Network Systems Design
(Agere Version)

Douglas Comer

Computer Science Department
Purdue University
250 N. University Street
West Lafayette, IN 47907-2066

http://www.cs.purdue.edu/people/comer

I

Course Introduction And Overview

Topic And Scope

The concepts, principles, and technologies that underlie the design of hardware and software systems used in computer networks and the Internet, focusing on the emerging field of network processors.

You Will Learn

- Review of
  - Network systems
  - Protocols and protocol processing tasks
- Hardware architectures for protocol processing
- Software-based network systems and software architectures
- Classification
  - Concept
  - Software and hardware implementations
- Switching fabrics

You Will Learn
(continued)

- Network processors: definition, architectures, and use
- Design tradeoffs and consequences
- Survey of commercial network processors
- Details of one example network processor
  - Architecture and instruction set(s)
  - Programming model and program optimization
  - Cross-development environment

What You Will NOT Learn

- EE details
  - VLSI technology and design rules
  - Chip interfaces: ICs and pin-outs
  - Waveforms, timing, or voltage
  - How to wire wrap or solder
- Economic details
  - Comprehensive list of vendors and commercial products
  - Price points

Background Required

- Basic knowledge of
  - Network and Internet protocols
  - Packet headers
- Basic understanding of hardware architecture
  - Registers
  - Memory organization
  - Typical instruction set
- Willingness to use an assembly language

Schedule Of Topics

- Quick review of basic networking
- Protocol processing tasks and classification
- Software-based systems using conventional hardware
- Special-purpose hardware for high speed
- Motivation and role of network processors
- Network processor architectures

Schedule Of Topics
(continued)

- An example network processor technology in detail
  - Hardware architecture and parallelism
  - Programming model
  - Testbed architecture and features
- Design tradeoffs
- Scaling a network processor
- Survey of network processor architectures

Course Administration

- Textbook
  - D. Comer, *Network Systems Design Using Network Processors*, Agere Version, Prentice Hall, 2005.
- Grade
  - Quizzes 5%
  - Midterm and final exam 35%
  - Programming projects 60%

Lab Facilities Available

- Extensive network processor testbed facilities
- Donations from
  - Agere Systems
  - IBM (now sold to Hifn)
  - Intel
- Includes hardware and cross-development software

What You Will Do In The Lab

- Write and compile software for an NP
- Download software into an NP
- Monitor the NP as it runs
- Interconnect Ethernet ports on an NP board
  - To other ports on other NP boards
  - To other computers in the lab
- Send Ethernet traffic to the NP
- Receive Ethernet traffic from the NP

Example Programming Projects

- A packet analyzer
  - IP datagrams
  - TCP segments
- An Ethernet bridge
- An IP fragmenter
- A classification program
- A bump-in-the-wire system using low-level packet processors

Questions?

A QUICK OVERVIEW

OF NETWORK PROCESSORS

The Network Systems Problem

- Data rates keep increasing
- Protocols and applications keep evolving
- System design is expensive
- System implementation and testing take too long
- Systems often contain errors
- Special-purpose hardware designed for one system cannot be reused

The Challenge

Find ways to improve the design and manufacture of complex networking systems.

The Big Questions

- What systems?
  - Everything we have now
  - New devices not yet designed
- What physical communication mechanisms?
  - Everything we have now
  - New communication systems not yet designed / standardized
- What speeds?
  - Everything we have now
  - New speeds much faster than those in use

More Big Questions

- What protocols?
  - Everything we have now
  - New protocols not yet designed / standardized
- What applications?
  - Everything we have now
  - New applications not yet designed / standardized

The Challenge (restated)

Find flexible, general technologies that enable rapid, low-cost design and manufacture of a variety of scalable, robust, efficient network systems that run a variety of existing and new protocols, perform a variety of existing and new functions for a variety of existing and new, higher-speed networks to support a variety of existing and new applications.

Special Difficulties

- Ambitious goal
- Vague problem statement
- Problem is evolving with the solution
- Pressure from
  - Changing infrastructure
  - Changing applications

Desiderata

- High speed
- Flexible and extensible to accommodate
  - Arbitrary protocols
  - Arbitrary applications
  - Arbitrary physical layer
- Low cost

Desiderata

- High speed
- Flexible and extensible to accommodate
  - Arbitrary protocols
  - Arbitrary applications
  - Arbitrary physical layer
- Low cost

Statement Of Hope
(1995 version)

If there is hope, it lies in ASIC designers.

Statement Of Hope
(1999 version)

???

If there is hope, it lies in ~~ASIC designers~~.

Statement Of Hope

(2004 version)

programmers!

If there is hope, it lies in ~~ASIC designers~~.

