

Lecture #13: packet forwarding pipelines

WPI CS4516 Spring 2019 D term

Instructor: Lorenzo De Carli (ldecarli@wpi.edu)

Routers and switches

- **Switch: layer-II device** (forwards Ethernet frames)
 - Ethernet uses a flat addressing scheme – there is no implicit structure in an address
- **Router: layer-III device** (forward IP packet)
 - IP uses a hierarchical addressing scheme – an IP address represents both a subnet and a host
- But regardless... both devices need to examine data received on input interfaces and decide on which output it should be forwarded
- Other approaches are possible (e.g., OpenFlow), but they all require a device to make a **forwarding decision**

The case for flexibility in forwarding

- Categorization of devices among layer-II and layer-III nowadays is blurry
 - E.g., an OpenFlow switch can perform routing
- Moreover, the **same device** may need to perform **multiple types of forwarding**
 - E.g., a router with Ethernet interfaces needs to perform both layer-II and layer-III forwarding
 - And perhaps even support OpenFlow!
- It is important that modern networking hardware is designed with **multiple applications** in mind!

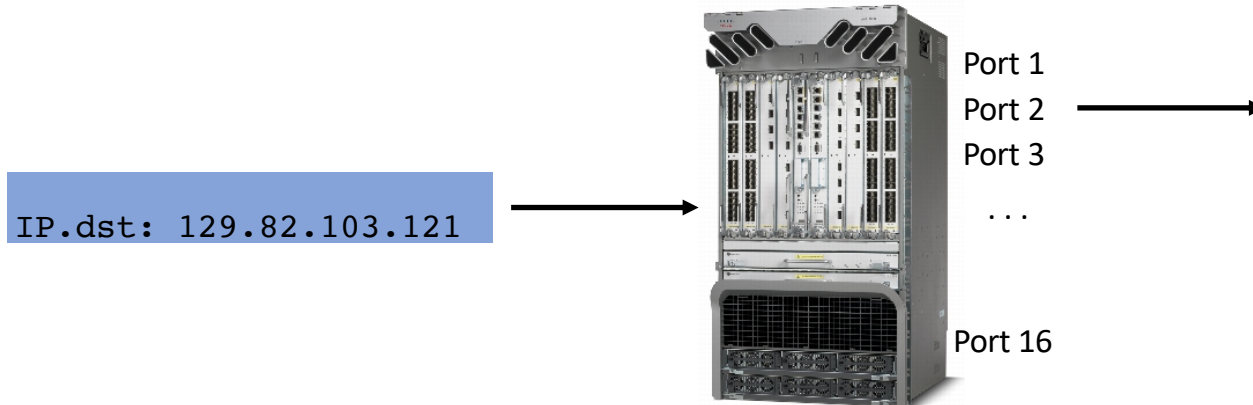
Structure of this lecture

- **IP forwarding:** how routers process IP packets and decide where to forward them next
- **PLUG:** a flexible forwarding engine that can accommodate various forwarding algorithms
- **Intel FlexPipe:** a flexible commercial forwarding pipeline

Let's begin with routers &
“traditional” IP forwarding

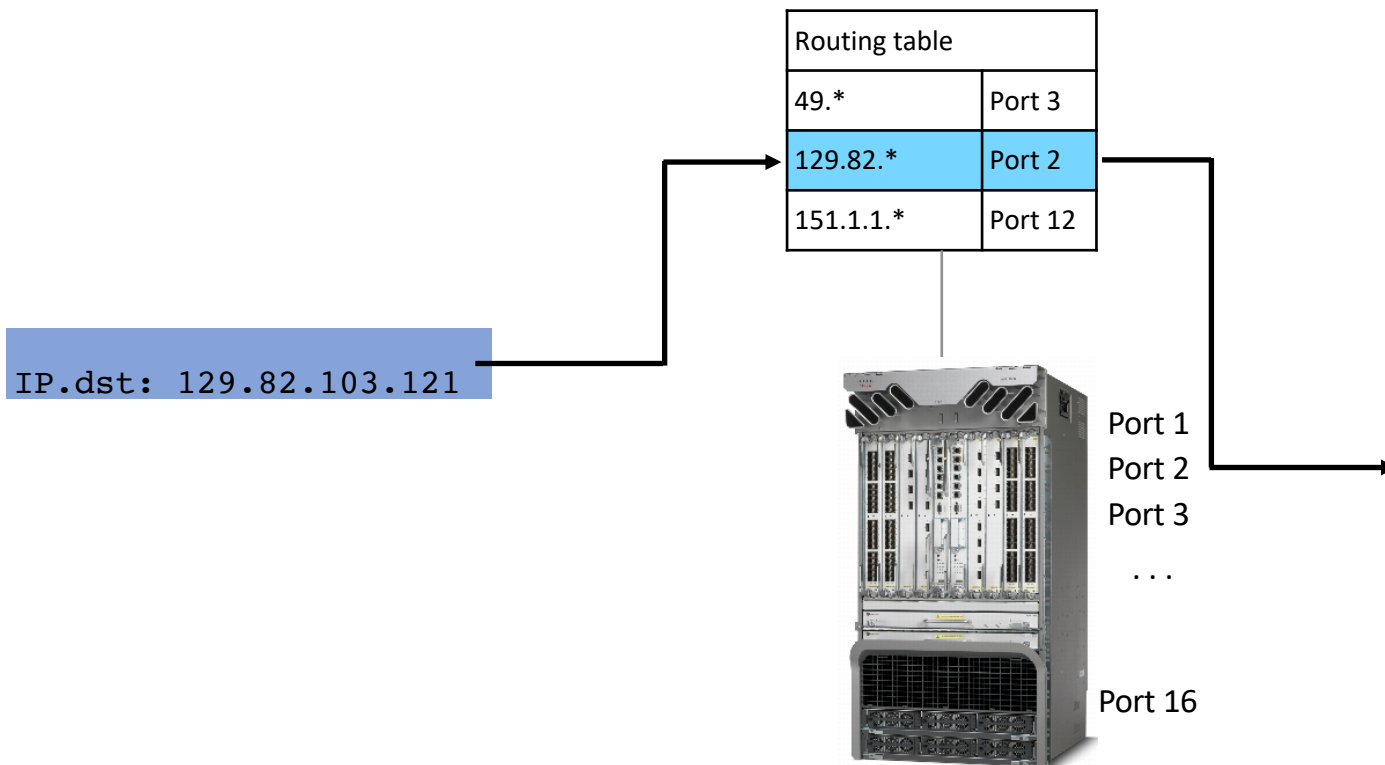
IP forwarding

- Input: a source IP address
- Output: an output port
 - Defines the link on which the packet must be sent



IP forwarding - II

- Internally, router determines output port by looking up a **routing table**

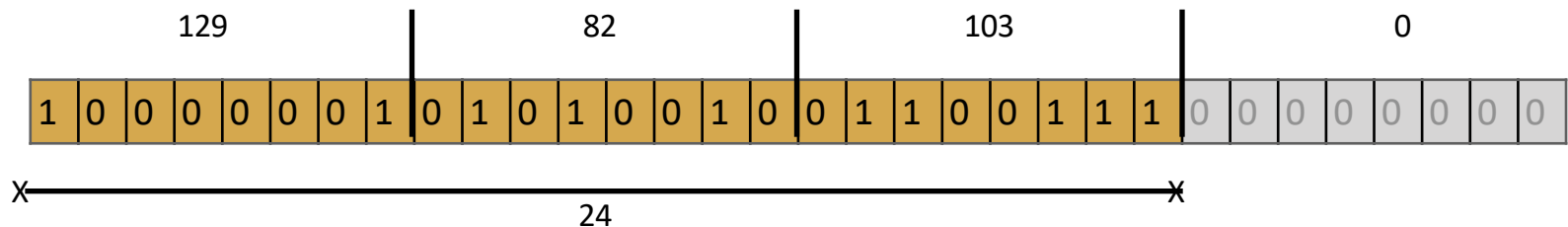


Longest prefix matching

- There are 4B+ possible 32-bit IP addresses
 - Not really (due to reserved/private IP address spaces), but still...
- There are 2^{128} possible 128-bit IPv6 addresses
- It would be **impractical** to encode individual (destination IP, port) **for each possible destination**

Longest prefix matching - II

- What do we do then? **Longest prefix matching**
- Observation: IP addresses are not randomly assigned
 - Organizations receive blocks of IP address space
 - Each block defined by IP+mask: e.g. 129.82.103.0/24

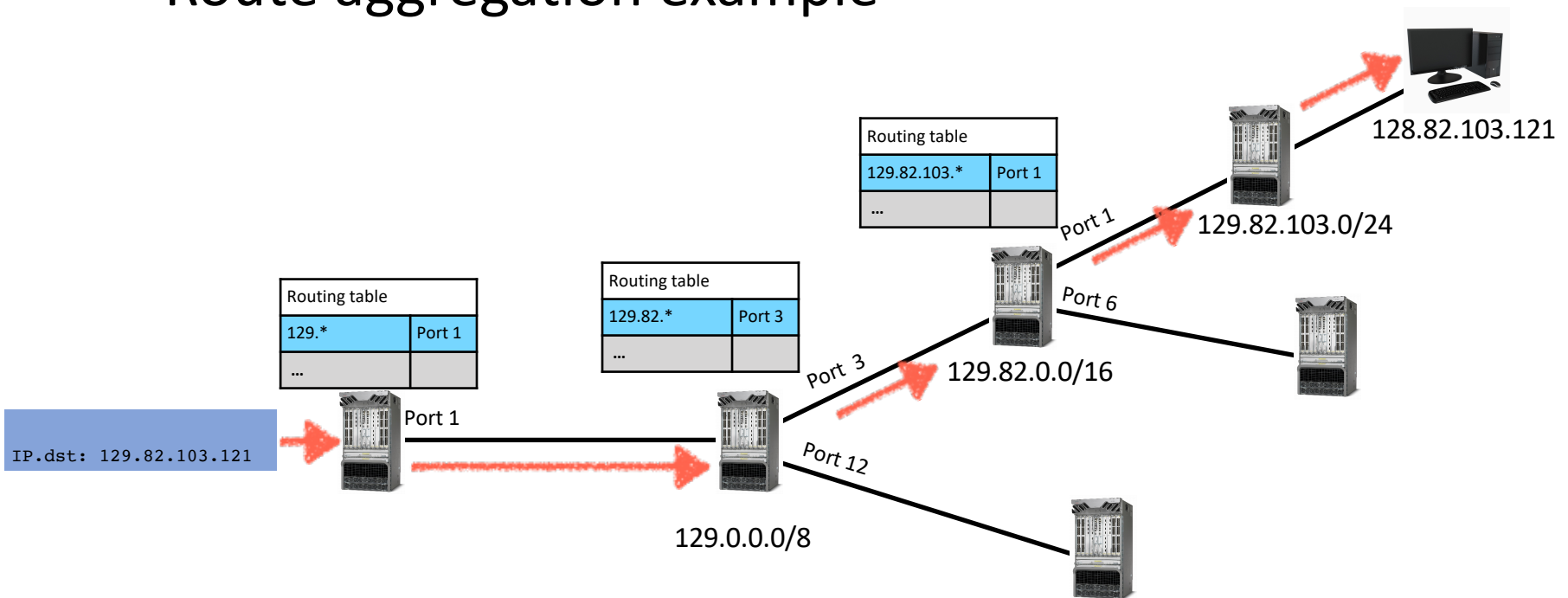


Longest prefix matching - III

- Organizations receive blocks of IP address space
 - Each block defined by **IP+mask**: e.g. 129.82.103.0/24
 - Machines in this block will be assigned addresses with the 129.82.103 prefix, e.g., 129.82.103.1, 129.82.103.24, 129.82.103.11, etc....
- Implication: **IP addresses which share the same prefix (and are hence part of the same subnet) also tend to share the same route**

