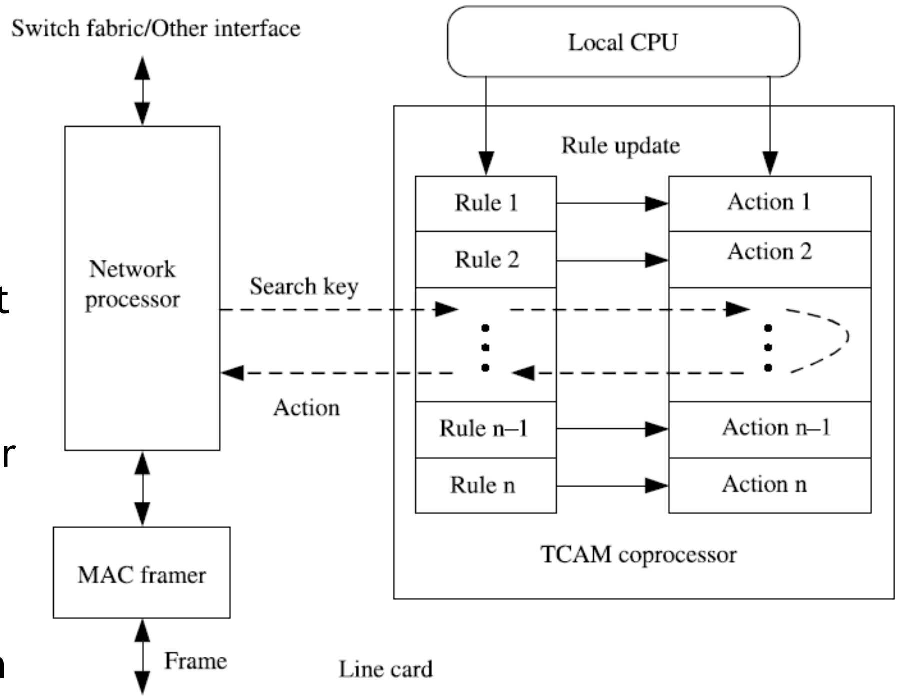


Figure 3.25 Network processor and its TCAM coprocessor.

TCAM-Based Algorithms

Range splitting

- TCAMs have been widely used in IP route lookup for longest prefix matching,
  - Rules are stored as (*val*,*mask*) pairs in decreasing order of prefix lengths
- For range matching, utilization is more critical
- <u>Range splitting</u> must be performed to convert the ranges into prefix formats to fit the bit boundary
  - Increases the number of entries by a factor of O(W^d) in the worst case
  - A range is said to be exactly implemented in a TCAM if it is expressed in a TCAM without being encoded
    - E.g. six rule entries are needed to express a range {>1023} = {> 0000 0011 1111 1111}

0000 01** **** ****	0000 1*** **** ****
0001 **** **** ****	001* **** **** ****
01** **** **** ****	1*** **** **** ****

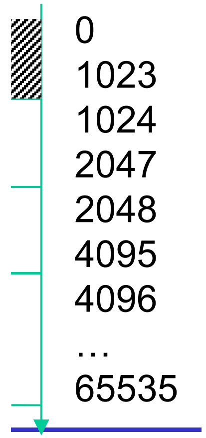


Figure 3.26 Range >1023 is expressed in terms of 6 sub-ranges in a TCAM.

TCAM-Based Algorithms

Rule / key encoding

- Reduced TCAM memory efficiency due to range matching (up to even 16%) → power consumption, footprint, and cost serious concern

- Solution:
  Range preprocessing/encoding by mapping ranges to a short sequence of encoded bits, known as bit-mapping
- Rule encoding (pre-processed in software)
  - view a d-tuple rule as a region in a d-dimensional rule space
  - encode any distinct overlapped regions among all the rules
  - each rule can be translated into a sequence of encoded bits
- Search key encoding (performed on a per packet basis in hardware at wire-speed)
  - the information is extracted from the packet header
  - a search key is preprocessed to be encoded
- The encoded search key is matched against all the encoded rules to find the best matched rule

- An efficient mapping scheme of range classifier into TCAM expands TCAM horizontally (using more bits per entry)
- For width limited application, another algorithm allows both horizontal and vertical expansion

- Rule storing organization: For each range field, an n bits vector $B = \{b_1, b_2, \ldots, b_n\}$ is used to represent it
  - n is the number of distinct ranges specified for this field
  - The B vector for a range E_i has 1 at bit position i, i.e., $b_i = 1$ and all other bits are set to don't care
    - The number of distinct ranges specified for any range field is very limited
    - Exact match specification (n = 1) also happens frequently for a range field

TCAM-Based Algorithms
Rule encoding

- Example

R_i	Dest IP Addr (IP/mask)	Dest Port Range	Action
1	10.0.0.0/255.0.0.0	> 1023	Deny
2	192.168.0.0/255.255.0.0	50–2000	Allow
3	192.169.0.0/255.255.0.0	80(http)	High Priority
4	172.16.0.0/255.255.0.0	23(telnet)	Route through Port A
5	172.16.0.0/255.255.0.0	21(ftp)	Rate Limit to 1Mbit/s

The range of greater than 1023 in $R1$ is represented By 'xxxx1'.

The bit vector representation (n=5 in this case).