Programmability

- Key to low-cost hardware for next generation network systems
- More flexibility than ASIC designs
- Easier / faster to update than ASIC designs
- Less expensive to develop than ASIC designs
- What we need: a programmable device with more capability than a conventional CPU

The Idea In A Nutshell

- Devise new hardware building blocks
- Make them programmable
- Include support for protocol processing and I/O
  - General-purpose processor(s) for control tasks
  - Special-purpose processor(s) for packet processing and table lookup
- Include functional units for tasks such as checksum computation
- Integrate as much as possible onto one chip
- Call the result a *network processor*

The Rest Of The Course

- We will
  - Examine the general problem being solved
  - Survey some approaches vendors have taken
  - Explore possible architectures
  - Study example technologies
  - Consider how to implement systems using network processors

Disclaimer #1

In the field of network processors, I am a tyro.

Definition

Tyro \Ty’ro\, n.; pl. *Tyros*. A beginner in learning; one who is in the rudiments of any branch of study; a person imperfectly acquainted with a subject; a novice.

By Definition

In the field of network processors, you are all tyros.

In Our Defense

When it comes to network processors, everyone is a tyro.

Questions?

II

Basic Terminology And Example Systems (A Quick Review)

Packets Cells And Frames

- *Packet*
  - Generic term
  - Small unit of data being transferred
  - Travels independently
  - Upper and lower bounds on size

Packets Cells And Frames
(continued)

- *Cell*
  - Fixed-size packet (e.g., ATM)
- *Frame* or *layer-2 packet*
  - Packet understood by hardware
- *IP datagram*
  - Internet packet

Types Of Networks

- Paradigm
  - *Connectionless*
  - *Connection-oriented*
- Access type
  - *Shared* (i.e., multiaccess)
  - *Point-To-Point*

Connection-Oriented Networks

- Telephone paradigm (connection, use, disconnect)
- Examples
  - Frame Relay
  - Asynchronous Transfer Mode (ATM)

Point-To-Point Network

- Connects exactly two systems
- Often used for long distance
- Example: data circuit connecting two routers

Data Circuit

- Leased from phone company
- Also called *serial line* because data is transmitted bit-serially
- Originally designed to carry digital voice
- Cost depends on speed and distance
- T-series standards define low speeds (e.g. T1)
- STS and OC standards define high speeds

Digital Circuit Speeds

Standard Name	Bit Rate	Voice Circuits
–	0.064 Mbps	1
T1	1.544 Mbps	24
T3	44.736 Mbps	672
OC-1	51.840 Mbps	810
OC-3	155.520 Mbps	2430
OC-12	622.080 Mbps	9720
OC-24	1,244.160 Mbps	19440
OC-48	2,488.320 Mbps	38880
OC-192	9,953.280 Mbps	155520
OC-768	39,813.120 Mbps	622080

Digital Circuit Speeds

Standard Name	Bit Rate	Voice Circuits
–	0.064 Mbps	1
T1	1.544 Mbps	24
T3	44.736 Mbps	672
OC-1	51.840 Mbps	810
OC-3	155.520 Mbps	2430
OC-12	622.080 Mbps	9720
OC-24	1,244.160 Mbps	19440
OC-48	2,488.320 Mbps	38880
OC-192	9,953.280 Mbps	155520
OC-768	39,813.120 Mbps	622080

- Holy grail of networking: devices capable of accepting and forwarding data at 10 Gbps (OC-192).

Local Area Networks

- Ethernet technology dominates
- Layer 1 standards
  - Media and wiring
  - Signaling
  - Handled by dedicated interface chips
  - Unimportant to us
- Layer 2 standards
  - MAC framing and addressing

MAC Addressing

- Three address types
  - Unicast (single computer)
  - Broadcast (all computers in broadcast domain)
  - Multicast (some computers in broadcast domain)

More Terminology

- *Internet*
  - Interconnection of multiple networks
  - Allows heterogeneity of underlying networks
- Network scope
  - *Local Area Network (LAN)* covers limited distance
  - *Wide Area Network (WAN)* covers arbitrary distance

Network System

- Individual hardware component
- Serves as fundamental building block
- Used in networks and internets
- May contain processor and software
- Operates at one or more layers of the protocol stack

Example Network Systems

- Layer 2
  - Bridge
  - Ethernet switch
  - VLAN switch

VLAN Switch

- Similar to conventional layer 2 switch
  - Connects multiple computers
  - Forwards frames among them
  - Each computer has unique *unicast address*
- Differs from conventional layer 2 switch
  - Allows manager to configure *broadcast domains*
- Broadcast domain known as *virtual* network