Longest prefix matching - IV

- Route aggregation example




Longest prefix matching - V

- Why “longest”?
 - Given a destination IP address, there may be multiple entries in the routing table with matching prefixes of different length
 - In this case, **the most specific (longer) prefix is chosen**

Longest prefix matching - VI

- Longest prefix matching - general description:
 - **Identify all rules** whose prefix matches the destination IP address from the current packet
 - Among all the results, **pick the rule with the longest prefix**



Routing table	
129.82.103.*	Port 1
129.82.103.121	Port 6

Longest prefix matching - VII

- Problems with longest prefix matching:
 - Routing tables may contain **hundreds of thousands of rules**
 - Need to find all rules that match a certain IP, then pick the one with the longest prefix
 - **Need to do it fast!** (1 Tb/s w/ 512B packets: 250M+ packets/s)
 - Impossible to use linear search
 - Routing “tables” are not really tables
 - **Need optimized data structure!**

Efficient longest prefix matching

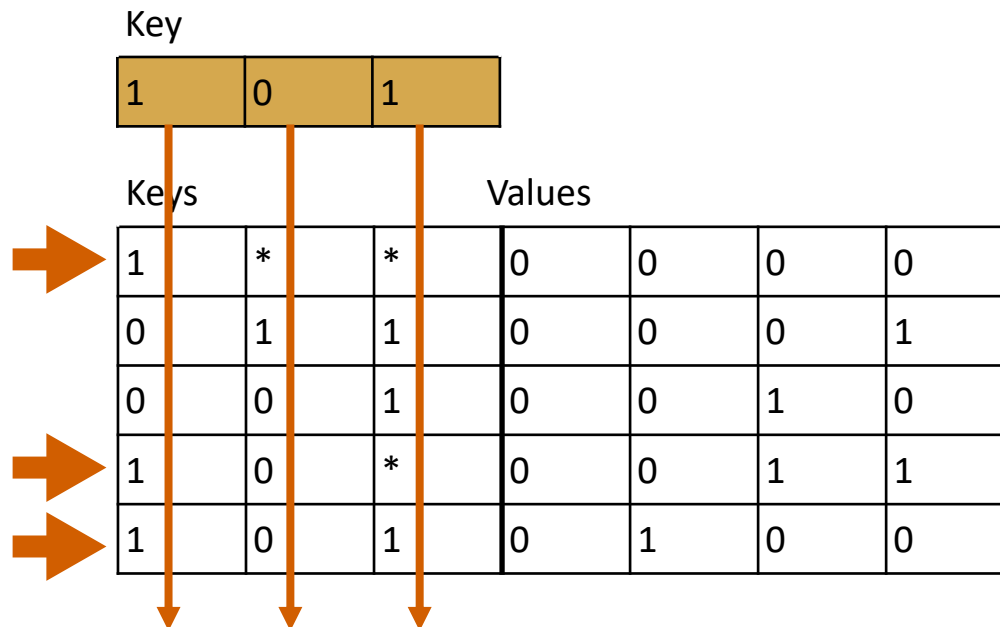
- Router designers have looked at a number of approaches to efficient longest prefix matching:
 - **TCAM-based approach** (specialized memory)
 - **Algorithmic approaches** (SRAM + optimized data structures + HW or SW lookup engine)
 - Tries
 - Lulea Tries
 - Tree bitmaps
 - ...

TCAMs

- **TCAM: Ternary Content-Access Memory**
- **Content-Access Memory:** a hardware memory which is indexed by **keys** instead of **memory addresses**
 - Think of it as an hardware key-value store
 - Given a key, returns the corresponding value
- **Ternary:** not all key bits need to be specified in each entry
 - **Some bits can be set to “don't care”**
 - Those parts of the key always match

TCAM - example

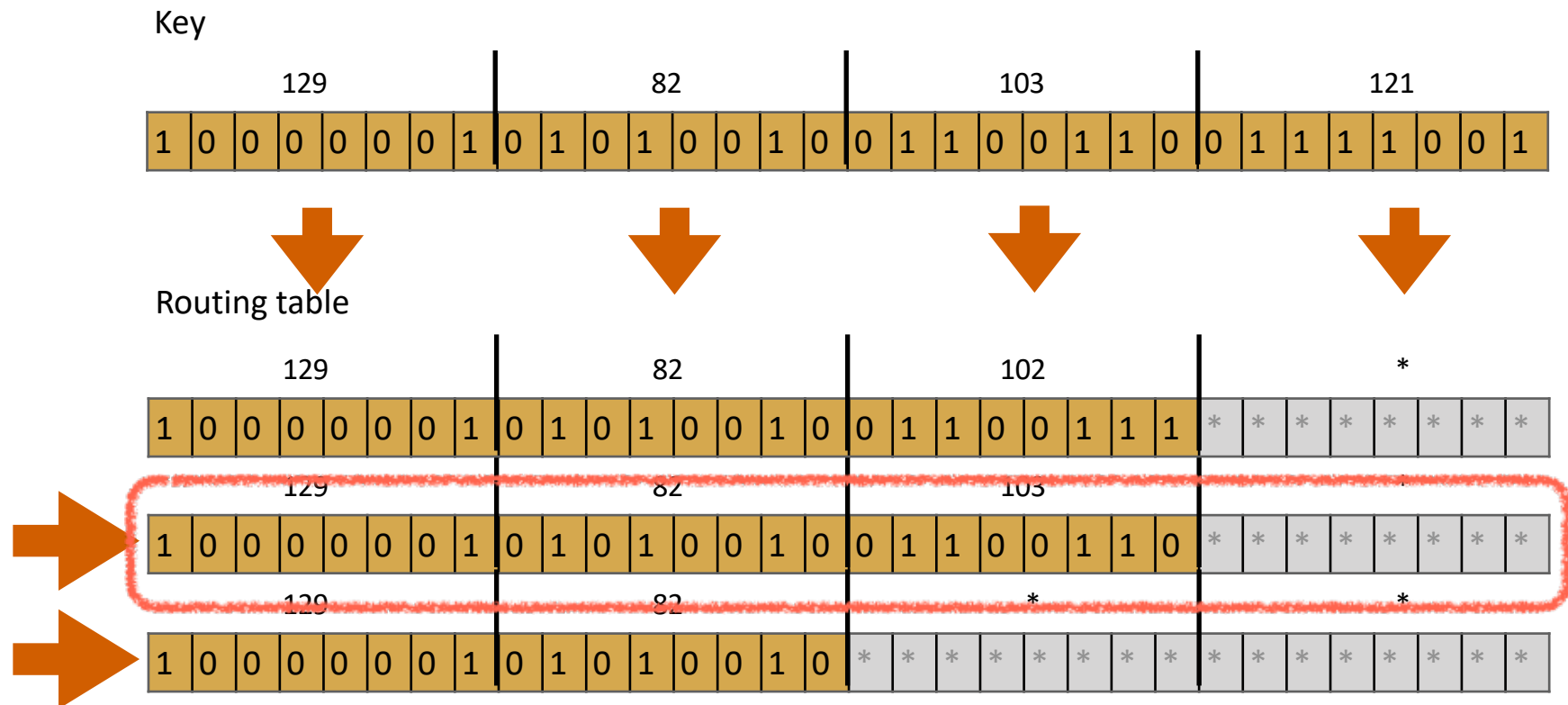
- Keys: 3-bit wide
- Values: 4-bit wide



How is this useful?

- **Route prefixes** can be directly stored as **keys**
 - The part of the prefix which is not covered by the mask is set as “don’t care”
- **Output ports** stored as **values**
- In general, searching for prefix in a TCAM will return **multiple entries**
 - Given a search key, among all active entries **pick the one with the least “don't care” bits**

TCAM+LPM - Example



TCAM - pros and cons

- **Advantages:**

- Fast! Can search or update in 1 memory cycle ($\sim 10\text{ns}$)
- Can handle large routing tables (~ 100000 prefixes)

