Packet-Switching

Decision Logic in Routers

Outline

- Introduction
- Packet Parsing
- Types of decision logic
  - Ethernet switching, IP routing, Access Control Lists, Generic Table Matching
- Decision Pipeline
  - Fixed or Programmable
- Implementations
  - CAMs,TCAMs
  - Data structures (cuckoo hash and tries)

Introduction

- The decision logic analyzes the packets and based on a set of rules decides what actions to take on each packet
- In most cases, the process can be structured as a sequence of table lookups
- The first step is to parse the packet headers to extract the relevant fields that are used for decisions
- Decisions have to be made at wire speed and in some cases against large sets of rules

Packet Parsing

- The first step is to extract the relevant information from the packet
- This is then used for the rest of the process
- For example, in Ethernet switching we only need to extract the destination MAC address that will be used to lookup the switching table
- In one of the first versions of Openflow 12 header fields are extracted
- A flexible parser is needed to support a flexible decision logic

Table Lookup

- Once the relevant information has been extracted, it is used as a key to find the best matching rule to process the packet
- Traditional examples of table lookups are
  - Ethernet switching
  - IP routing
  - ACLs
- They can be generalized into a table lookup operation on arbitrary packet header fields

Ethernet switching

- The key is the destination MAC address (48 bits)
- The switching table contains a list of the known destination MAC addresses
- A lookup up is done to check if the MAC address of the packet is stored in the table
- If it is found, the action (outgoing port) is retrieved and the packet is sent to that port
- The check is an <u>exact match </u>bit by bit
- The number of entries is generally not too high, less than 100,000

Ethernet switching

	VLAN ID	MAC address	Out Port/Action
0	25	2c:44:fd:d0:e4:b8	5
1	31	38:72:c0:73:4c:ab	3
2	-	4:e5:36:c6:2a:ed	1
3	-	01:00:5e:01:02:03	action23
4	31	8c:15:c7:8f:a3:57	action36
5			
6			
7			

Match type: exact match

IP routing

- The key is the destination IP address (32 bits in IPv4 and 128 in IPv6)
- The routing table contains a list of the IP prefixes for which a destination is known
- A lookup up is done to find the Longest Prefix Match
- If it is found, the action (outgoing port, [next-hop]) is retrieved and the packet is sent to that port after decrementing the TTL
- The check is no longer an exact match as prefixes have wildcard bits

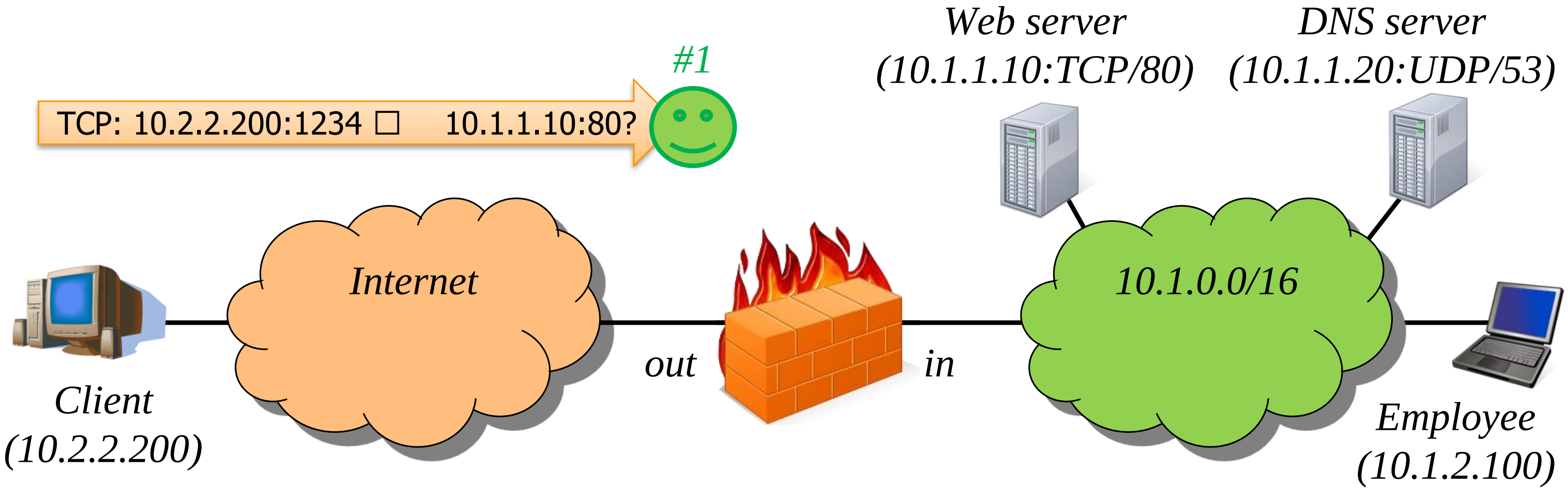
IP forwarding

- A simple routing table
  - 138.100.17.0/24 -> Port 1
  - 138.100.0.0/16 -> Port 2
- A packet to 138.100.17.10 matches both routes, the first one is selected as it has a longer prefix (24 vs 16)
- Modern Internet routing tables have more than 800,000 entries for IPv4 and keep growing
- Can be challenging to check packets at wire speed

Access Control List

- Several header fields are checked to allow/deny the packet to traverse the device
- A typical case is to use the 5-tuple
  - Source IP address
  - Source L4 port
  - Destination IP address
  - Destination L4 port
  - Protocol
- We can have wildcards on several fields
- Also flags and port ranges may be part of the key

Access Control List



#	Iface	Src addr	Dst addr	Proto	Src port	Dst port	Action
1	out	ANY	10.1.1.10	TCP	ANY	80	Allow
2	out	ANY	10.1.1.20	UDP	ANY	53	Allow
3	in	10.1.0.0/16	ANY	ANY	ANY	ANY	Allow
4	ANY	ANY	ANY	ANY	ANY	ANY	Deny