R_i	TCAM rules	
1	10.x.x.x	xxxx1
2	192.168.x.x	xxx1x
3	192.169.x.x	xx1xx
4	172.16.x.x	x1xxx
5	172.16.x.x	1xxxx

- Classification Scheme
  - A lookup key $v \in \left[0,2^k\right]$ is translated into an n bit vector V = {v_1, v_2, …, v_n}
  - Bit v_i is set to 1 if the key v falls into the corresponding range E_i, otherwise it will be set to 0
- Lookup key translation could be implemented as a direct memory lookup since most of the range fields are less than 16-bit wide

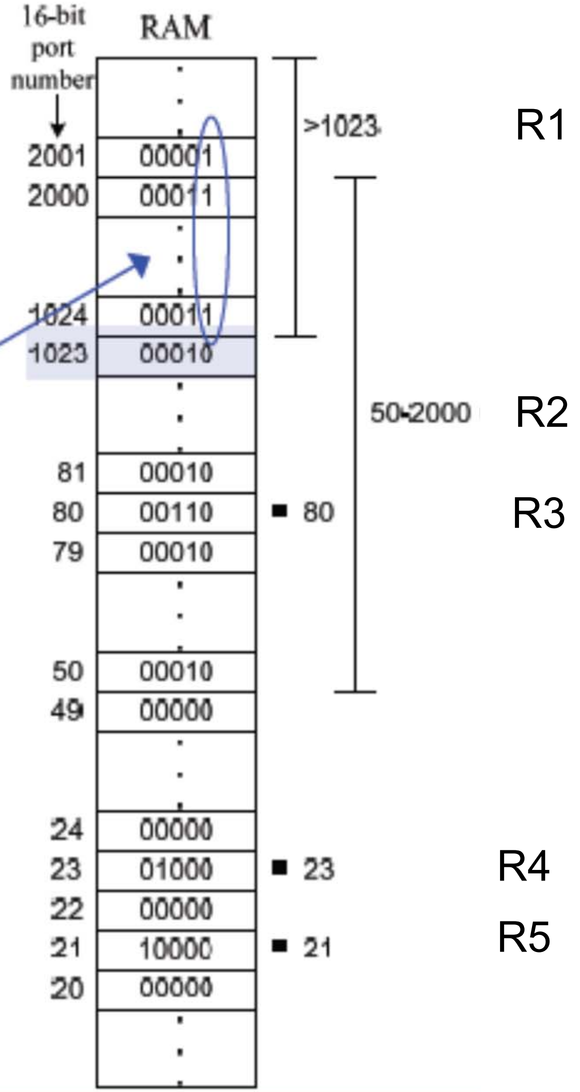
TCAM-Based Algorithms

Lookup key translation table

A complete lookup key translation table for the example classifier for each possible lookup key value.

For example, the most right of the bit vector is set to 1 for all the locations above 1023.

Second bit from the right between the locations 50 and 2000 (including them) is also set to 1 … etc.



R_i	Dest Port Range
1	> 1023
2	50–2000
3	80(http)
4	23(telnet)
5	21(ftp)

1	xxxx1
2	xxx1x
3	xx1xx
4	x1xxx
5	1xxxx

TCAM-Based Algorithms

Exact matching optimization

- It is possible to reduce the number of bits used to $\log_2(m + 1)$, where m is the number of exact matches
- The bit representation will contain two parts $<B_e, B>$: B_e for exact matches, and B for all others
  - $B_e = \{b_1, b_2, \ldots, b_t\}$ is a t bit vector, where $t \geq \log_2(m + 1)$ and m is the number of exact matches
  - For a normal range, $<B_e=0, B>$ and its B portion is the same as before
  - For an exact match, $<B_e=i, B = 0>$, if it is the i-th exact match

- Example
  - If we use bit 2 and 3 as B_e (where bit 1 is the left most significant bit), the resulting classifier stored in TCAM will as shown on the right side table
  - only two bits are needed to represent the three distinct exact matches

R_i	TCAM rules	
1	10.x.x.x	xxxx1
2	192.168.x.x	xxx1x
3	192.169.x.x	xx1xx
4	172.16.x.x	x1xxx
5	172.16.x.x	1xxxx

R_i	TCAM rules	
1	10.x.x.x	xxxx1
2	192.168.x.x	xxx1x
3	192.169.x.x	x01xx
4	172.16.x.x	x10xx
5	172.16.x.x	x11xx

TCAM-Based Algorithms

Exact matching optimization

- The lookup key translation table needs to be changed accordingly
- It also contains two parts <V_e,V> V_e corresponding to all exact matches and V to the rest
  - V_e= {v_1, v_2, …, v_t} is a *t* bit vector
  - V_e= I if the lookup key v equals to the *i*-th exact match, otherwise V_e=0

Lookup example

192.169.10.1 / 80

$\langle V_e, V\rangle = \langle\{01\},10\rangle$

R_i	TCAM rules	
1	10.x.x.x	xxxx1
2	192.168.x.x	xxx1x
3	192.169.x.x	x01xx
4	172.16.x.x	x10xx
5	172.16.x.x	x11xx

R3

- A new packet arrives with destination IP address 192.169.10.1 and port number 80
  - The port number is indexed into the lookup key translation table
  - The resulting V = 10 because 80 falls in range 50-2000 but not range >1023
  - The resulting V_e= 01 because 01 is the value assigned to exact match value of 80
  - Together with destination IP address, the final result is rule 3

TCAM-Based Algorithms

Performance

- The storage complexity
  - This scheme requires less TCAM storage space
  - It can accommodate a much larger number of rules in a single TCAM table and reduce system cost and power consumption
- Adding/Deleting rules cause the changes of bit vectors and may require re-computation of the entire translation table, a time consuming process