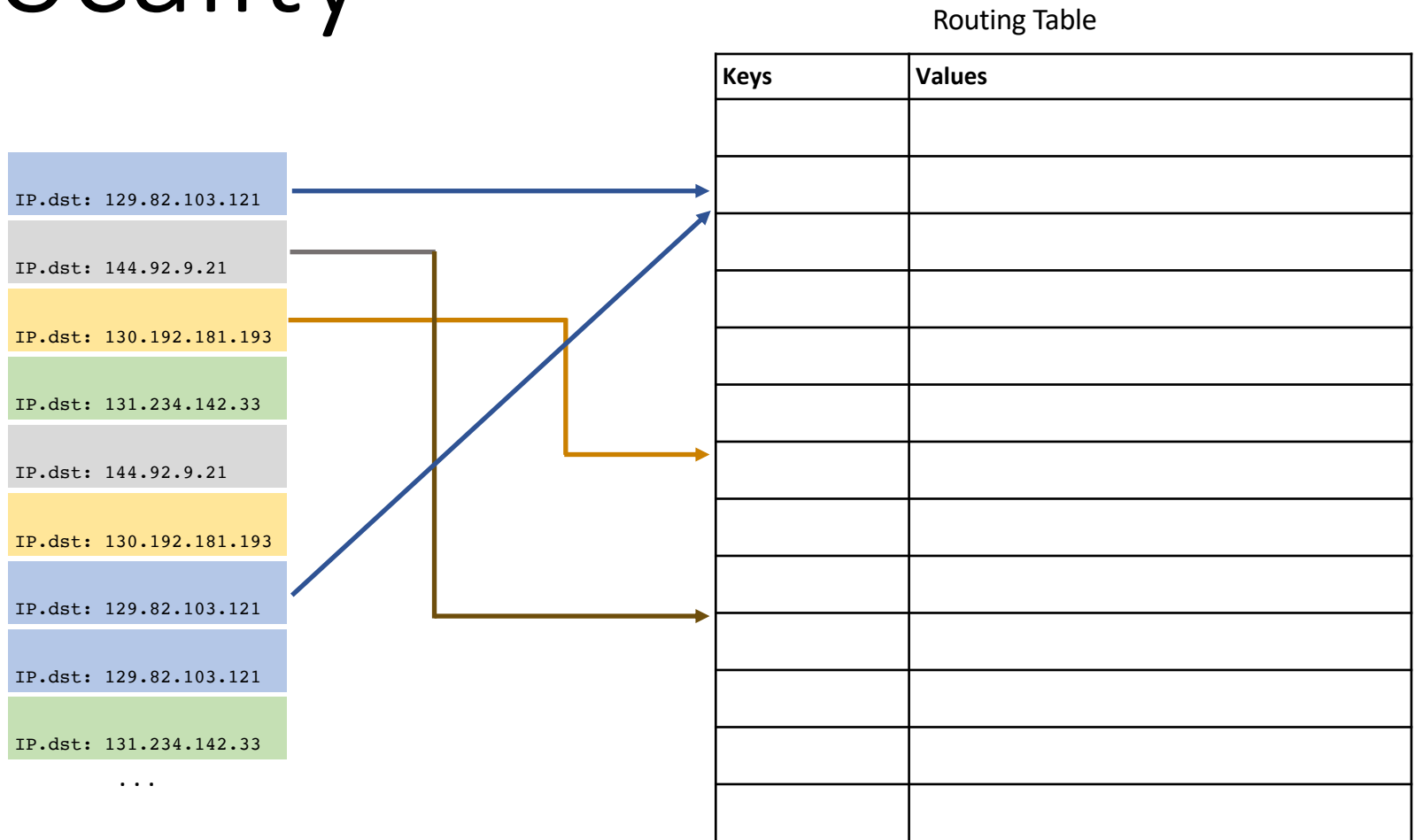
- **Disadvantages:**

- Area-hungry (12 transistors per bit - compare to 6 transistors per bit required by regular SRAM and 1 transistor + 1 capacitor per bit in DRAM)
- Power-hungry (because of the parallel compare between keys and all entries)
- Arbitration delay ($\sim 5\text{ ns}$ to pick winning entry)
- Require dedicated chip
- **Can we have an alternative approach not requiring specialized memory?**

LPM - algorithmic approaches

- Easiest: use generic data structure
 - E.g. **potentially slow, unpredictable latency**
 - Most conventional structures are designed for exact matching
- Next: use generic data structure + cache
 - Problem: **IP lookups have poor memory locality**
 - Why? Core routers process **tens of thousands of network flows at the same time**
 - Different flows have different destination IPs, which cause **different memory locations to be accessed**

LPM - poor memory locality



Latency

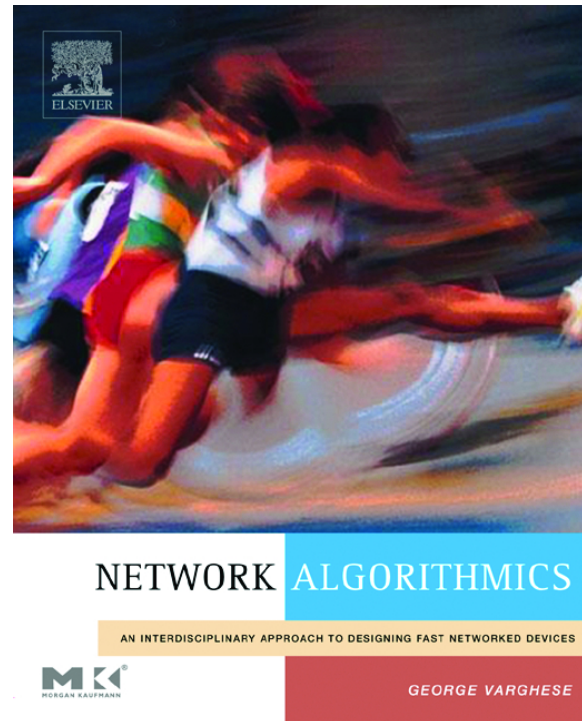
- **Latency is also a problem!**
 - DRAM latency: $\sim 30\text{ns}$; SRAM latency $\sim 3\text{ns}$
 - For large routing tables, SRAM may be too expensive - so need to use DRAM
 - Slow, and caching does not help!

Take away points

- **Generic data structures/memory** hierarchy have **poor performance** on IP lookup operations
- Need **specialized algorithm w/ dedicated data structures for efficiency**
- We are going to see one example: **tries**
- Typically, algorithm and memory are implemented as **specialized hardware**

Sources

- George Varghese, “Network Algorithmics”, Morgan Kauffmann, December 2004

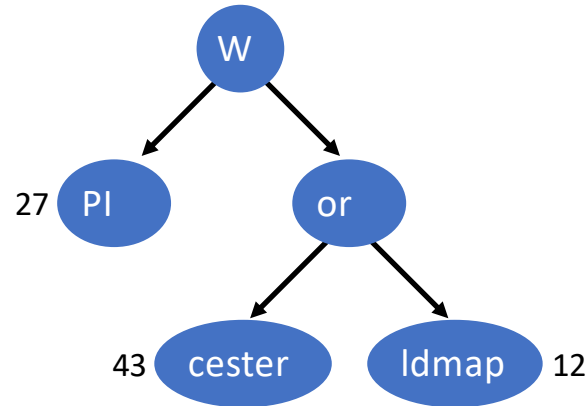


Lookup tries

- **Trie:** tree data structure used to store values associated with a **set of strings**
- Each **node** represents a **prefix** of one or more strings in the set
- Allows **compact encoding**, particularly when many strings share prefixes

Lookup trie - example

- Example: need to store the following keys and values:
 - WPI: 27
 - Worcester: 43
 - Worldmap: 12



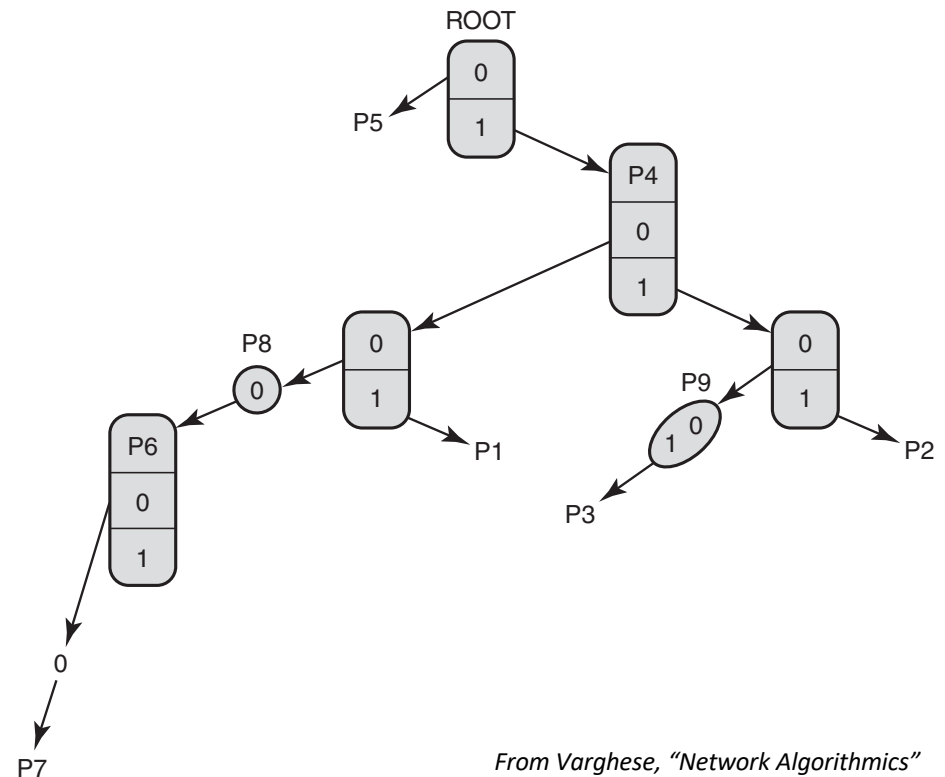
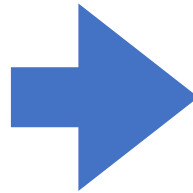
Unibit tries

- In LPM, **strings are IP addresses** and **values are output ports**
- Simplest possible trie-based approach: **unibit trie**
- Each node “consumes” 1 bit of the destination IP address
- Each **node** which stores **the last bit of a prefix** in the routing table also stores a pointer to the **corresponding port number**

Unibit tries - example

Routing Table

Prefix	Port
101*	1
111*	2
11001*	3
1*	4
0*	5
1000*	6
100000*	7
100*	8
110*	9

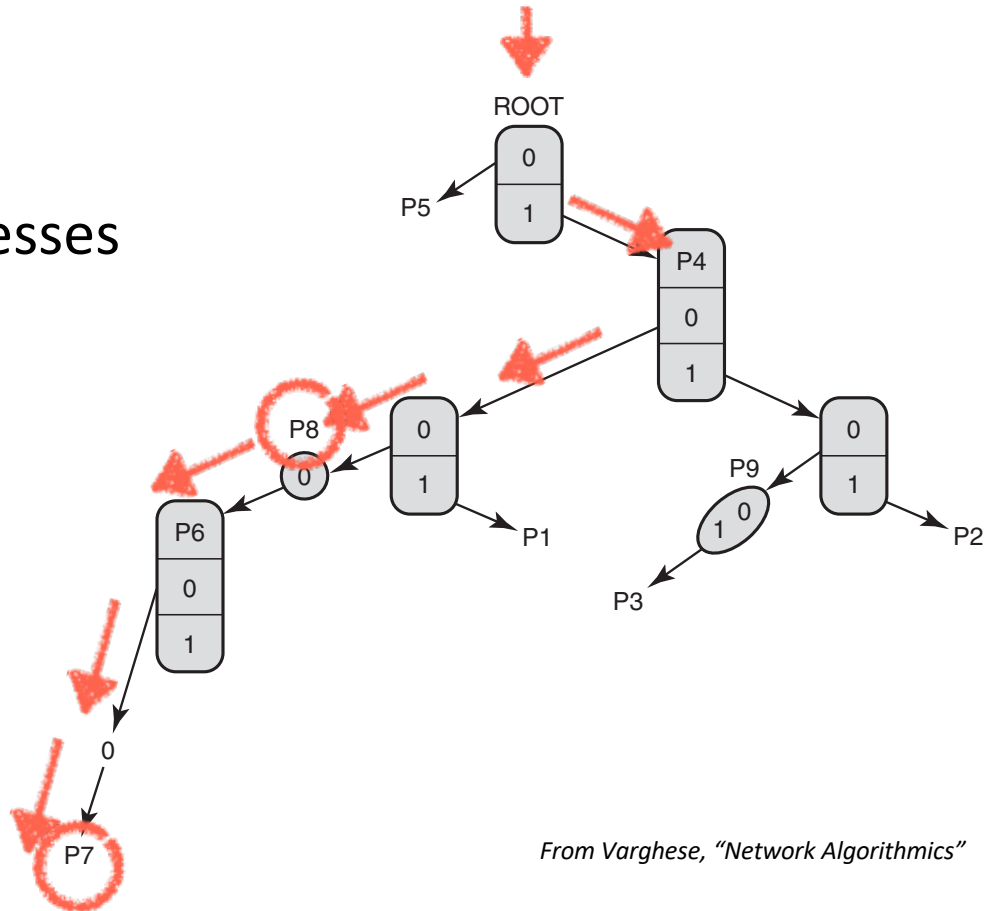


From Varghese, "Network Algorithmics"