Broadcast Domain

- Determines propagation of broadcast / multicast
- Originally corresponded to fixed hardware
  - One per cable segment
  - One per hub or switch
- Now configurable via VLAN switch
  - Manager assigns ports to VLANs

Example Network Systems
(continued)

- Layer 3
  - Internet host computer
  - IP router (layer 3 switch)
- Layer 4
  - Basic Network Address Translator (NAT)
  - Round-robin Web load balancer
  - TCP terminator

Example Network Systems
(continued)

- Layer 5
  - Firewall
  - Intrusion Detection System (IDS)
  - Virtual Private Network (VPN)
  - Softswitch running SIP
  - Application gateway
  - TCP splicer (also known as NAPT — Network Address and Protocol Translator)
  - Smart Web load balancer
  - Set-top box

Example Network Systems
(continued)

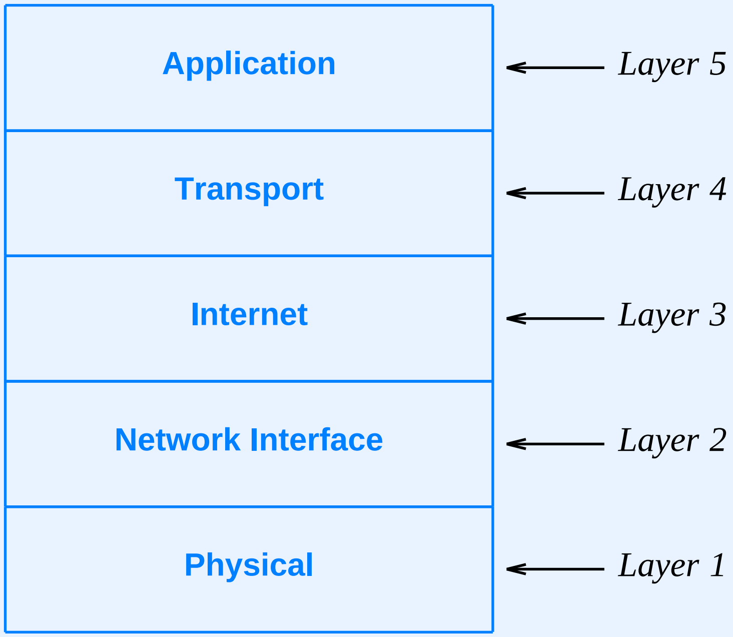
- Network control systems
  - Packet / flow analyzer
  - Traffic monitor
  - Traffic policer
  - Traffic shaper

Questions?

III

Review Of Protocols And Packet Formats

Protocol Layering



Application ← *Layer 5*

Transport ← *Layer 4*

Internet ← *Layer 3*

Network Interface ← *Layer 2*

Physical ← *Layer 1*

- Five-layer Internet reference model
- Multiple protocols can occur at each layer

Layer 2 Protocols

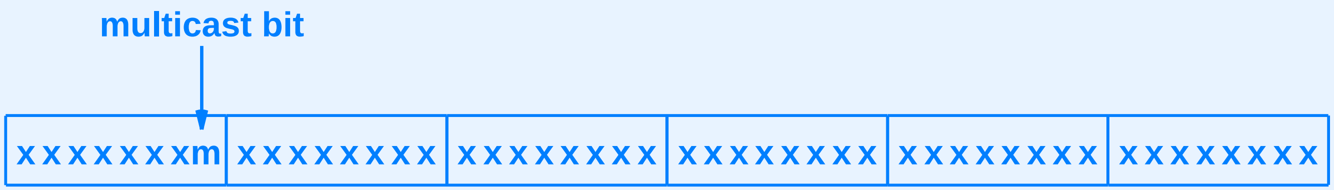
- Two protocols are important
  - Ethernet (widely used)
  - ATM (defines per-flow QoS)
- We will concentrate on Ethernet

Ethernet Addressing

- 48-bit addressing
- Unique address assigned to each station (NIC)
- Destination address in each packet can specify delivery to
  - A single computer (unicast)
  - All computers in broadcast domain (broadcast)
  - Some computers in broadcast domain (multicast)

Ethernet Addressing
(continued)

- Broadcast address is all 1s
- Single bit determines whether remaining addresses are unicast or multicast