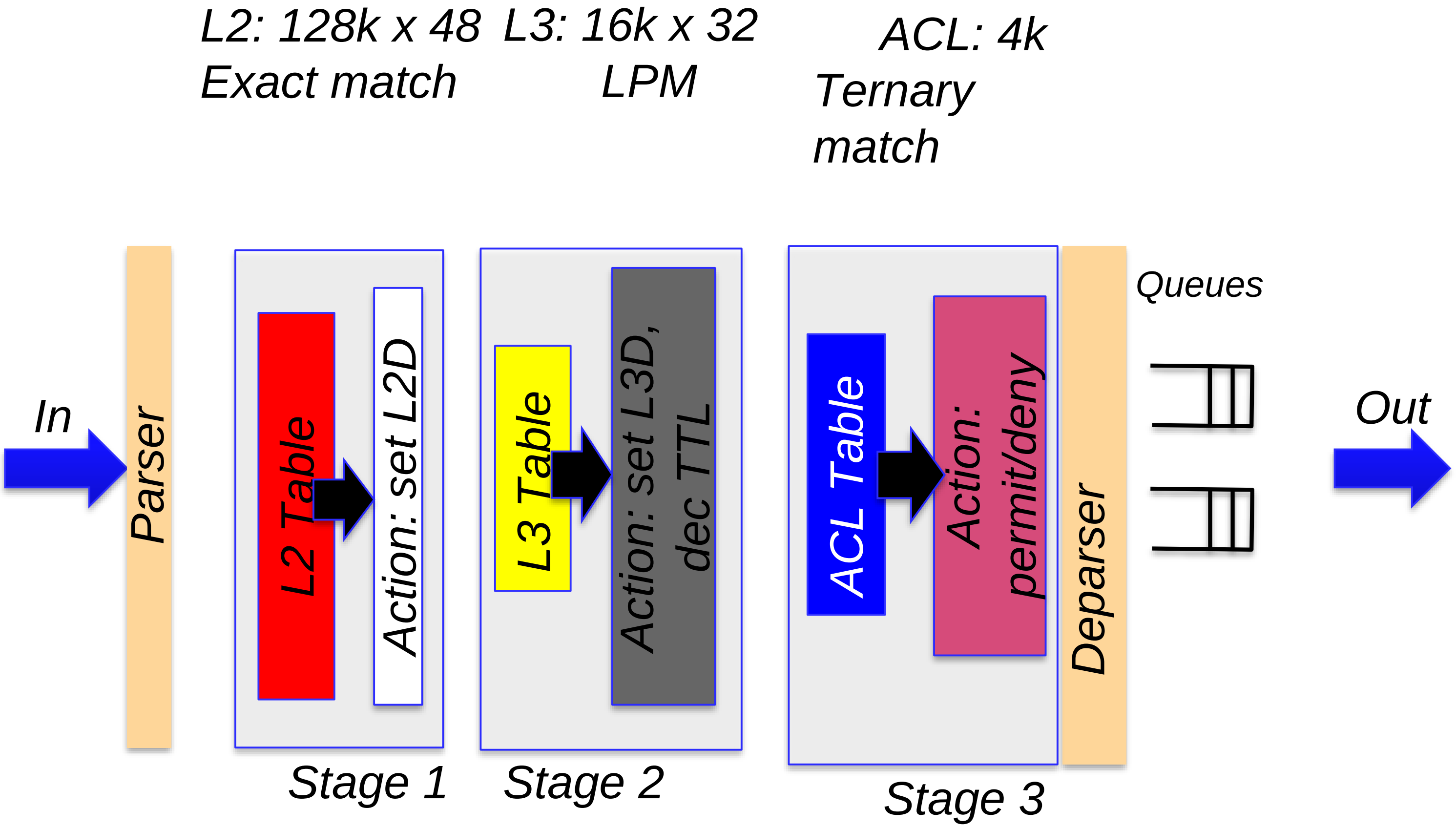
Generic Match

- Extract a set of fields from the packet header
- Implement matching, possibly against rules with wildcard bits
- This is one of the key abstractions for Software Defined Networks (SDN)
- Some examples are Openflow tables or the P4 language to describe packet processing
- Adding fields to the key can lead to very large keys

Decision Pipeline

- In most cases, a router performs a sequence of table lookups on a packet
- This is known as the decision pipeline
- For example, we can apply first an ACL and then if the packet is allowed perform IP route lookup
- The pipeline can be fixed or configurable so that the number and types of tables can be changed

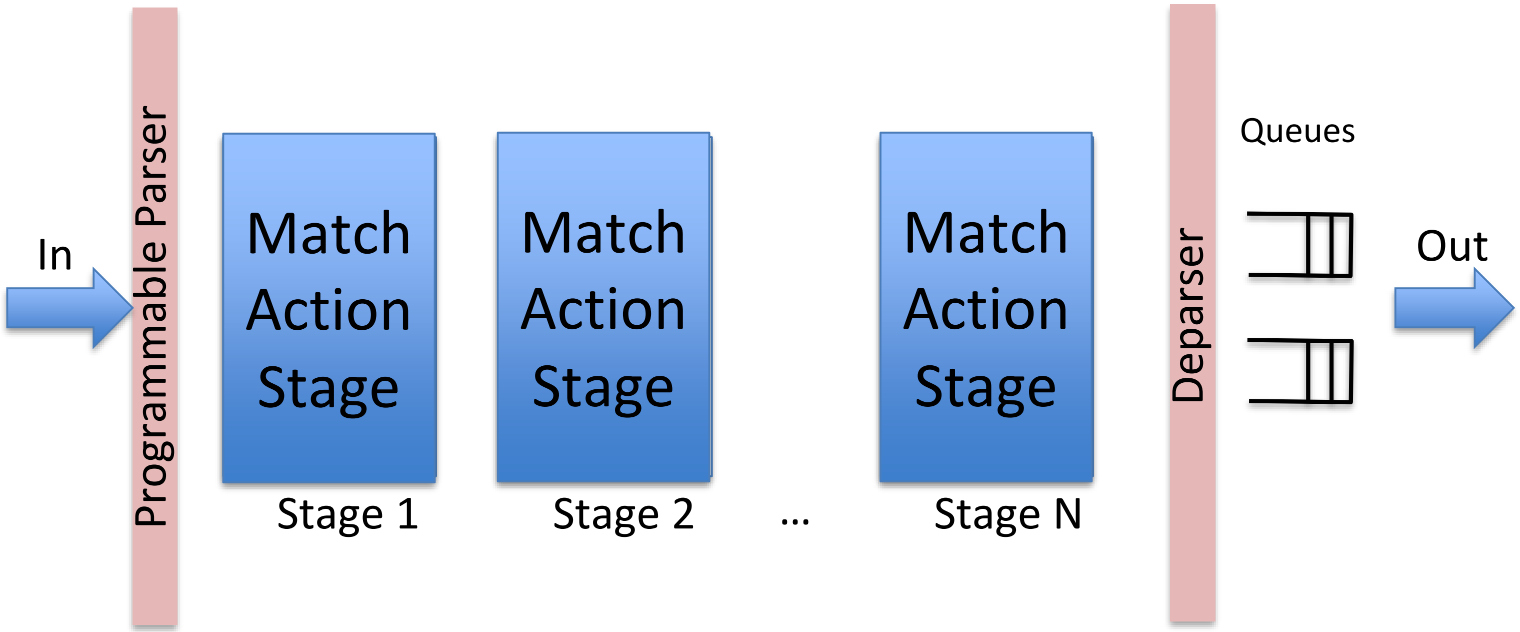
Fixed pipeline



Decision Pipeline (2)

- The decision pipeline can be modeled as a set of generic table match operations
- Then, the pipeline can be programmed to implement very flexible processing of packets
- This is the approach used in Software Defined Networking (SDN)

Programmable Pipeline



Implementation

- The decision logic can be implemented differently depending on the requirements:
  - Number of packets per second to process
  - Size of the tables
  - Need to reconfigure the number and sizes of tables
- For a single table, the main options:
  - Content Addressable Memory (CAM) for exact match
  - Ternary Content Addressable Memory (TCAM) if there are wild card bits
  - Standard memories combined with a data structure

CAMs/TCAMs

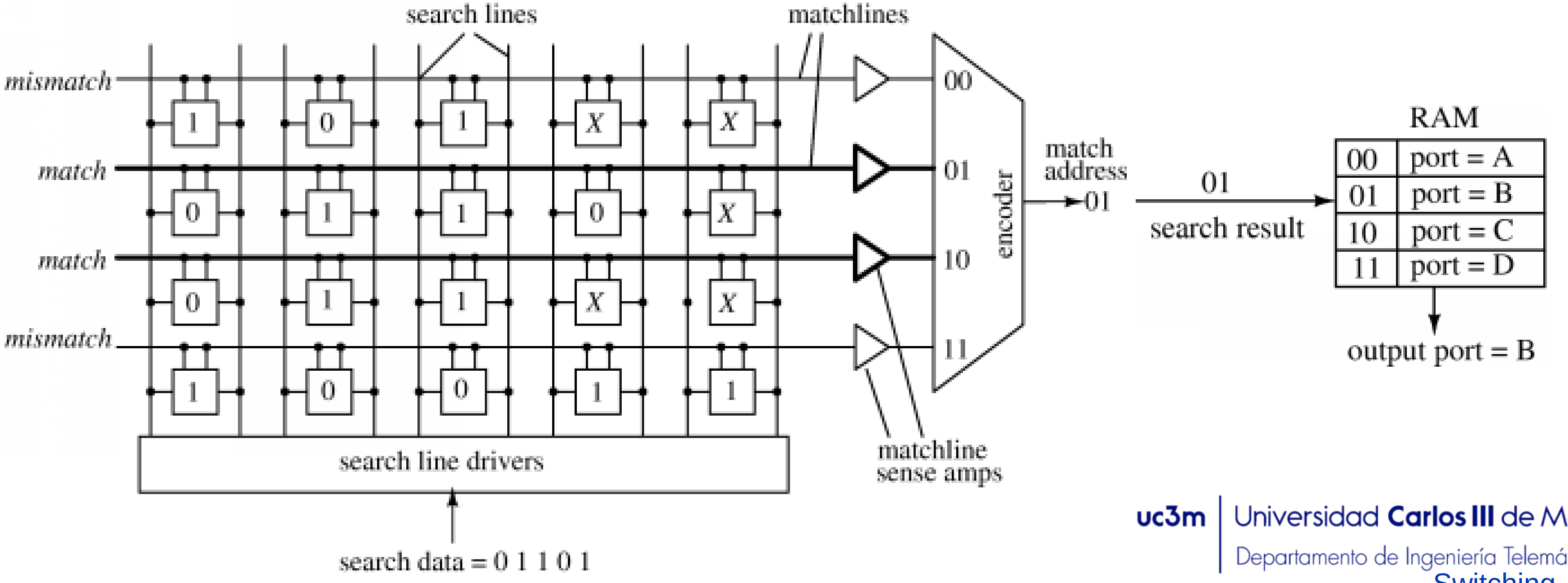
- Special type of memory that compares the incoming key with the values stored