Note, unibit tries are an example of **fixed-stride** tries

Unibit tries - example II

- Need to lookup **10000011**
 - (Using 8-bit addresses for simplicity)



From Varghese, "Network Algorithmics"

P7 wins over P8 (longest prefix)

Considerations on unibit tries

- Convenient for understanding...
- ...not so useful in practice
- **Slow!** Up to 32 memory accesses per lookup
- **Idea:** improve efficiency by looking up **multiple bits in each trie node**

Multibit tries

- 1 bit per node is too slooow!
- What about $B > 1$ bits per node? (**Multibit tries**)
 - #memory accesses cut to $32/B$ (B is the **stride**)
- **Advantage:** faster (less latency)
- **Disadvantage:** what do we do with prefixes whose length is not a multiple of N?
 - E.g. prefix 10101*** but stride $B = 3$

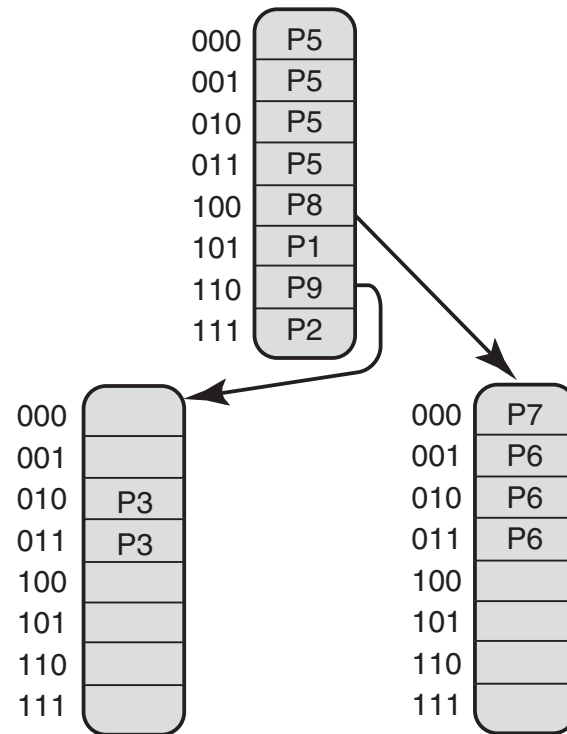
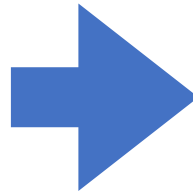
Multibit tries - II

- Problem: how can we support every prefix length with multibit tries?
 - Use a technique called **prefix expansion**
 - For each stride, generate all possible prefixes
 - Simple, but...
 - Generates entries even for prefixes that are not in the routing table

Prefix expansion example

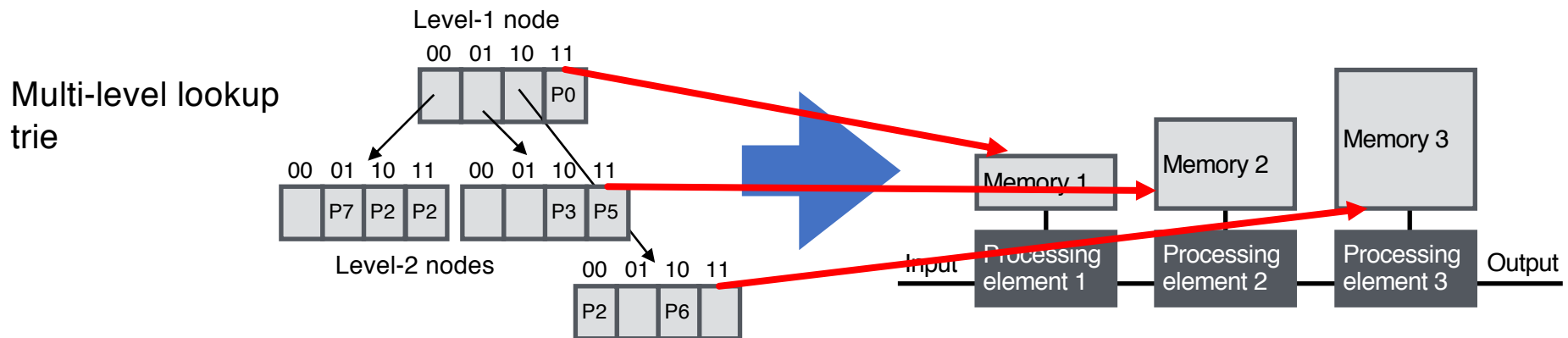
Routing Table

Prefix	Rule
101*	P1
111*	P2
11001*	P3
1*	P4
0*	P5
1000*	P6
100000*	P7
100*	P8
110*	P9



How can we implement this?

- Trie-based lookup suitable for SW implementation, but still **slow due to DRAM memory latency**
- For high-throughput applications, trie-based algorithms can be implemented using **specialized hardware pipeline**:



An outline of my own work

PLUG: Flexible Lookup Modules for Rapid Deployment of New Protocols in High-speed Routers

Lorenzo De Carli[†] Yi Pan[†] Amit Kumar[†] Cristian Estan^{†‡} Karthikeyan Sankaralingam[†]

[†]University of Wisconsin-Madison [‡]NetLogic Microsystems
{lorenzo,yipan,akumar,karu}@cs.wisc.edu estan@netlogicmicro.com

ABSTRACT

New protocols for the data link and network layer are being proposed to address limitations of current protocols in terms of scalability, security, and manageability. High-speed routers and switches that implement these protocols traditionally perform packet processing using ASICs which offer high speed, low chip area, and low power. But with inflexible custom hardware, the deployment of new protocols could happen only through equipment upgrades. While newer routers use more flexible network processors for data plane processing, due to power and area constraints lookups in forwarding tables are done with custom lookup modules. Thus most of the proposed protocols can only be deployed with equipment upgrades.

To speed up the deployment of new protocols, we propose a flexible lookup module, PLUG (Pipelined Lookup Grid). We can achieve generality without loosing efficiency because various custom lookup modules have the same fundamental features we retain: area dominated by memories, simple processing, and strict access patterns defined by the data structure. We implemented IPv4, Ethernet, Ethane, and SEATTLE in our dataflow-based programming model for the PLUG and mapped them to the PLUG hardware which consists of a grid of tiles. Throughput, area, power, and latency of PLUGs are close to those of specialized lookup modules.

Categories and Subject Descriptors: B.4.1 [Data Communication Devices]: Processors; C.2.2 [Network Protocols]: Protocol Architecture (OSI model)

General Terms: Algorithms, Design, Performance

Keywords: lookup, flexibility, forwarding, dataflow, tiled architectures, high-speed routers

1. INTRODUCTION

The current Internet relies extensively on two protocols designed in the mid-'70s: Ethernet and IPv4. With time, due to needs not anticipated by the designs of these protocols, a number of new protocols, techniques and protocol extensions have been deployed in routers and switches changing how they process packets: Ethernet

[†]Work done while at University of Wisconsin-Madison.

bridging, virtual LANs, tunnels, classless addressing, access control lists, network address translation, to name a few. Yet, the existing infrastructure has many acute shortcomings and new protocols are sorely needed. Data link and network layer protocols have been proposed recently to improve scalability [23, 29], security [51, 53, 15, 14], reduce equipment cost [1, 25], to ease management [23, 28, 14], or to offer users more control over their traffic [44, 52]. The two main factors that slow down the deployment of new protocols are the inevitable tussle of the various stakeholders [19] and the need for physical equipment upgrades.

The use of new equipment is a necessity for changes at the physical layer such as switching to new media or upgrades to higher link speeds. But data link layer and network layer changes do not necessarily require changes to the hardware and can be accomplished through software alone if the hardware is sufficiently flexible. Our goal is to enable such deployment of innovative new data plane protocols without having to change the hardware.

Sustained efforts by academia and industry have produced mature systems that take us closer to this goal. The NetFPGA project's [39] FPGA-based architecture allows experimental deployment of changes to the data plane into operational backbones [10]. While FPGAs are an ideal platform for building high-speed prototypes for new protocols, high power and low area efficiency make them less appealing for commercial routers. Many equipment manufacturers have taken the approach of developing network processors that are more efficient than FPGAs. For example Cisco's Silicon Packet Processor [22] and QuantumFlow [18] network processors can handle tens of gigabits of traffic per second with packet processing done by fully programmable 32-bit RISC cores. But for efficiency they implement forwarding table lookup with separate hardware modules customized to the protocol. Many of the proposed new protocols use different forwarding table structures and on these platforms, they cannot be deployed with software updates (without degrading throughput). Thus we are left with the original problem of hardware upgrades for deploying new protocols.