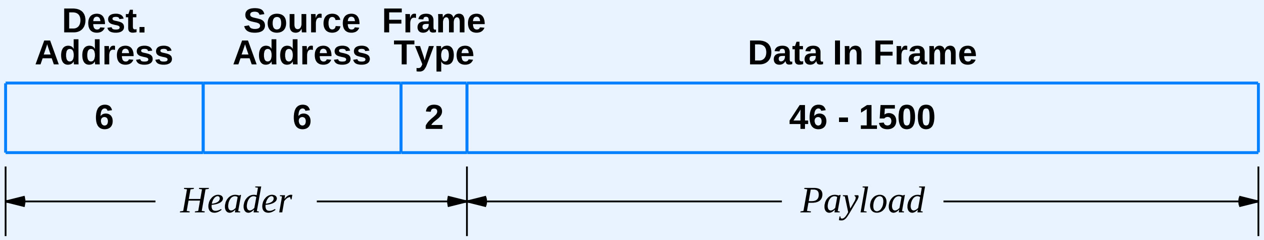


multicast bit

xxxxxxxm	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx

- Multicast bit travels first on the wire

Ethernet Frame Processing



Dest. Address	Source Address	Frame Type	Data In Frame
6	6	2	46 - 1500

Header | *Payload*

- Dedicated physical layer hardware
  - Checks and removes preamble and CRC on input
  - Computes and appends CRC and preamble on output
- Layer 2 systems use source, destination and (possibly) type fields

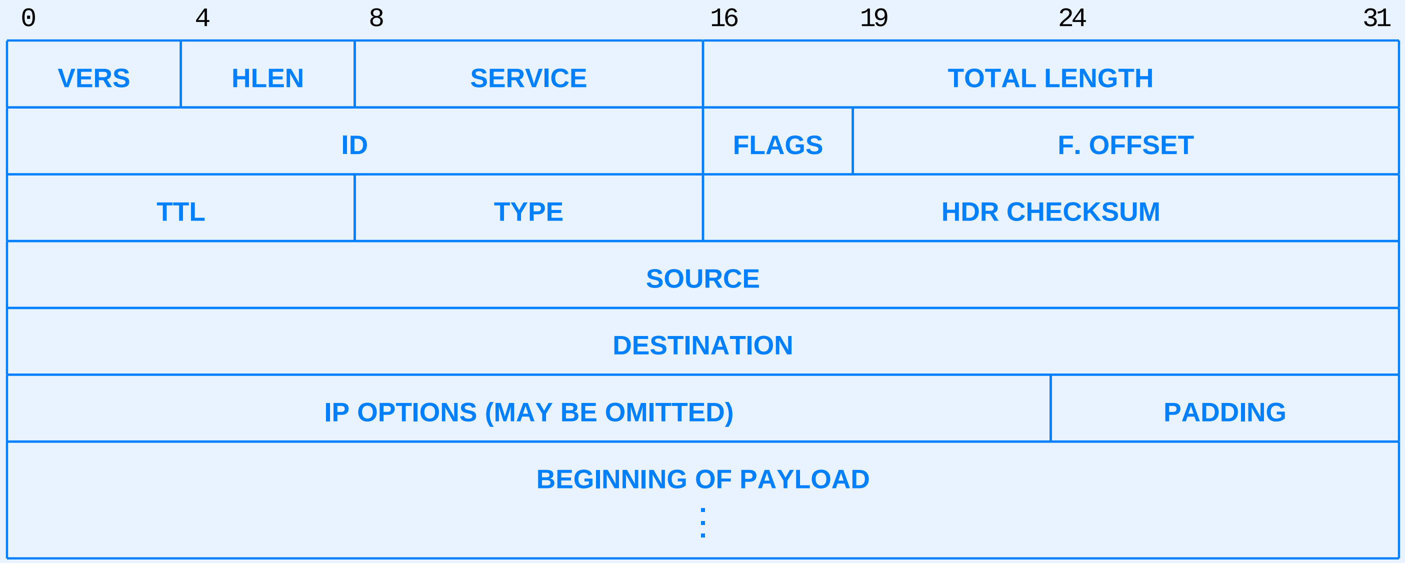
Internet

- Set of (heterogeneous) computer networks interconnected by *IP routers*
- End-user computers, called *hosts*, each attach to specific network
- Protocol software
  - Runs on both hosts and routers
  - Provides illusion of homogeneity

Internet Protocols Of Interest

- Layer 2
  - Address Resolution Protocol (ARP)
- Layer 3
  - Internet Protocol (IP)
- Layer 4
  - User Datagram Protocol (UDP)
  - Transmission Control Protocol (TCP)

IP Datagram Format


0 4 8 16 19 24 31

VERS HLEN SERVICE TOTAL LENGTH

ID FLAGS F. OFFSET

TTL TYPE HDR CHECKSUM

SOURCE

DESTINATION

IP OPTIONS (MAY BE OMITTED) PADDING

BEGINNING OF PAYLOAD

⋮

- Format of each packet sent across Internet
- Fixed-size fields make parsing efficient