- Two main types
  - Binary: support only ‘0’ and ‘1’ bits
  - Ternary: support ‘0’,‘1’ and wildcard ‘x’ bits

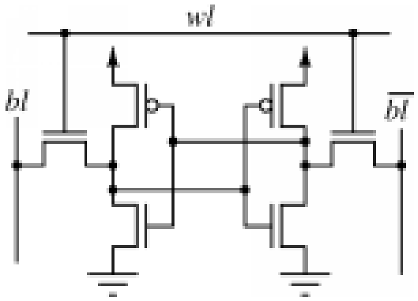
- Wildcards are used in LPM and ACL rules

CAMs/TCAMs

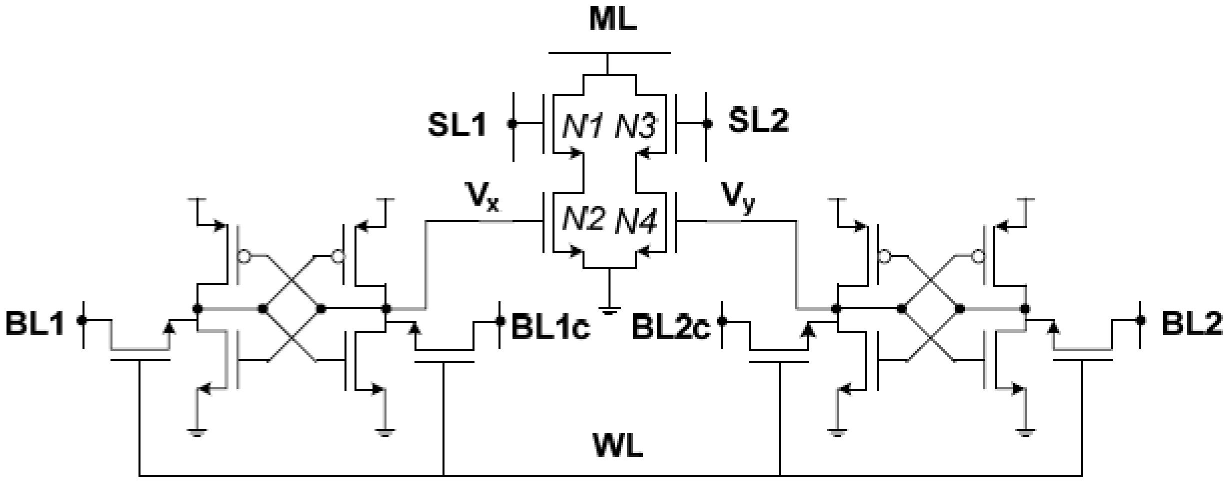
Tabla de encaminamiento simplificada		
No. línea	Dirección (binaria)	Puerto de salida
1	101XX	A
2	0110X	B
3	011XX	C
4	10011	D



CAMs/TCAMs



Conventional SRAM cell



TCAM cell

CAMs/TCAMs

- **They provide the result in one memory access**
- **Large area and power compared to SRAM**
- **Power can be optimized by segmenting the memory**
- **Not available in some platforms like general purpose computers or FPGAs**
- **Limited ability to support large tables**
- **Limited flexibility as they cannot be reused to store data/actions**

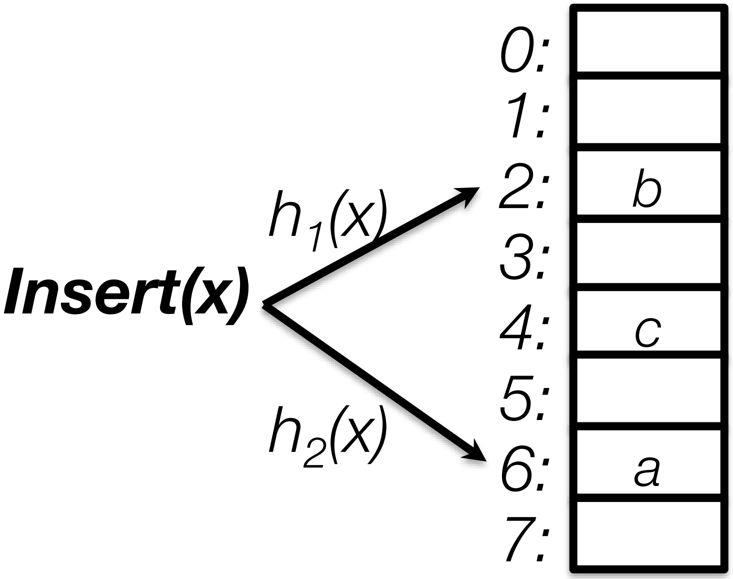
Data structures

- Many of them based on using hashing over the key
- They can use standard SRAM or DRAM memories
- The same memory can be reused for other purposes if needed
- The data structure to use depends on the underlying hardware:
  - Processor
  - FPGA/ASIC

Exact Match

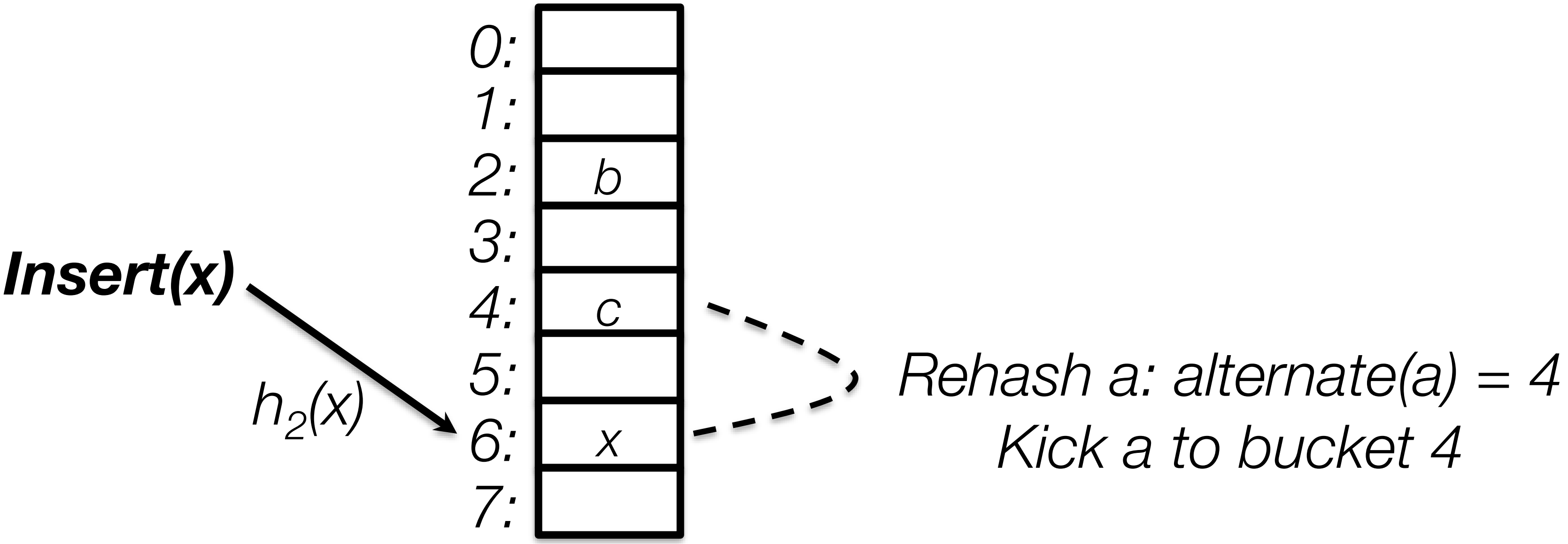
- Hashing with separate chaining can be used but collisions force us to perform several memory accesses per lookup
- The number of accesses depends on the occupancy and is not constant
- In the last decade, cuckoo hashing has been widely adopted as it provides a constant and known worst-case number of memory accesses
- Cuckoo hashing currently used in Hardware and Software implementations

Cuckoo Hash



R. Pagh and F. F. Rodler, "Cuckoo Hashing", Journal of Algorithms, pages 122–144. Elseiver, 2004.
https://www.sciencedirect.com/science/article/pii/S0196677403001925