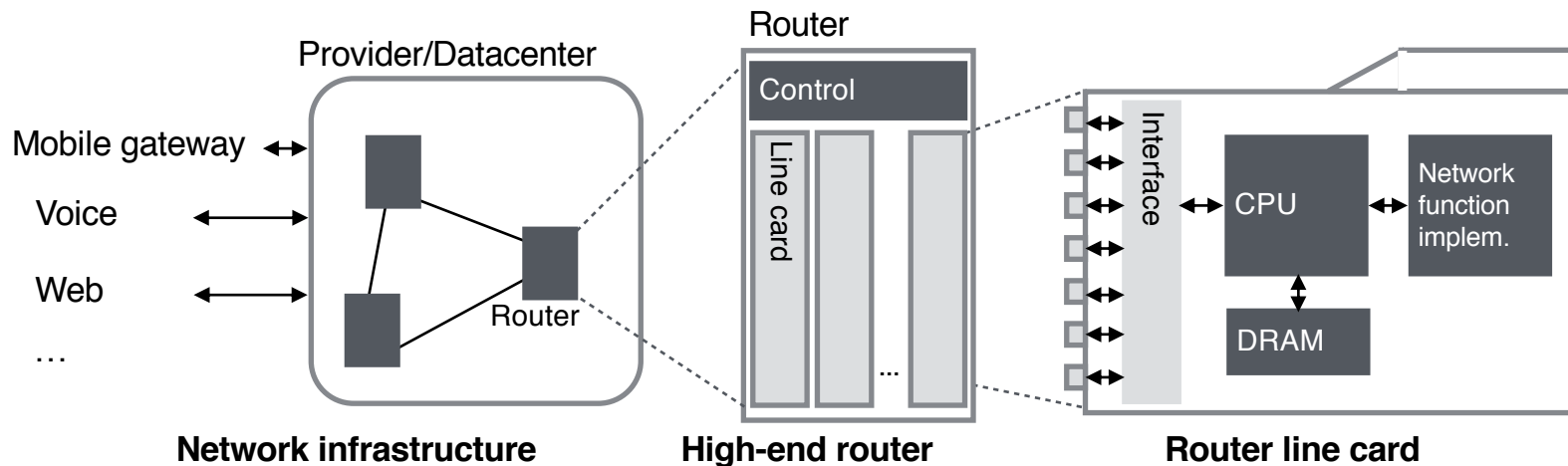
In this paper, we propose replacing custom lookup with flexible lookup modules that can accommodate new forwarding structures thus removing an important impediment to the speedy deployment of new protocols. The current lookup modules for different protocols have many fundamental similarities: they consist mostly of memories accessed according to strict access patterns defined by the data structure, they perform simple processing during lookup, and they function like deep pipelines with fixed latency and predictable throughput. We present a design for a general lookup module, PLUG (Pipelined Lookup Grid) which builds on these properties to achieve performance and efficiency close to those of existing custom lookup modules. Instead of using a completely flexible hardware substrate like an FPGA, PLUGs contain lightweight pro-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCOMM '09, August 17–21, 2009, Barcelona, Spain.
Copyright 2009 ACM 978-1-60558-594-9/09/08 ...\$10.00.

Background

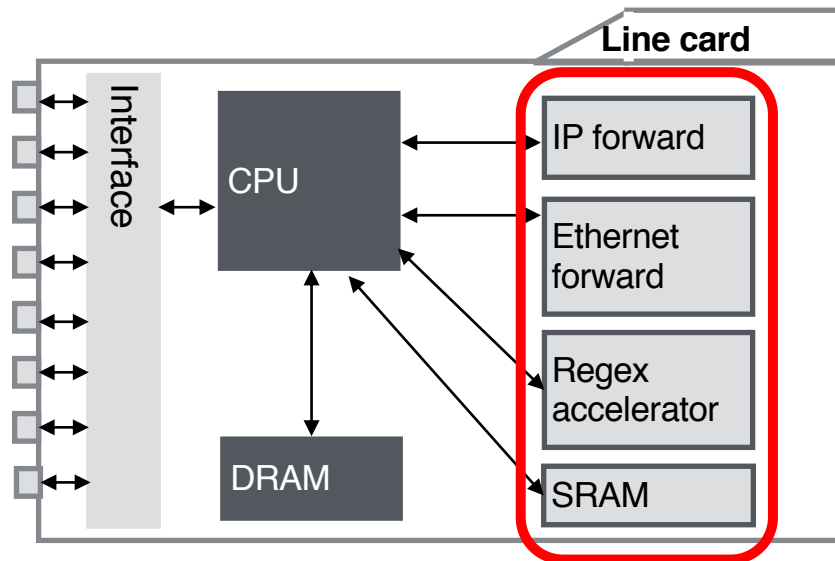
- Context: high-speed packet processing on routers



- Requirements: real-time processing, high throughput

Background - II

- Problem: different functions require different modules



Typically, custom ASICs and/or FPGAs:

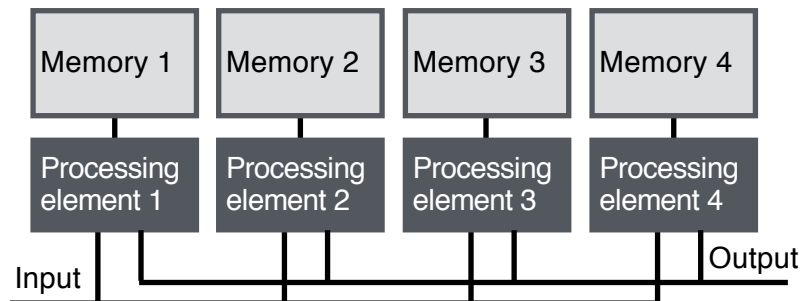
- Complicate to design
- Expensive
- Inflexible
- **Can we replace them with a single module?**

- Accelerator modules focus on data structure lookup

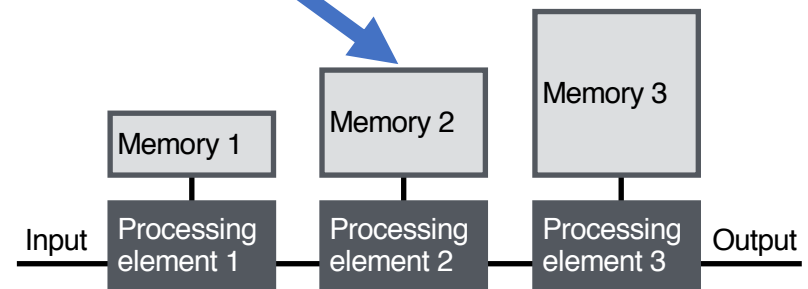
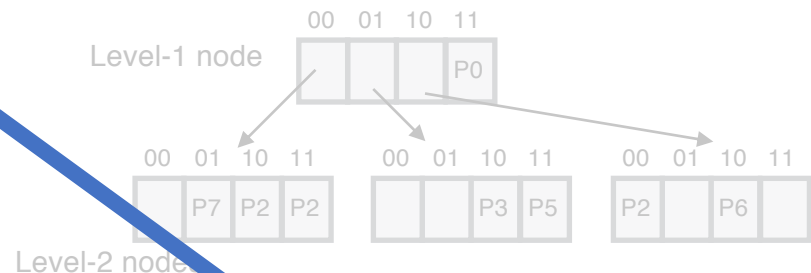
Background - III

- **Idea:** exploit commonalities between accelerators

- Orderly combination of steps
- Each step includes dedicated computation & memory
- Predictable communication patterns