IP Datagram Fields

Field	Meaning
VERS	Version number of IP being used (4)
HLEN	Header length measured in 32-bit units
SERVICE	Level of service desired
TOTAL LENGTH	Datagram length in octets including header
ID	Unique value for this datagram
FLAGS	Bits to control fragmentation
F. OFFSET	Position of fragment in original datagram
TTL	Time to live (hop countdown)
TYPE	Contents of payload area
HDR CHECKSUM	One’s-complement checksum over header
SOURCE	IP address of original sender
DESTINATION	IP address of ultimate destination
IP OPTIONS	Special handling parameters
PADDING	To make options a 32-bit multiple

IP addressing

- 32-bit Internet address assigned to each computer
- Virtual, hardware independent value
- Prefix identifies network; suffix identifies host
- Network systems use an address mask to specify the boundary between prefix and suffix

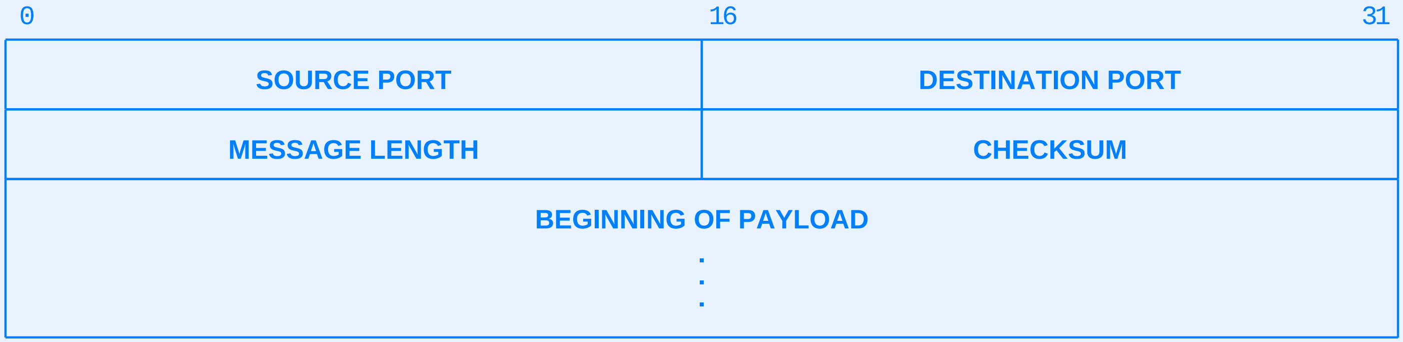
Next-Hop Forwarding

- Routing table
  - Found in both hosts and routers
  - Stores (destination, mask, next_hop) tuples
- Route lookup
  - Takes destination address as argument
  - Finds next hop
  - Uses longest-prefix match

Next-Hop Forwarding

- Routing table
  - Found in both hosts and routers
  - Stores (destination, mask, next_hop) tuples
- Route lookup
  - Takes destination address as argument
  - Finds next hop
  - Uses longest-prefix match

UDP Datagram Format



0 16 31

SOURCE PORT

DESTINATION PORT

MESSAGE LENGTH

CHECKSUM

BEGINNING OF PAYLOAD

. . .

Field	Meaning
SOURCE PORT	ID of sending application
DESTINATION PORT	ID of receiving application
MESSAGE LENGTH	Length of datagram including the header
CHECKSUM	One’s-complement checksum over entire datagram