Basic idea



Insert(x)

$h_2(x)$

0:

1:

2: b

3:

4: c

5:

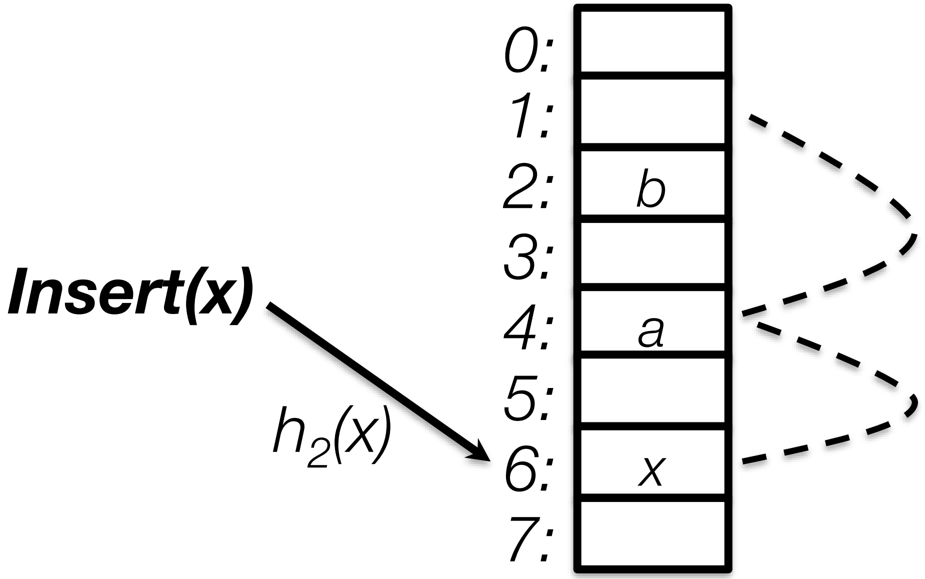
6: x

7:

Rehash a: alternate(a) = 4

Kick a to bucket 4

Basic idea



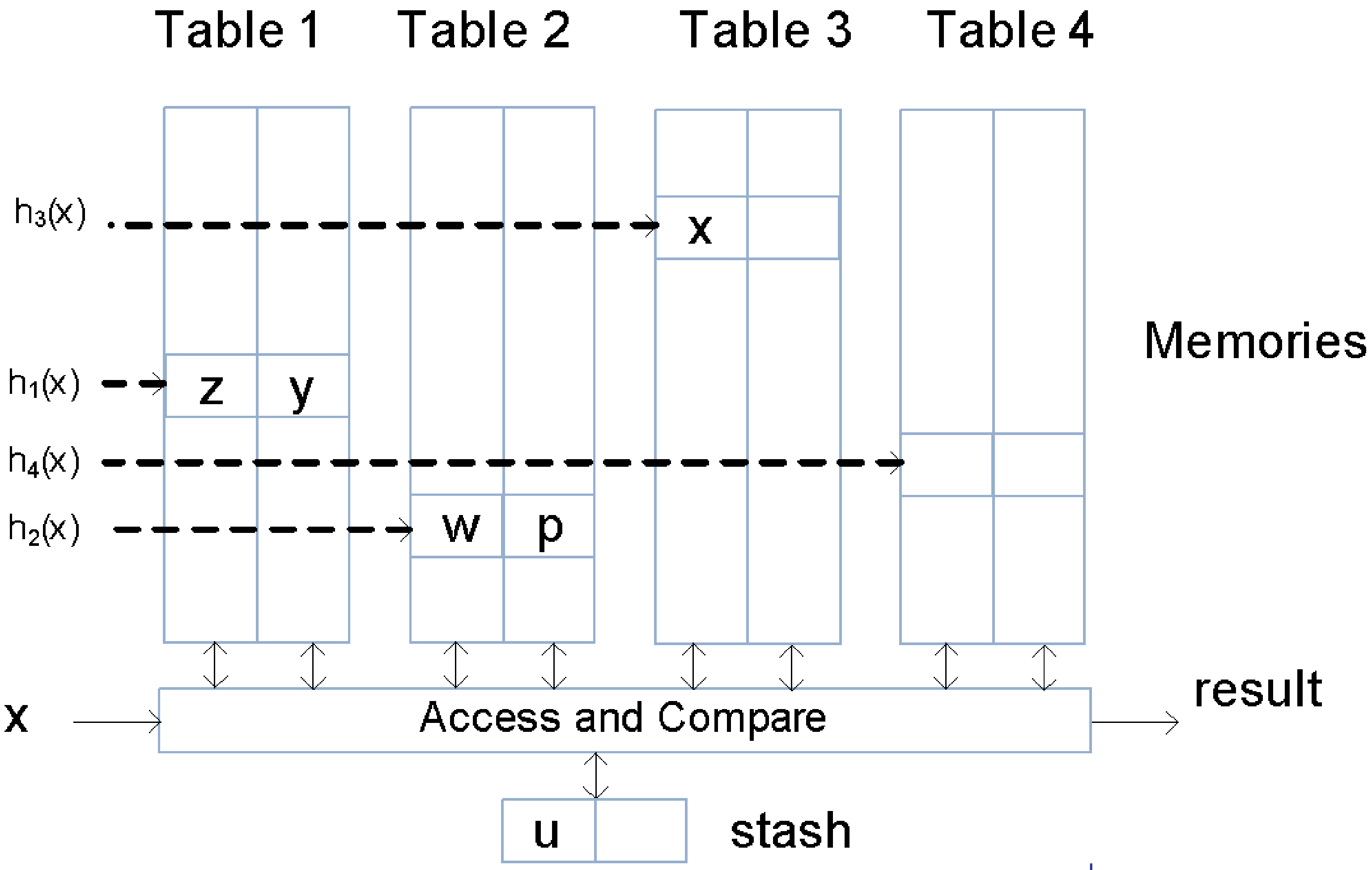
Rehash c: alternate(c) = 1
Kick c to bucket 1

Rehash a: alternate(a) = 4
Kick a to bucket 4

General Implementation

- Use one table per hash function and more than two tables
- Store several elements on each bucket
- This provides more choices for movements and thus achieves better occupancy
- But requires a larger number of accesses and memory bandwidth
- In general, we have *d* tables with buckets that have *b* cells that can store one element each
- Use a small stash to temporarily store elements being inserted/moved