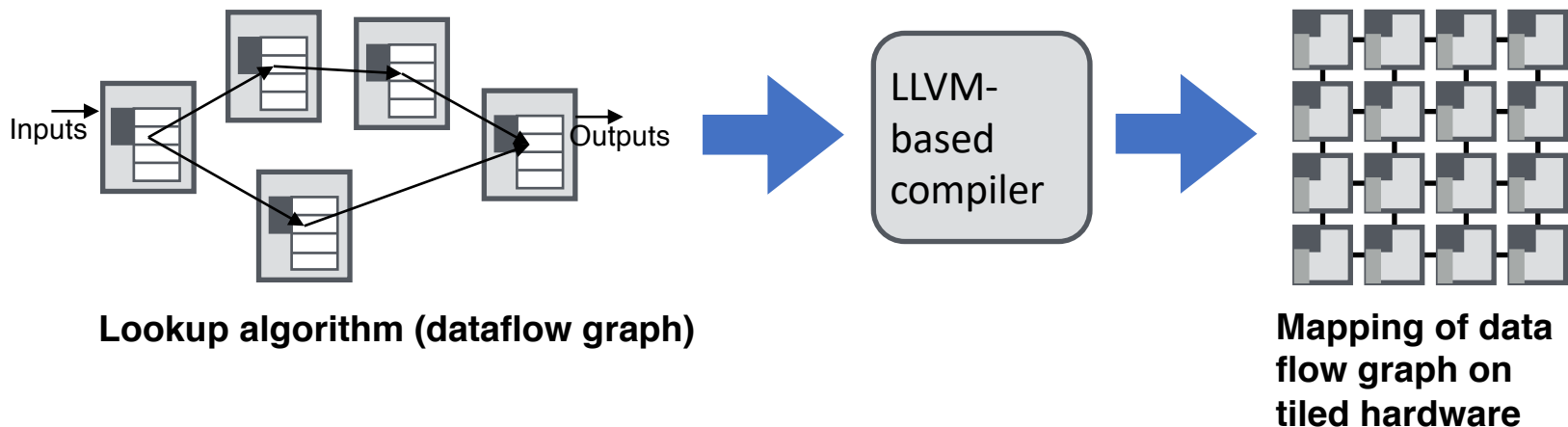


IP forwarding - lookup trie

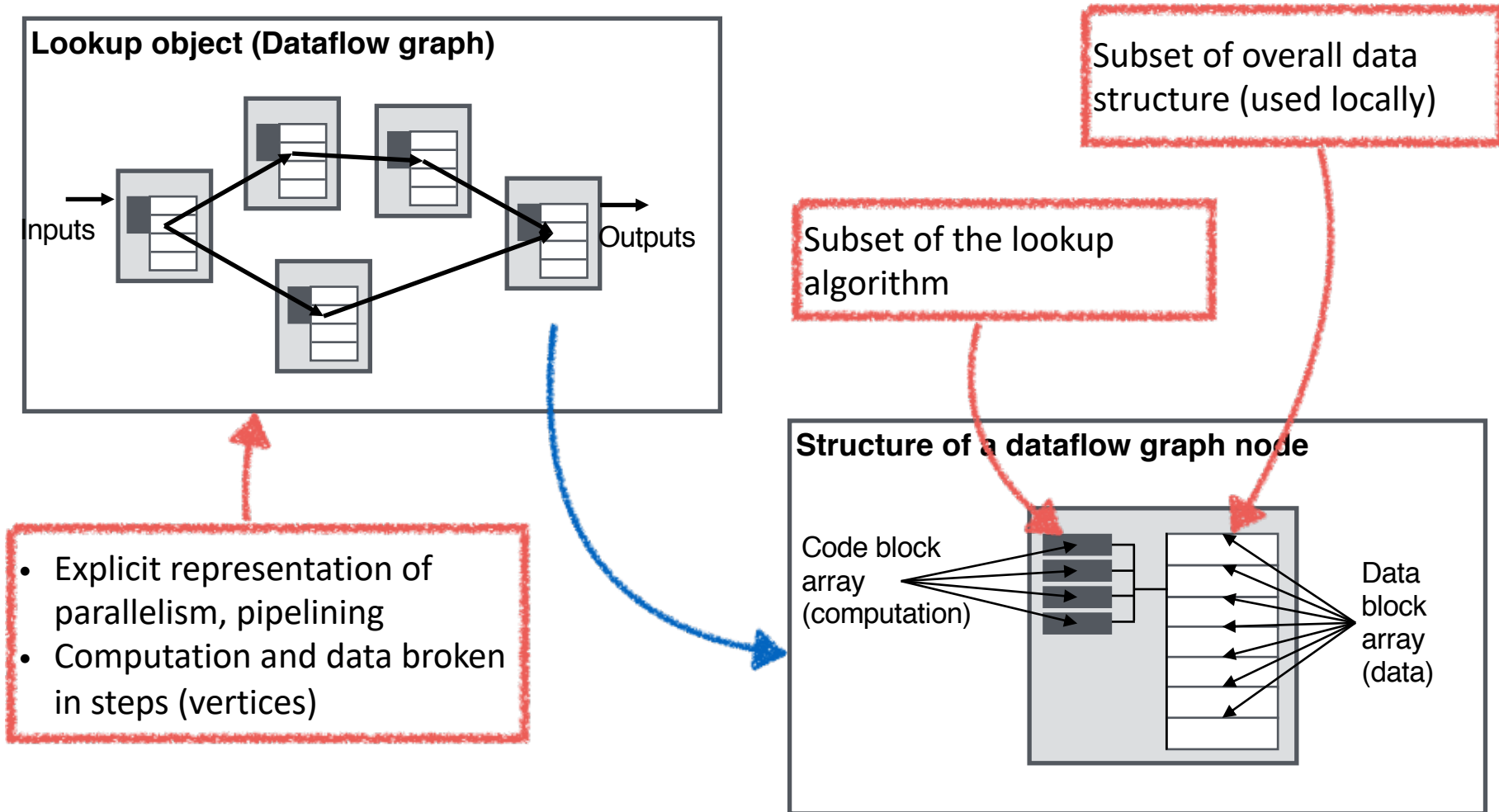


PLUG (Pipelined LookUp Grid)

- Programming model + HW
- Efficient execution of algorithmic steps with local data+computation
- Programming model: algorithms as **dataflow graphs** (DFGs)
- Massively parallel HW efficiently executes DFG applications (**tile array**)

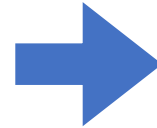
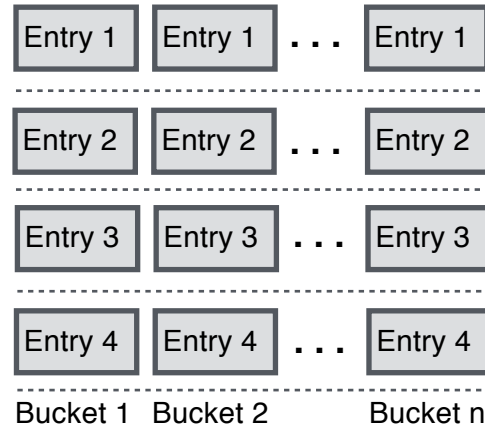


Structure of a lookup object

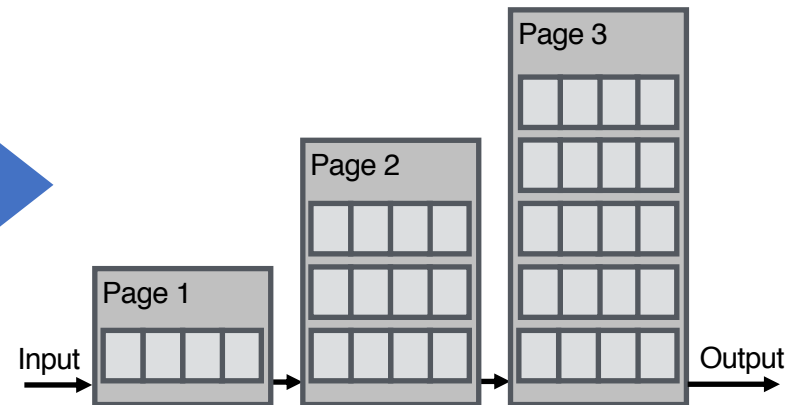
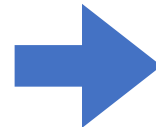
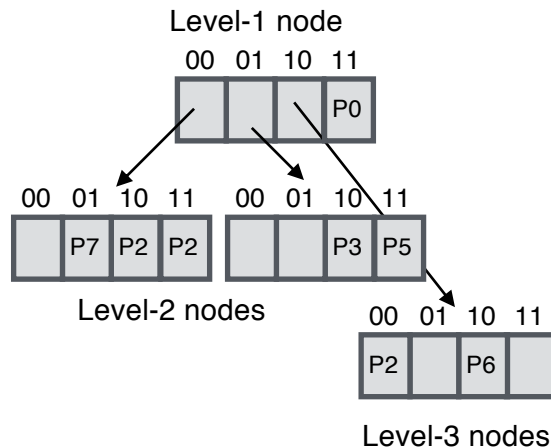


Lookups as DFGs

**Ethernet lookup:
hash table w/ 4
entries per bucket**

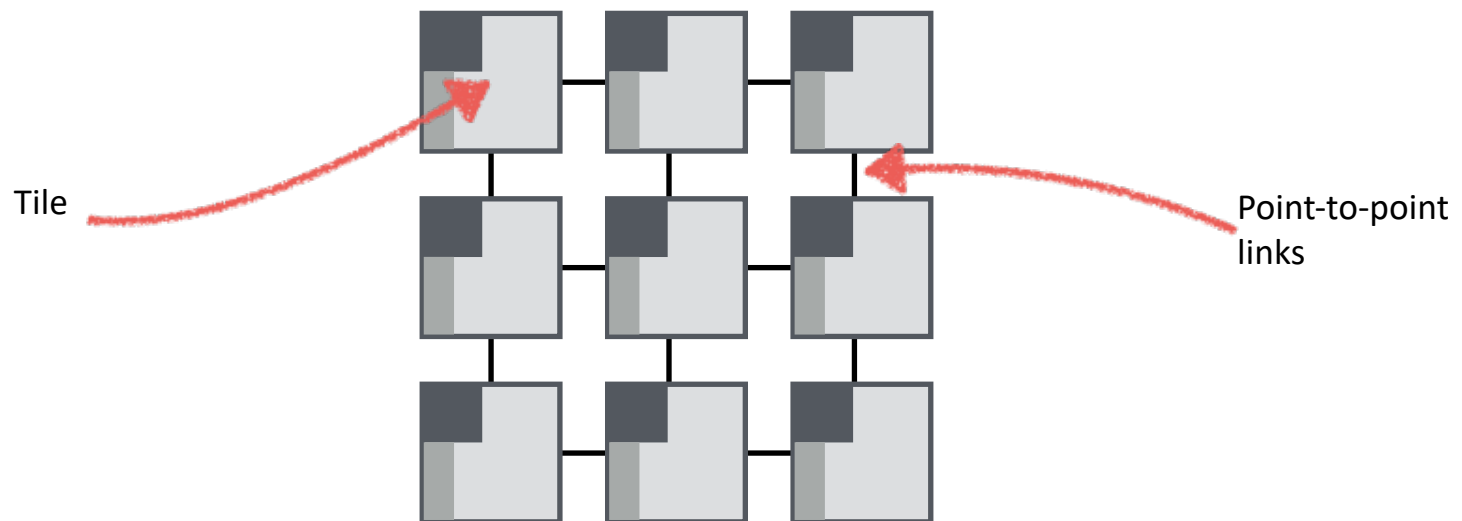


**IP forwarding:
multi-level
lookup trie**



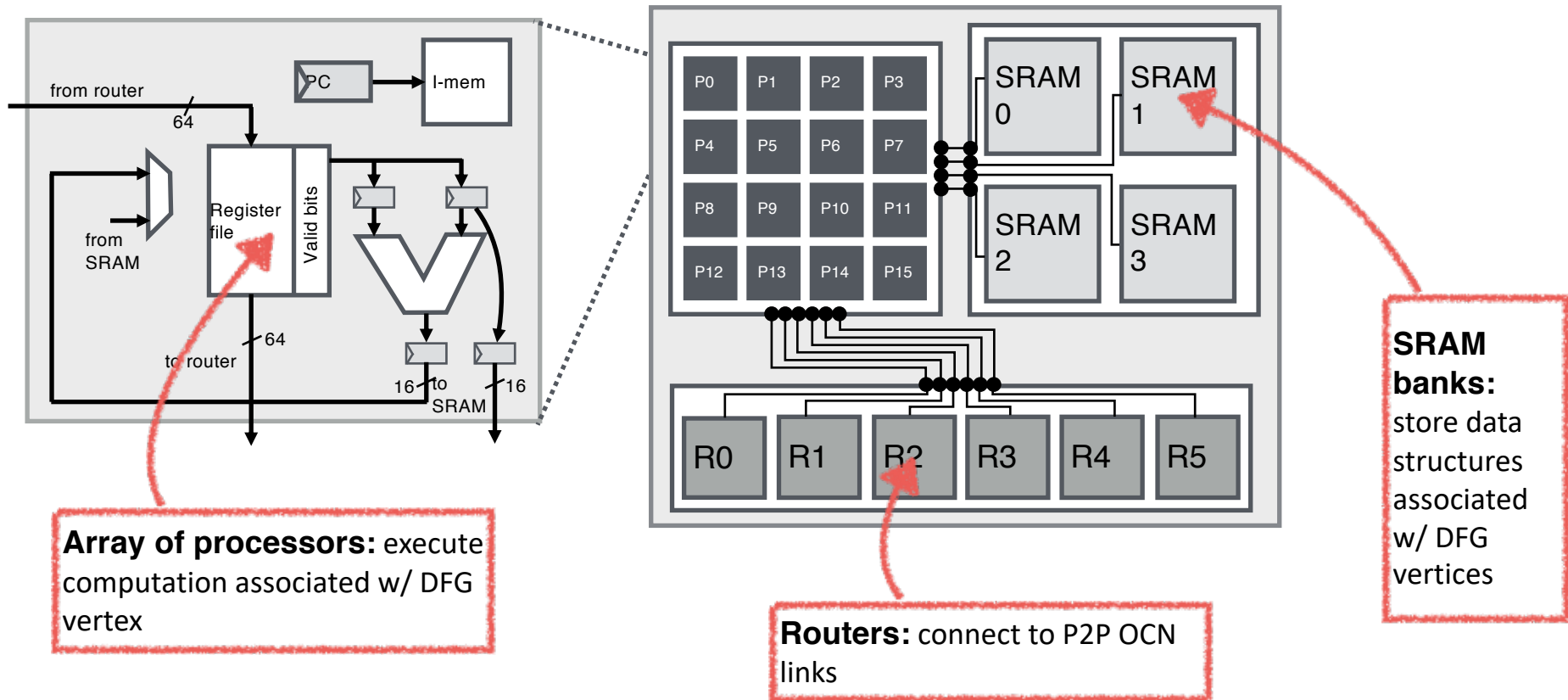
PLUG hardware

- Massively parallel HW architecture
- Matrix of elements (**tiles**), each including **memory** and **computation**, connected by **on-chip network**