TCP Segment Format

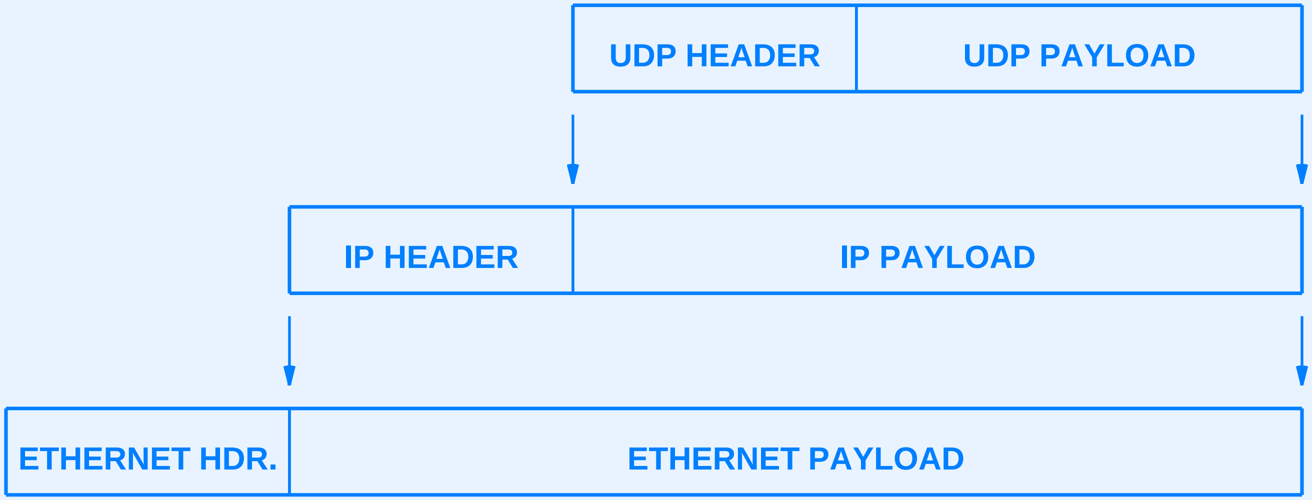
<table>
<tr><th>0</th><th>4</th><th>10</th><th>16</th><th>24</th><th>31</th></tr>
<tr><td colspan="3">SOURCE PORT</td><td colspan="3">DESTINATION PORT</td></tr>
<tr><td colspan="6">SEQUENCE</td></tr>
<tr><td colspan="6">ACKNOWLEDGEMENT</td></tr>
<tr><td>HLEN</td><td>NOT USED</td><td>CODE BITS</td><td colspan="3">WINDOW</td></tr>
<tr><td colspan="3">CHECKSUM</td><td colspan="3">URGENT PTR</td></tr>
<tr><td colspan="4">OPTIONS (MAY BE OMITTED)</td><td colspan="2">PADDING</td></tr>
<tr><td colspan="6">BEGINNING OF PAYLOAD
⋮</td></tr>
</table>

- Sent end-to-end
- Fixed-size fields make parsing efficient

TCP Segment Fields

Field	Meaning
SOURCE PORT	ID of sending application
DESTINATION PORT	ID of receiving application
SEQUENCE	Sequence number for data in payload
ACKNOWLEDGEMENT	Acknowledgement of data received
HLEN	Header length measured in 32-bit units
NOT USED	Currently unassigned
CODE BITS	URGENT, ACK, PUSH, RESET, SYN, FIN
WINDOW	Receiver’s buffer size for additional data
CHECKSUM	One’s-complement checksum over entire segment
URGENT PTR	Pointer to urgent data in segment
OPTIONS	Special handling
PADDING	To make options a 32-bit multiple

Illustration Of Encapsulation



- Field in each header specifies type of encapsulated packet

Example ARP Packet Format

<table>
<tr><th>0</th><th>8</th><th>16</th><th>24 31</th></tr>
<tr><td colspan="2">ETHERNET ADDRESS TYPE (1)</td><td colspan="2">IP ADDRESS TYPE (0800)</td></tr>
<tr><td>ETH ADDR LEN (6)</td><td>IP ADDR LEN (4)</td><td colspan="2">OPERATION</td></tr>
<tr><td colspan="4">SENDER’S ETH ADDR (first 4 octets)</td></tr>
<tr><td colspan="2">SENDER’S ETH ADDR (last 2 octets)</td><td colspan="2">SENDER’S IP ADDR (first 2 octets)</td></tr>
<tr><td colspan="2">SENDER’S IP ADDR (last 2 octets)</td><td colspan="2">TARGET’S ETH ADDR (first 2 octets)</td></tr>
<tr><td colspan="4">TARGET’S ETH ADDR (last 4 octets)</td></tr>
<tr><td colspan="4">TARGET’S IP ADDR (all 4 octets)</td></tr>
</table>

- Format when ARP used with Ethernet and IP
- Each Ethernet address is six octets
- Each IP address is four octets

End Of Review

Questions?

IV

Conventional Computer Hardware Architecture

Software-Based Network System

- Uses conventional hardware (e.g., PC)
- Software
  - Runs the entire system
  - Allocates memory
  - Controls I/O devices
  - Performs all protocol processing

Why Study Protocol Processing On Conventional Hardware?