Example *d* = 4, *b* = 2

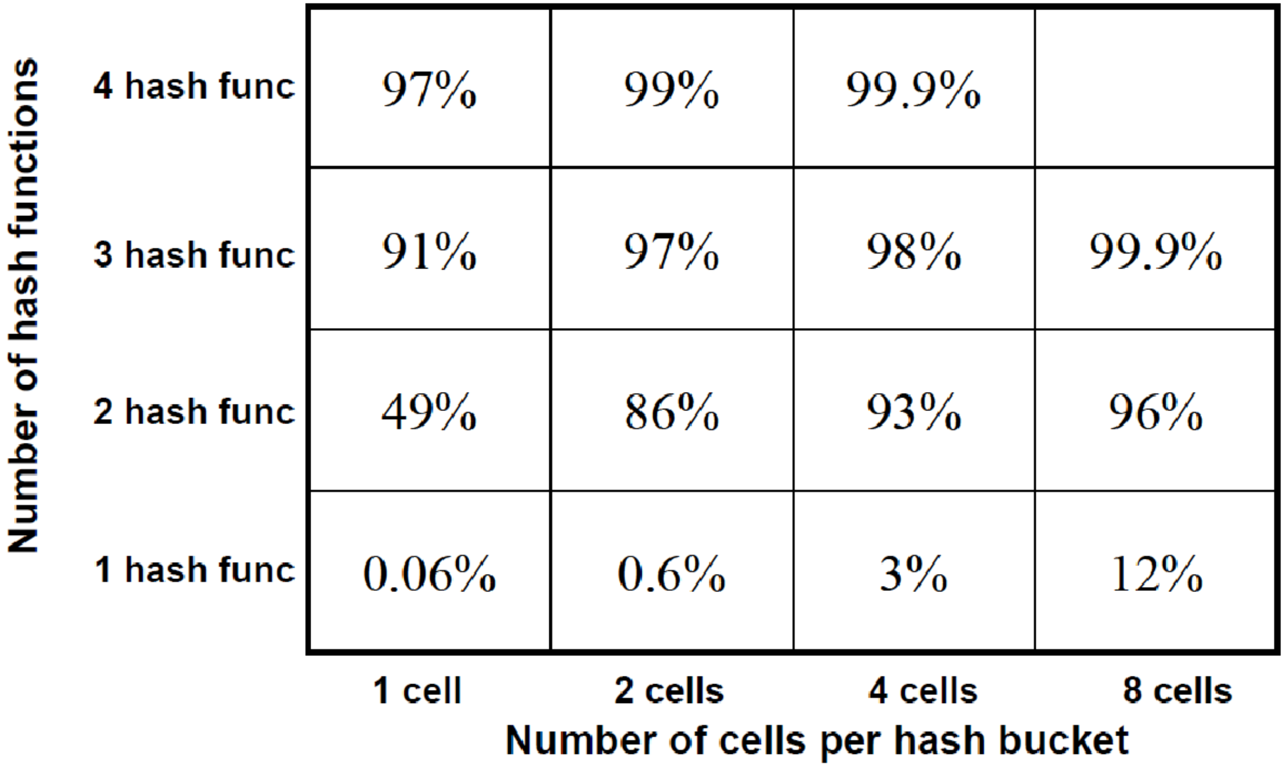


Performance

- Lookups
  - At most *d* memory accesses for a serial implementation
  - One memory access if we use *d* memories
- Insertions
  - May require a large number of accesses if many elements need to be displaced
  - This is only likely when the tables are close to the maximum occupancy
  - The stash can be used to place elements temporarily

Performance

- **Achievable occupancy**



Number of hash functions

	1 cell	2 cells	4 cells	8 cells
4 hash func	97%	99%	99.9%	
3 hash func	91%	97%	98%	99.9%
2 hash func	49%	86%	93%	96%
1 hash func	0.06%	0.6%	3%	12%

Number of cells per hash bucket

U. Erlingsson et al, ``A cool and practical alternative to traditional hash tables,'' in Proceedings of the Seventh Workshop on Distributed Data and Structures (WDAS), 2006. https://www.ru.is/faculty/ulfar/CuckooHash.pdf

Common Configurations

- **Parallel implementations**
  - ***d* = 4, *b* = 1,2**
  - ***d* memories**
  - **Used in HW implementations (ASICs and FPGAs)**

- **Sequential implementations**
  - ***d* = 2, *b* = 4,8**
  - **Single memory**
  - **Used in SW implementations**

Implementations

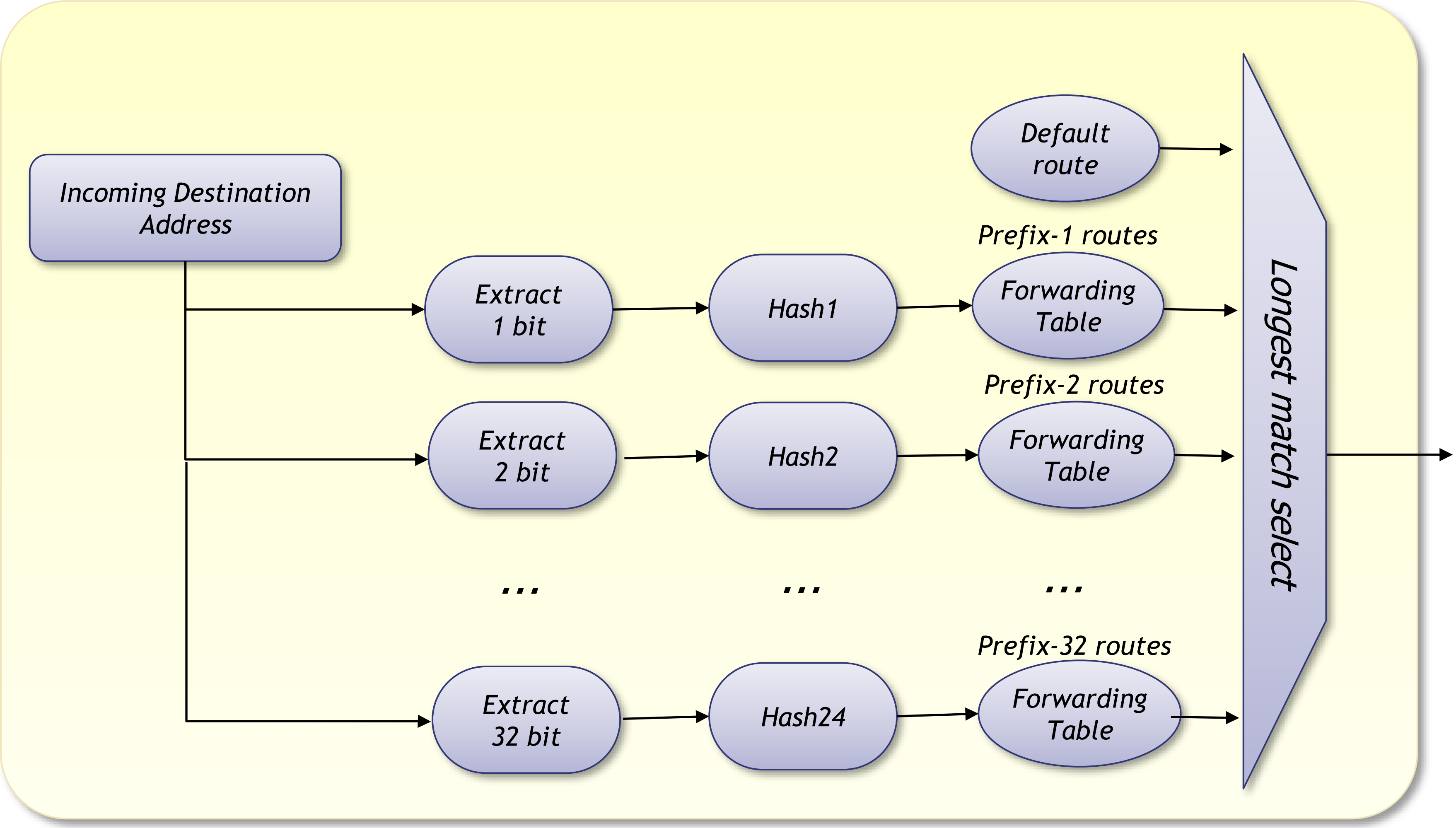
- Cuckoo hashing implementations
  - D. Zhou et al, “Scalable, High Performance Ethernet Forwarding with CuckooSwitch”, CONEXT 2013. https://www.cs.cmu.edu/~binfan/papers/conext13_cuckooswitch.pdf
  - N. Le Scouarnec, “ Cuckoo++ hash tables: high-performance hash tables for networking applications”, ANCS 2018. https://dl.acm.org/citation.cfm?doid=3230718.3232629
  - P. Bosshart et al, “Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN”, Sigcomm 2013. http://yuba.stanford.edu/~grg/docs/sdn-chip-sigcomm-2013.pdf
  - G. Levy et al, “Flexible Packet Matching with Single Double Cuckoo Hash”, IEEE Communications Magazine, June 2017.https://ieeexplore.ieee.org/document/7934182

LPM

- **Implement LPM as a sequence of exact match lookups**
    - ➢Notice that once the LPM (or any other complex processing) has been performed on the first packet of a flow the result (the action) can be cached in a cuckoo hash table: "route caching" .
- **One option is to use a binary search on prefix lengths[1]**
    - ❖ **This requires in the worst case five lookups for IPv4 and seven for IPv6**
    - ❖ **Enables the use of SRAM/DRAM to store the prefixes**

[1] M. Waldvogel et al, "Scalable High Speed IP Routing Lookups", Sigcomm 1997. https://kops.uni-konstanz.de/bitstream/handle/123456789/6014/scalable_high_speed_IP_Routing_table_lookups.pdf