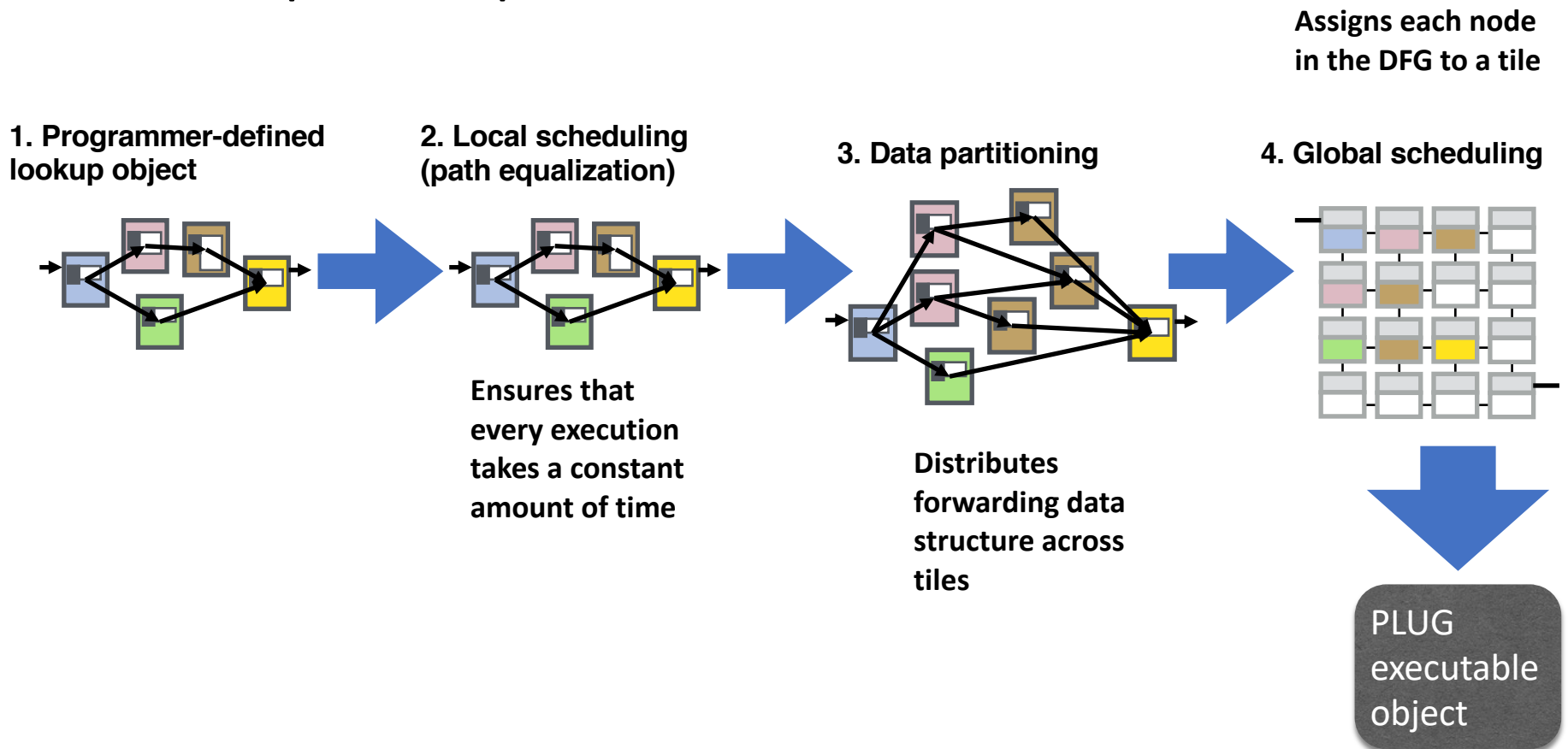
DFG nodes→tiles; edges→network links

Tile design



PLUG compiler

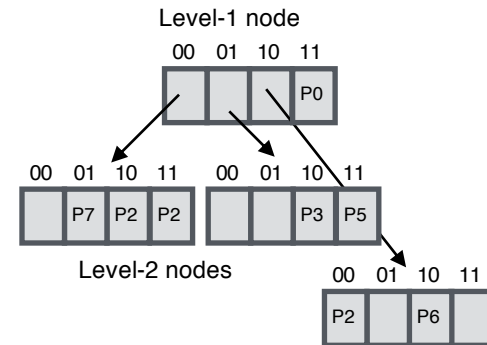
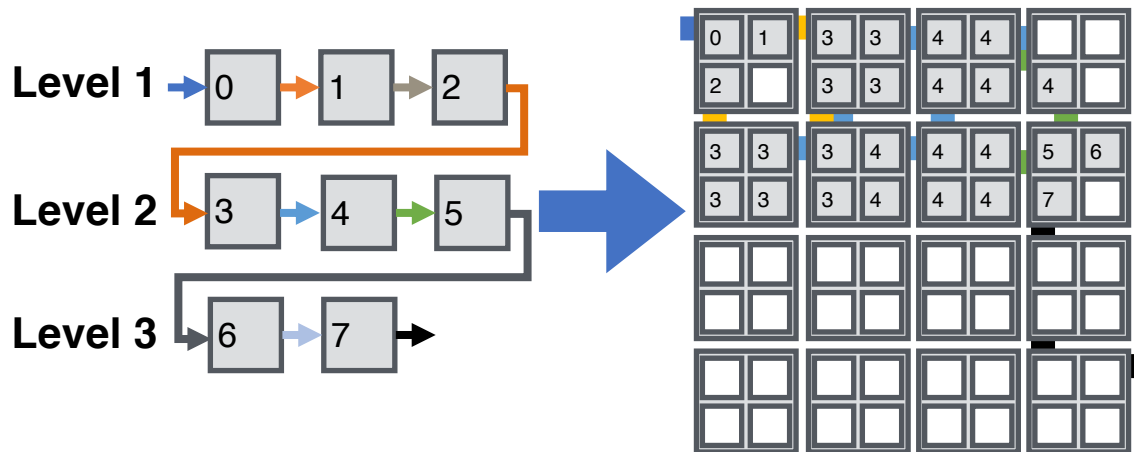
- Compilation process:



Some case studies

IPv4 forwarding

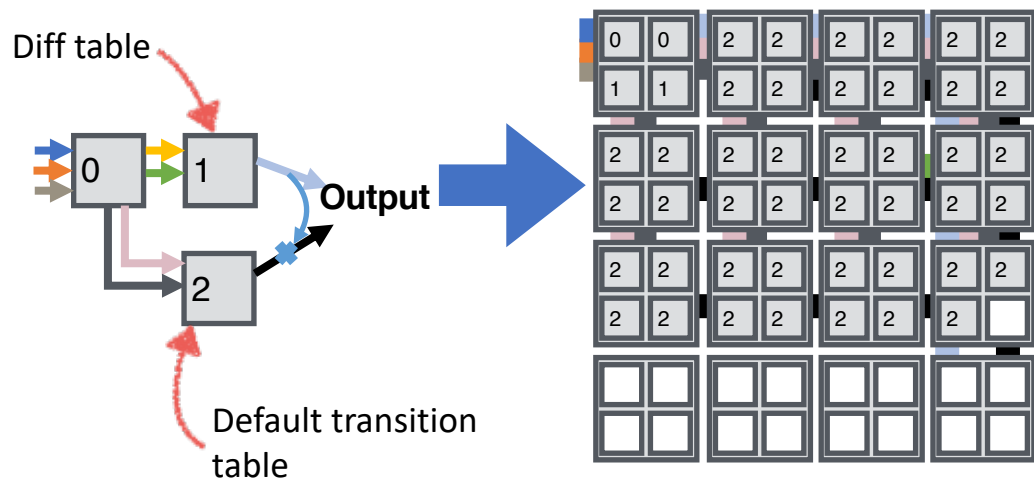
- **Three-level multibit trie w/ 16/8/8 stride**
- 3 pages per level: 2 for rule array, 1 for children ptrs