- Past
  - Employed in early IP routers
  - Many algorithms developed / optimized for conventional hardware
- Present
  - Used in low-speed network systems
  - Easiest to create / modify
  - Costs less than special-purpose hardware

Why Study Protocol Processing On Conventional Hardware? (continued)

- Future
  - Processors continue to increase in speed
  - Some conventional hardware present in all systems

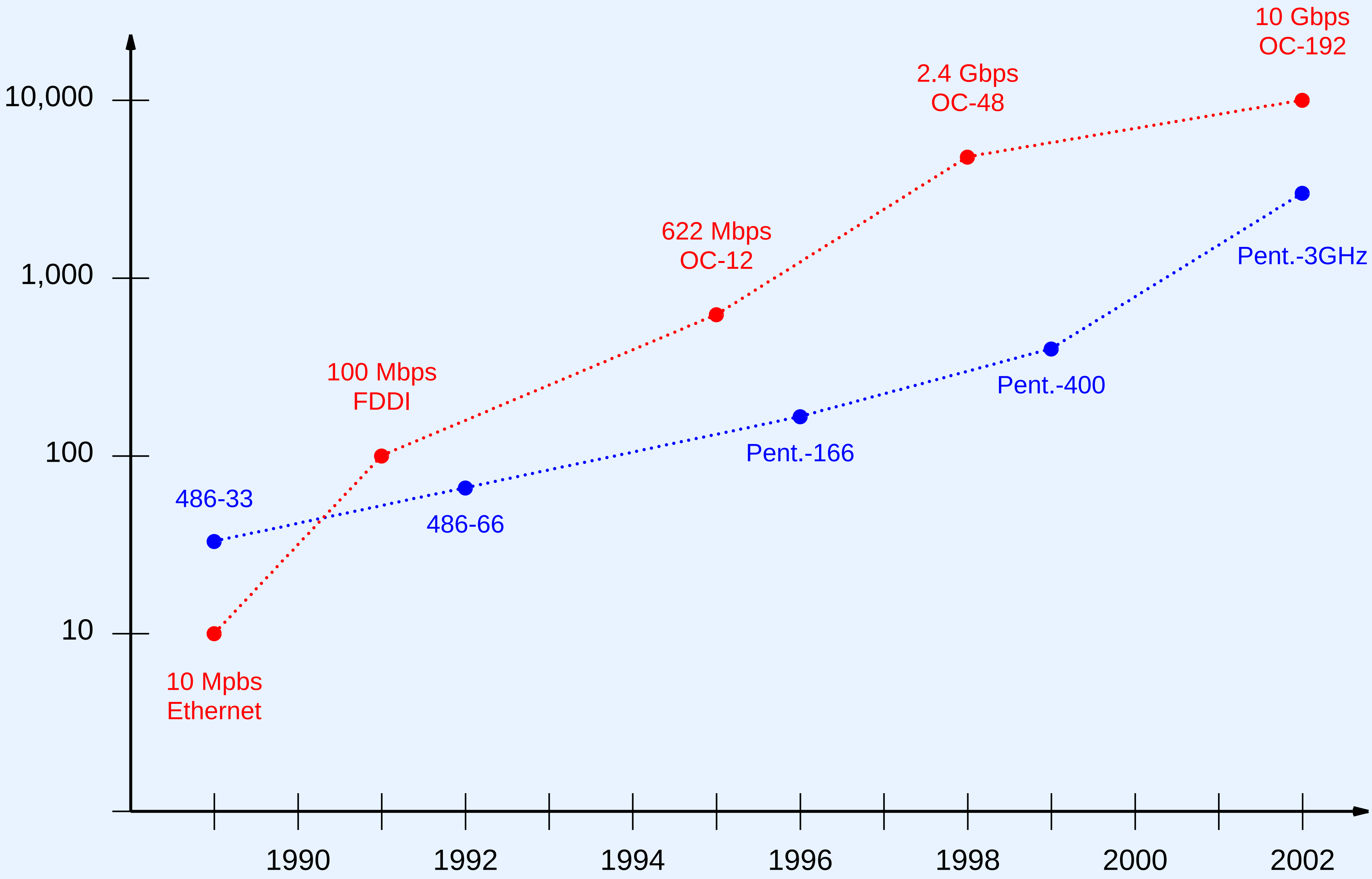
Why Study Protocol Processing On Conventional Hardware? (continued)

- Future
  - Processors continue to increase in speed
  - Some conventional hardware present in all systems
  - You will build software-based systems in lab!

Serious Question

- Which is growing faster?
  - Processing power
  - Network bandwidth
- Note: if network bandwidth growing faster
  - Need special-purpose hardware
  - Conventional hardware will become irrelevant