Naive implementation



Binary search on length

Routing Table

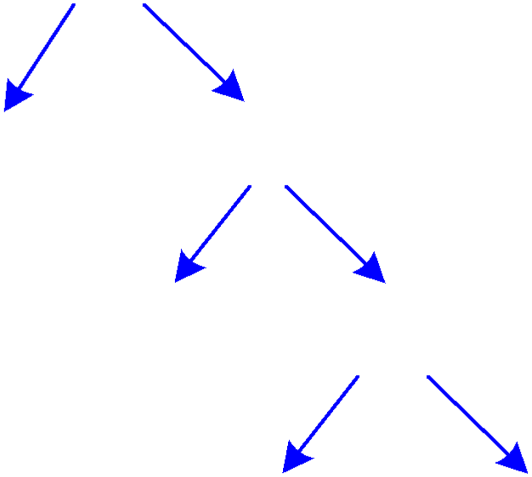
R1 139/8

R2 138.100/16

R3 138.100.17.128/26

R4 138.100.17.10/32

R5 139.23.100.43/32



M. Waldvogel et al, "Scalable High Speed IP Routing Lookups", Sigcomm 1997.
https://kops.uni-konstanz.de/bitstream/handle/123456789/6014/scalable_high_speed_IP_Routing_table_lookups.pdf

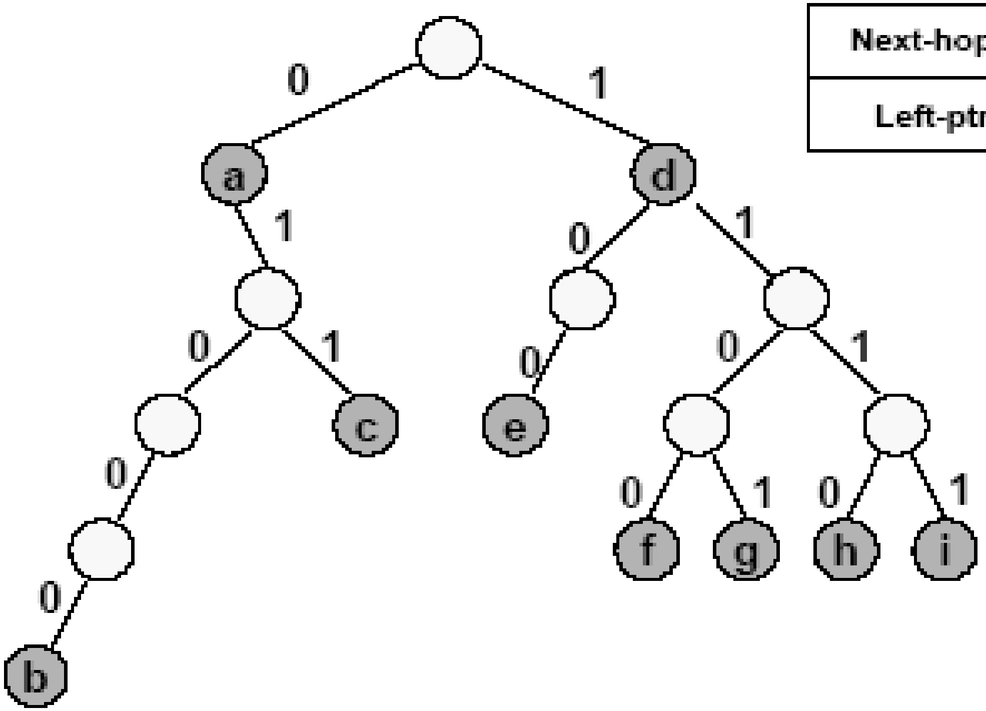
Tries

- *A trie is a tree-based structure allowing to organize prefixes on a digital basis by using the bits of prefixes to direct the branching*
- Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree implicitly denotes the key. The term trie comes from "retrieval".
- In a trie, a node on level k represents the set of all addresses that begin with the same k bits that label the path from the root to that node:
  - node c in the figure on the next slide is at level 3 and represents all addresses beginning with the sequence 011
- Nodes that correspond to prefixes are shown in a darker shade - these nodes contain the forwarding information or a pointer to it
- Some addresses may match several prefixes: addresses beginning with 011 will match prefixes c and a
  - prefix c is preferred because it is more specific (longest match rule)

Example of a binary Trie

Prefixes:

a	- 0*
b	- 01000*
c	- 011*
d	- 1*
e	- 100*
f	- 1100*
g	- 1101*
h	- 1110*
I	- 1111*



Information stored by a node:

Next-hop pointer (if prefix)	
Left-ptr	Right-ptr

LPM based on Tries

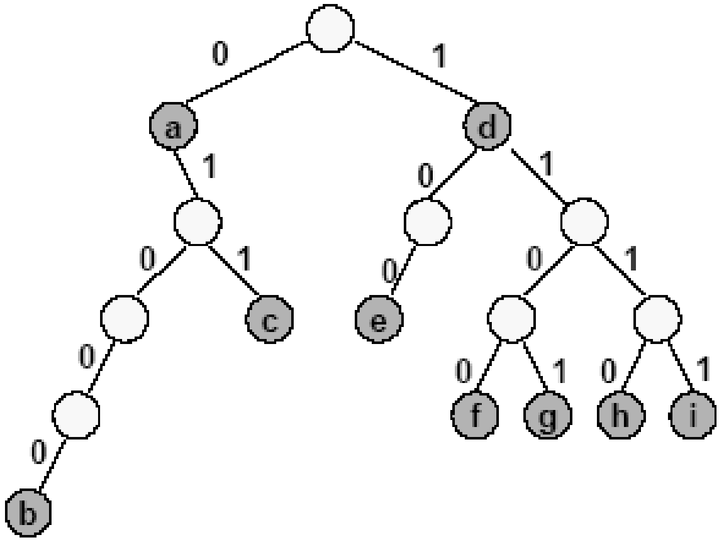
- **Tries allow finding the longest prefix that matches a given destination address and the search is guided by the bits of the destination address**
- **While traversing the trie and visiting a node marked as a prefix, this prefix is marked as the longest match found so far**
- **The search ends when no more branches can be followed in the trie. The longest match is the last visited prefix node.**

Example

- **An example: address 10110**
  - **from root move to the right (1st bit value = 1) to node d marked as a prefix, i.e. 1st prefix found is 1***
  - **then move to the left (2nd bit value = 0) to a node not marked as a prefix => prefix d still valid**
  - **3rd address bit = 1, but at this point there is no branch to the right.**
  - **=> search stops**
  - **=> d is the last visited prefix node and prefix of d is the longest match**

Prefixes:

a	- 0*
b	- 01000*
c	- 011*
d	- 1*
e	- 100*
f	- 1100*
g	- 1101*
h	- 1110*
I	- 1111*



LPM based on Tries

- Going through a trie is a sequential prefix search by length when trying to find a better match
  - begin looking in the set of length-1 prefixes, located at level 1
  - then proceed in the set of length-2 prefixes at level 2,
  - then proceed to level 3 and so on
- While stepping through a trie, the search space reduces hierarchically
  - Each step reduces the set of potential prefixes. The search ends when this set is just 1.
- Update operations are straightforward
  - Inserting a new prefix proceeds as a normal search and
    - If we arrive to a node with no branch to take, insert the necessary node
    - If we reach the last bit of our prefix: insert next-hop
  - Deleting a prefix: perform a search and when the required node has been found, unmark it as prefix and, if it has no descendants, delete it and all single- branched non-prefix nodes up the trie

Path Compressed Tries

- In binary tries, long sequences of one-child nodes may exist and these bits need to be inspected even though no branching decisions need to be made
  - => search time can be longer than necessary
  - => one-child nodes consume memory
- Lookup time of a binary trie is O(W) and memory requirement O(NW)
  - W is the address length in bits and N the number of entries in a table
- Path-compression techniques can be used to remove unnecessary one-branch nodes and reduce search time and memory consumption