Throughput	1G lookup/s
Latency	90 ns
Power	0.35 W

DFA matching

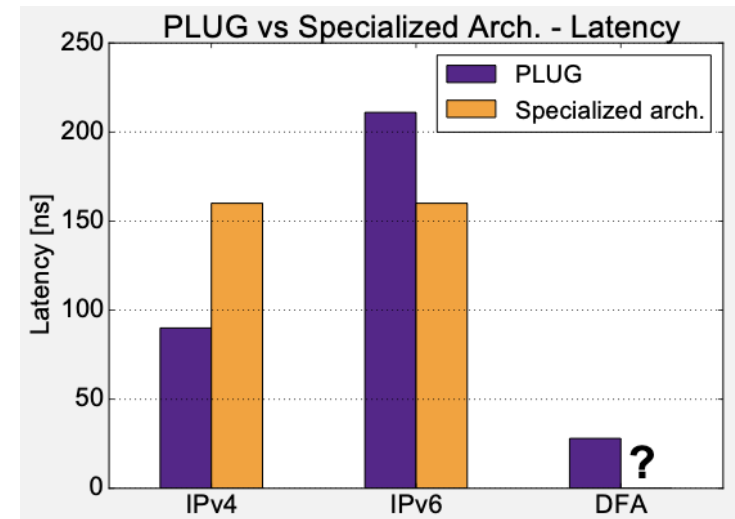
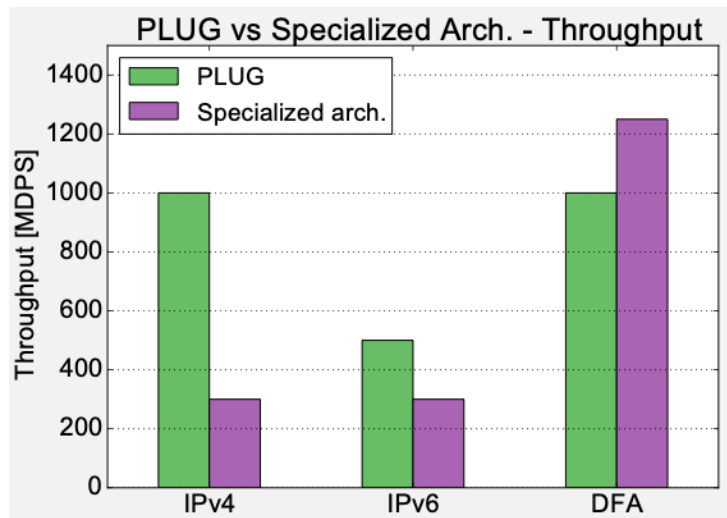
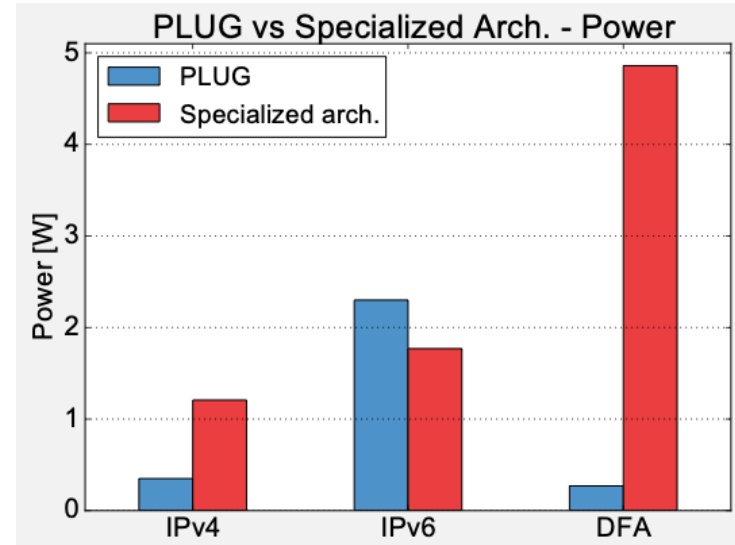
- D2FA compression w/ default transitions
 - Default transition table + diff table
- Loop runs on NPU, next-state lookup on PLUG



Throughput	1G lookup/s
Latency	28 ns
Power	0.27 W

Comparisons

- Applications:
 - IPv4/IPv6: **PLUG** vs **NetLogic NL9000**
 - Regex matching: **PLUG** vs **Tarari T10**



Also implemented

- Ethernet (layer-II forwarding)
- OpenFlow-style per-flow forwarding rules
- ...

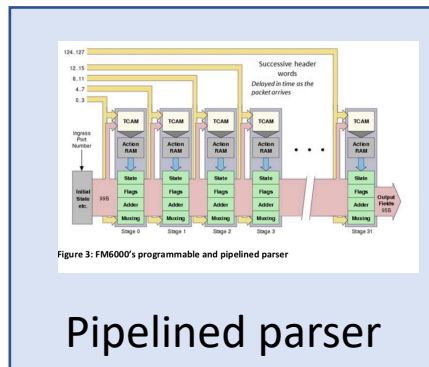
More flexible hardware: Intel FlexPipe

Intel FlexPipe

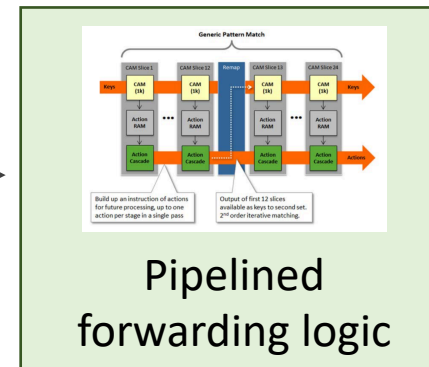
- **Flexible switching/routing hardware** from Intel
- Based on ideas similar to PLUG and other modern hardware:
 - **Multiple pipelined processing elements**
 - **Configurable processing elements** (can execute various network forwarding algorithms)
 - Each forwarding element has **small private memory**

Flexpipes – high level architecture

Input:
packet
data



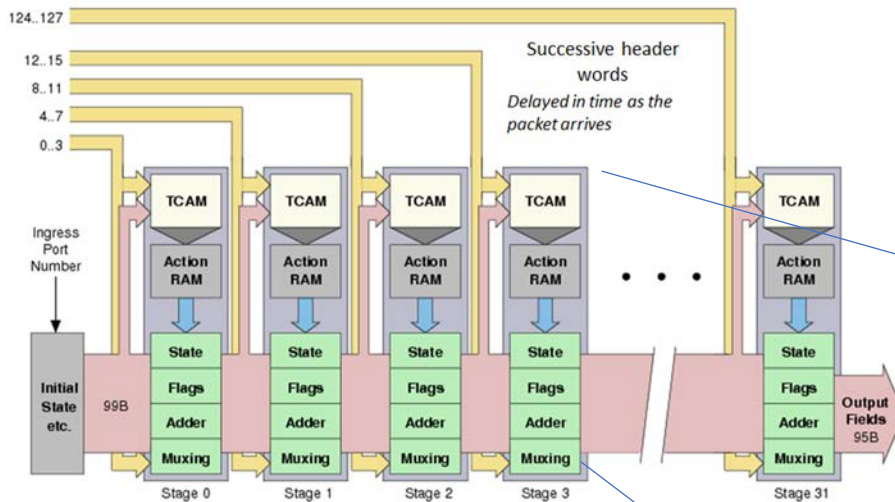
Pipelined parser



Pipelined
forwarding logic

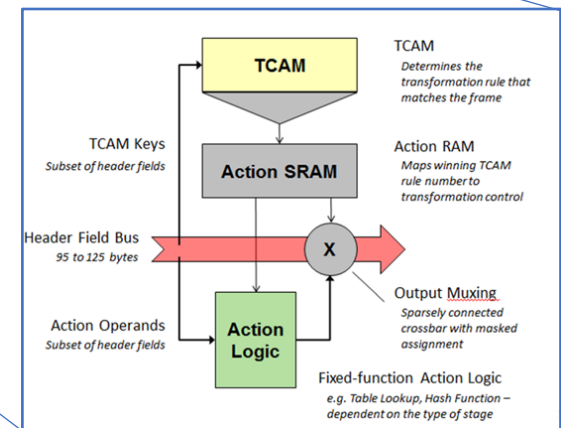
Output:
forwarding
action

Pipelined parsing unit



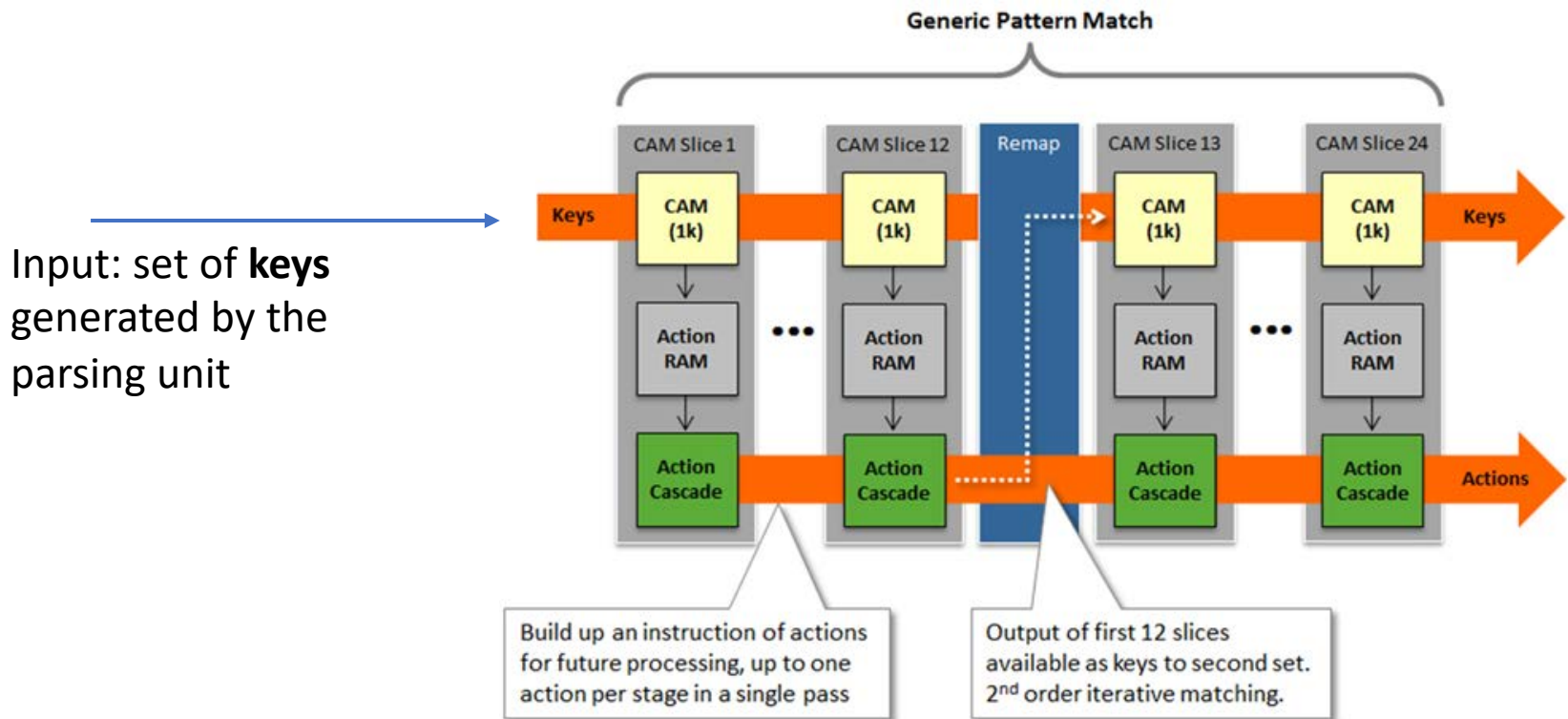
Each stage in the pipeline processes 4 bytes from the packet header and forwards the result to the next stage

Internally, each stage includes a **TCAM** that is looked up using the bytes of the header field – the output is an address in the action SRAM pointing to a **description of operations to perform**



Pipelined forwarding logic

- Organized similarly to the parsing unit



Additional features

- **BST-based lookup unit** (can look up rules using keys of up to 128 bits in length)
- MAC table (probably TCAM-based)

Where does flexibility come from?

- Specialized hardware which however is not bound to a particular algorithm
- Most forwarding algorithms require **fast associative memories** and **simple computations/transformations**
 - The designs we reviewed provide both!