Compressed Tries

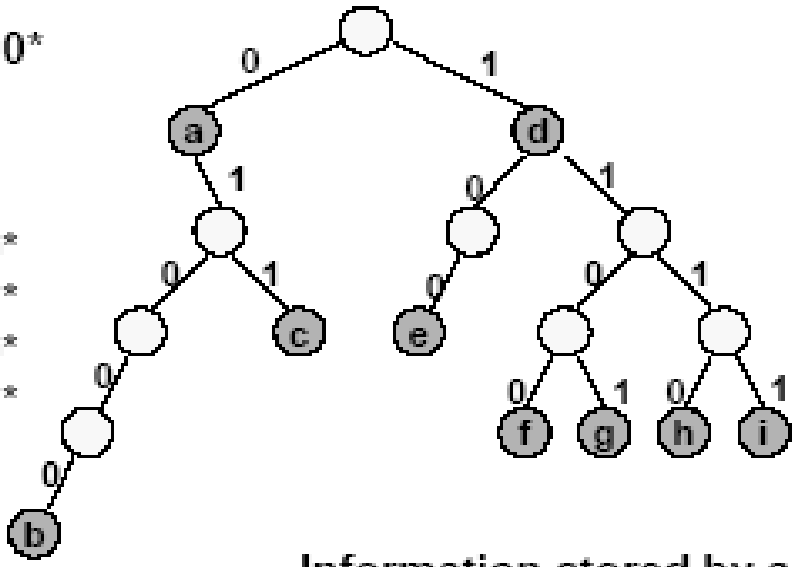
- Path-compression was first introduced in a scheme called Patricia (“Practical Algorithm to Retrieve Information Coded in Alphanumeric”) , which is an improvement of the binary trie structure
    - It is based on the observation that an internal node, which does not contain a prefix and is an only child, can be removed.
    - Removal of internal nodes requires that information in them be added in remaining nodes so that search operations can be performed correctly, e.g. a simple mechanism is to store a number, which indicates how many nodes have been skipped (skip value) or the number of the next address bit to be inspected
- There are many ways to exploit path-compression technique, an example is shown on the next slide
- Lookup time is O(W) and storage requirements O(N)

Example

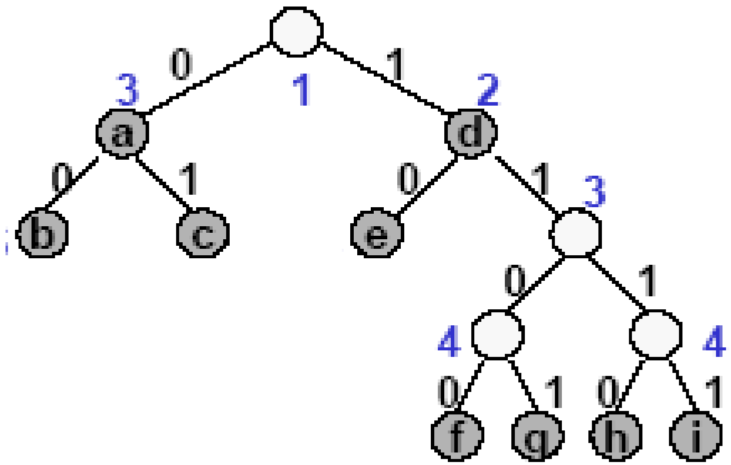
Prefixes:

a	- 0*
b	- 01000*
c	- 011*
d	- 1*
e	- 100*
f	- 1100*
g	- 1101*
h	- 1110*
I	- 1111*

Uncompressed binary trie



Compressed binary trie



Information stored by a node:

<table>
<tr><td colspan="2">Bit string</td><td colspan="2">Next-hop ptr
(if prefix)</td><td colspan="2">Bit position</td></tr>
<tr><td colspan="3">Left-ptr</td><td colspan="3">Right-ptr</td></tr>
</table>

Compressed Tries

- Two nodes preceding b have been removed
- Since prefix a was located at one child node, it was moved to the nearest descendant, which is not a one-child node
- If several one-child nodes, in a path to be compressed, contain prefixes, a list of prefixes must be maintained in some of the nodes
- Due to removal of one-child nodes, the search jumps directly to an address bit where a significant decision is to be made
  - => bit position of the next address bit to be inspected must be stored
  - => bit strings of prefixes must be explicitly stored

Search

- **LPM = Search (root, address[1..32], null)**

pseudo-code:

```
bitstring Search( node, address[], BMP){
    ❖ IF compare_bitstring_N( node->bitstring, address, length(node->bitstring))
        ✓ BMP = node->bitstring;   // match
    ❖ ELSE
        ✓ return BMP;              // no match
    ❖ IF (node->bit_position ==0)        // no children to check
        ✓ return BMP;
    ❖ IF  (address[node->bit_position] ==0)
        ✓ return Search(node->left_ptr, address[], BMP);
    ❖ ELSE        // bit checked ==1
        ✓ return Search(node->right_ptr, address[], BMP);
    }
```

Search example

In the previous example case, take an address beginning with 010110

- Start from root and since its bit position number is "1", inspect the first bit of the address
  - => 1st bit is "0" => go to the left
  - => since this node is marked as a prefix, compare prefix a ("0") with the corresponding part of the address => they match
  - => keep a as the BMP so far
  - => bit position number of the new node is "3" so skip the 2nd address bit and inspect the 3rd one, which is "0" => proceed left
  - => next node includes a prefix so compare prefix "b" with the corresponding part of address
  - => no match => stop search => the last recorded BMP is "a"

Multibit trie

- Drawback of binary (1-bit) trie is that one bit at a time is inspected and the number of memory accesses (in the worst case) can be 32 for IPv4 or 64 for IPv6
- Number of lookups can be substantially reduced by using a multibit trie structure, i.e. several bits are compared at a time
  - For example, inspecting four bits at a time would lead to only 8 memory accesses in the worst case for an IPv4 address
- Number of bits (K) to be inspected is called the *stride*
- There are fixed and variable stride algorithms.

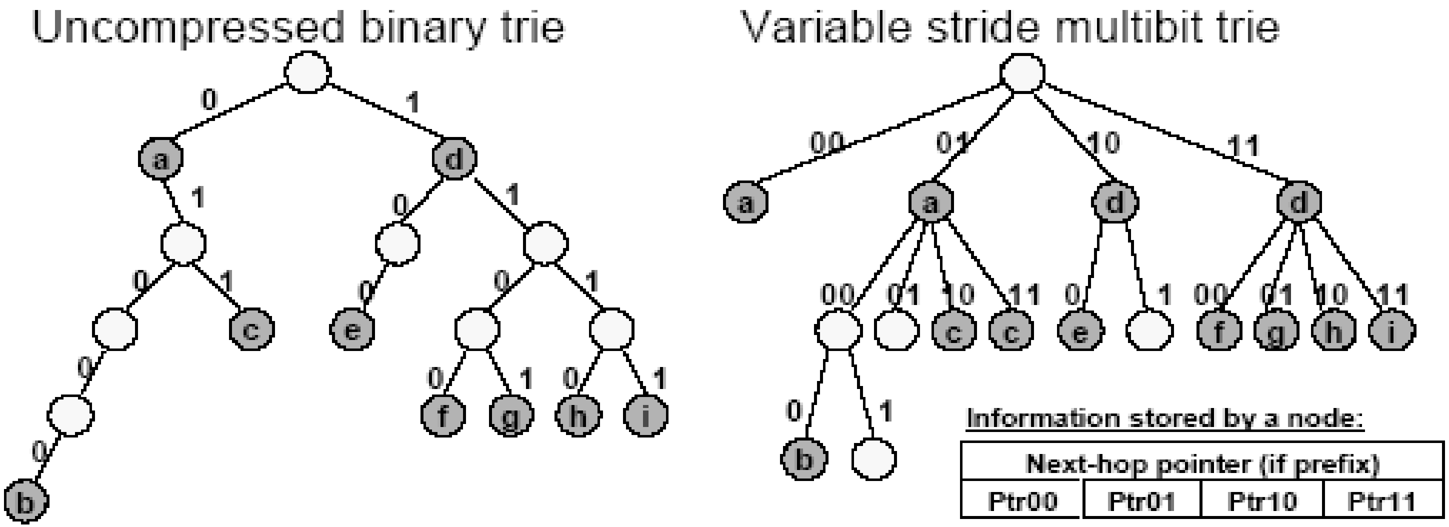
Multibit Tries

- In a K-bit trie, each node has 2^K pointers
- If a route prefix is not a multiple of K, it needs to be expanded to K or its multiples
- Lookup time is O(W/K) and storage requirements increase to $O(2^K NW/K)$

Prefix	Original	Exp. even	Exp. 3 and 5
a	0*	00* 01*	000* 001* 010* 011*
b	01000*	010000* 010001*	01000*
c	011*	0110* 0111*	011*
d	1*	10* 11*	100* 101* 110* 111*
e	100*	1000* 1001*	100*
f	1100*	1100*	11000* 11001*
g	1101*	1101*	11010* 11011*
h	1110*	1110*	11100* 11101*
i	1111*	1111*	11110* 11111*

Multibit Tries

- Prefixes “a” and “d” are expanded to length 2 and prefix “c” has been expanded to length 4 (rest of the prefixes remain unchanged)
- Height of the trie has been decreased and so has the number of memory accesses when doing a search



Uncompressed binary trie

Variable stride multibit trie

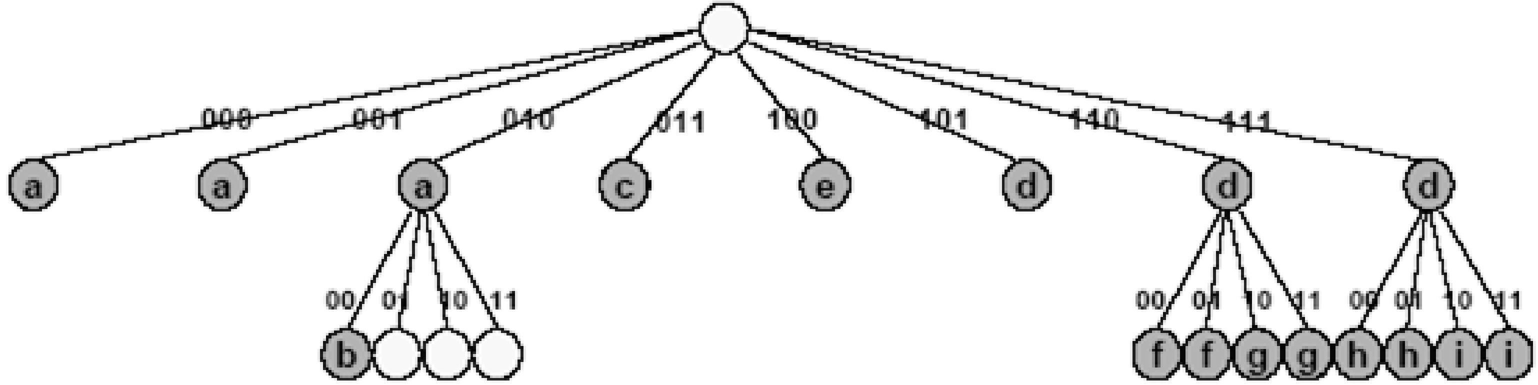
Information stored by a node:

Next-hop pointer (if prefix)			
Ptr00	Ptr01	Ptr10	Ptr11

Multibit Tries

- Prefixes a and d are expanded to length 3; f, g, h and i are expanded to length 5. The rest of the prefixes remain.
- When an expanded prefix collides with an existing unexpanded one, the forwarding information of the existing one must be preserved (to respect the longest match)

Fixed stride multibit trie



Multibit Tries

- Search in a multibit trie is essentially the same as search in a binary (1-bit) trie - successively look for longer prefixes that match and the last one found is the longest prefix for a given address
- Multibit tries do linear search on length as binary tries do, but the search is faster because the trie is traversed using larger strides
- A multibit trie is a fixed stride system, if all nodes at the same level have the same stride size, otherwise it is a variable stride system
- Fixed strides are simpler to implement than variable strides, but usually consume more memory

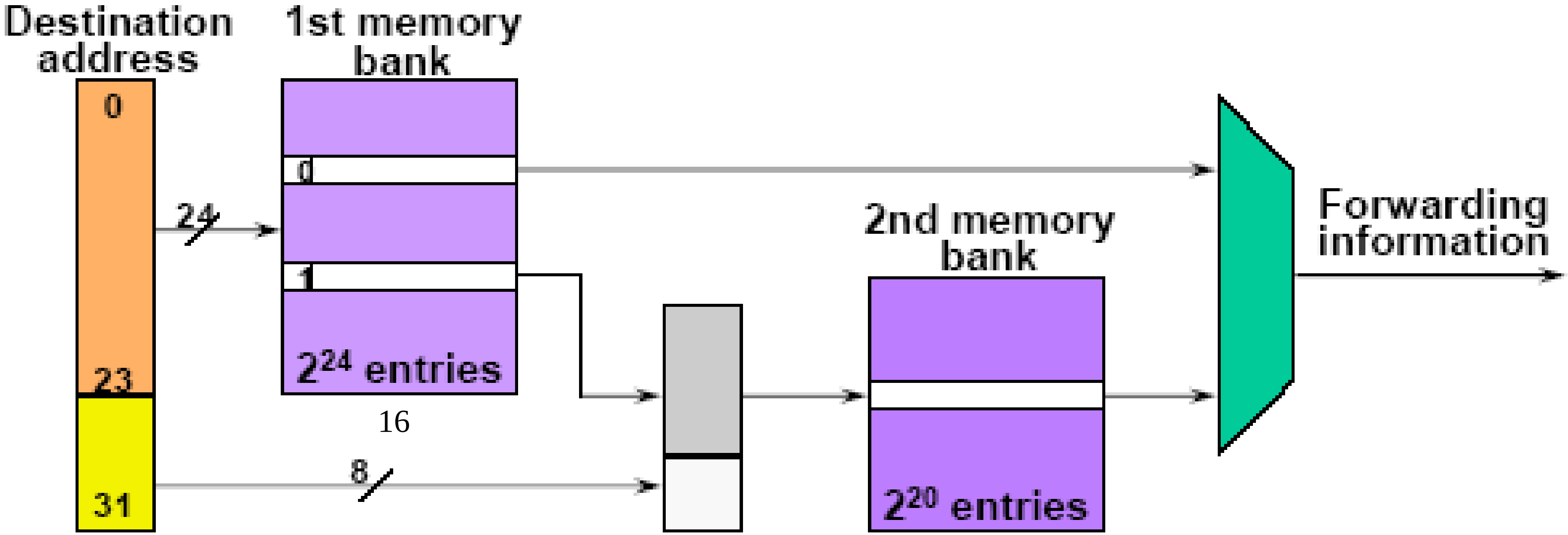
Stride size and update

- Choice of stride size is a trade-off between search speed and memory consumption
  - In the extreme case, a trie with a single level could be made (stride size = 32) and search would take only one memory access, but a huge amount of memory would be required (2^{32} entries for IPv4) and high update cost.
- A multibit trie with several levels allows, by varying stride K, an interesting trade-off between search time, memory consumption and update time
  - Larger strides make search faster..
  - .. but memory consumption increases and updates will require more entries to be modified (due to expansion)
  - See LC-tries (linux IPv4 route lookup)
    - https://vincent.bernat.ch/en/blog/2017-ipv4-route-lookup-linux

Multibit tries in HW

- In core network routers, target lookup times are very short and lookup algorithms are implemented in hardware to meet the required speed
- Basic scheme uses two level multibit trie with fixed strides - 24 bits at the first level and 8 at the second level
- In backbone routers, most of the entries have a prefix length of 24 bits or less => longest prefix match found in one memory access in the majority of cases
- Only a small number of sub-entries at the 2nd level
- 1st level has 2^{24} nodes and is implemented as a table with the same number of entries

Multibit tries in HW



If longest route with this 24 bit prefix < 25 bits

0	*Next hop*
1 bit	*15 bits*

If longest route with this 24 bit prefix > 24 bits

1	*Index to 2nd table*
1 bit	*15 bits*

An entry at the 1st level contains either the forwarding information or an index to the corresponding sub-trie at the 2nd level

=> a memory bank of 32 Mbytes is needed to store 2^{24} entries of the 1st bank, plus a number of 256x16 bit blocks in the second

Multibit tries in HW

- Number of sub-tries at the 2nd level depends on the number of prefixes longer than 24 bits
- 2nd level stride is 8 bits => a sub-trie at the 2nd level has 2^8=256 leaves
- Size of 2nd memory bank depends on the expected worst case prefix length distribution, e.g. 2^{20} one-byte entries (a memory bank of 1 Mbytes) supports a maximum of 2^{12}= 4096 sub-tries at the 2nd level
- Lookup requires a maximum of two memory accesses - memory accesses can be pipelined or parallelized to speed up performance
- Since the first stride is 24 bits and leaf pushing is used, updates may take a long time in some cases

References

- R. Rojas-Cessa, "Interconnections for Computer Communications and Packet Networks", ISBN 9781482226966, CRC Press, 2017.
- F. Baboescu, S. Singh, and G. Varghese. Packet Classification for Core Routers: Is There an Alternative to CAMs?, Proceedings of IEEE Infocom 2003.
- M. A. Ruiz-Sanchez, E.. Biersack, and W. Dabbous. Survey and Taxonomy of IP Address Lookup Algorithms. IEEE Network Magazine, 15(2):8--23, March/April 2001.
- M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed Prefix Matching. ACM Transactions on Computer Systems, 2001.