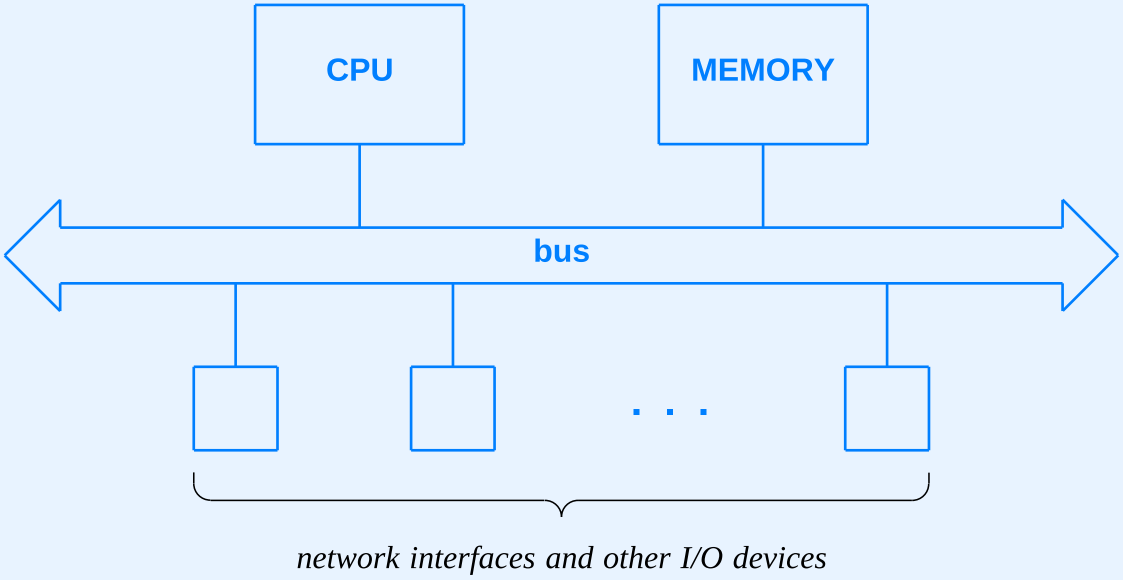
Growth Of Technologies



Conventional Computer Hardware

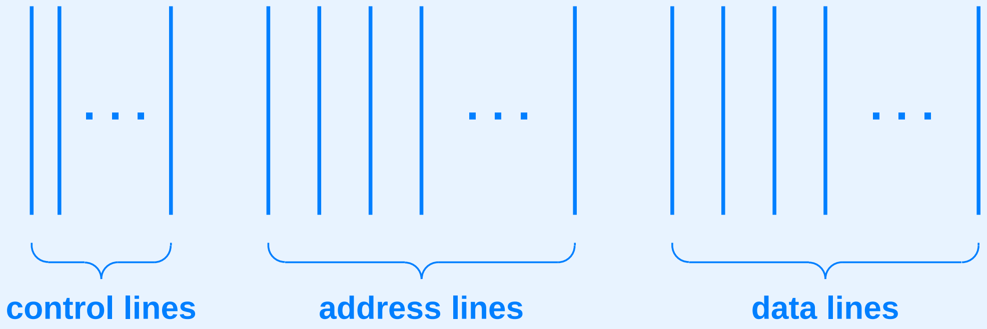
- Four important aspects
  - Processor
  - Memory
  - I/O interfaces
  - One or more buses

Illustration Of Conventional Computer Architecture



- Bus is central, shared interconnect
- All components *contend* for use

Bus Organization And Operations



- Parallel wires (*C*+*A*+*D* total)
- Used to pass
  - Control information (*C* bits)
  - An address (*A* bits)
  - A data value (*D* bits)

Bus Width

- Number of parallel data bits known as *width* of bus
- Wider bus
  - Transfers more data per unit time
  - Costs more
  - Requires more physical space
- Compromise: to simulate wider bus, use hardware that multiplexes transfers

Bus Paradigm

- Only two basic operations
  - Fetch
  - Store
- All operations cast as forms of the above

Fetch/Store

- Fundamental paradigm
- Used throughout hardware, including network processors

Fetch Operation

- Place address of a device on address lines
- Issue *fetch* on control lines
- Use control lines to wait for device that owns the address to respond
- If operation successful, extract value (response) from data lines
- If not successful, report error

Store Operation

- Place address of a device on address lines
- Place value on data lines
- Issue *store* on control lines
- Use control lines to wait for device that owns the address to respond
- If operation does not succeed, report error

Example Of Operations Mapped Into Fetch/Store Paradigm

- Imagine disk device attached to a bus
- Assume disk hardware supports three (nontransfer) operations:
  - Start disk spinning
  - Stop disk
  - Determine current status

Example Of Operations Mapped Into Fetch/Store Paradigm (continued)

- Assign the disk two contiguous bus addresses D and D+1
- Arrange for store of nonzero to address D to start disk spinning
- Arrange for store of zero to address D to stop disk
- Arrange for fetch from address D+1 to return current status
- Note: effect of store to address D+1 can be defined as
  - Appears to work, but has no effect
  - Returns an error

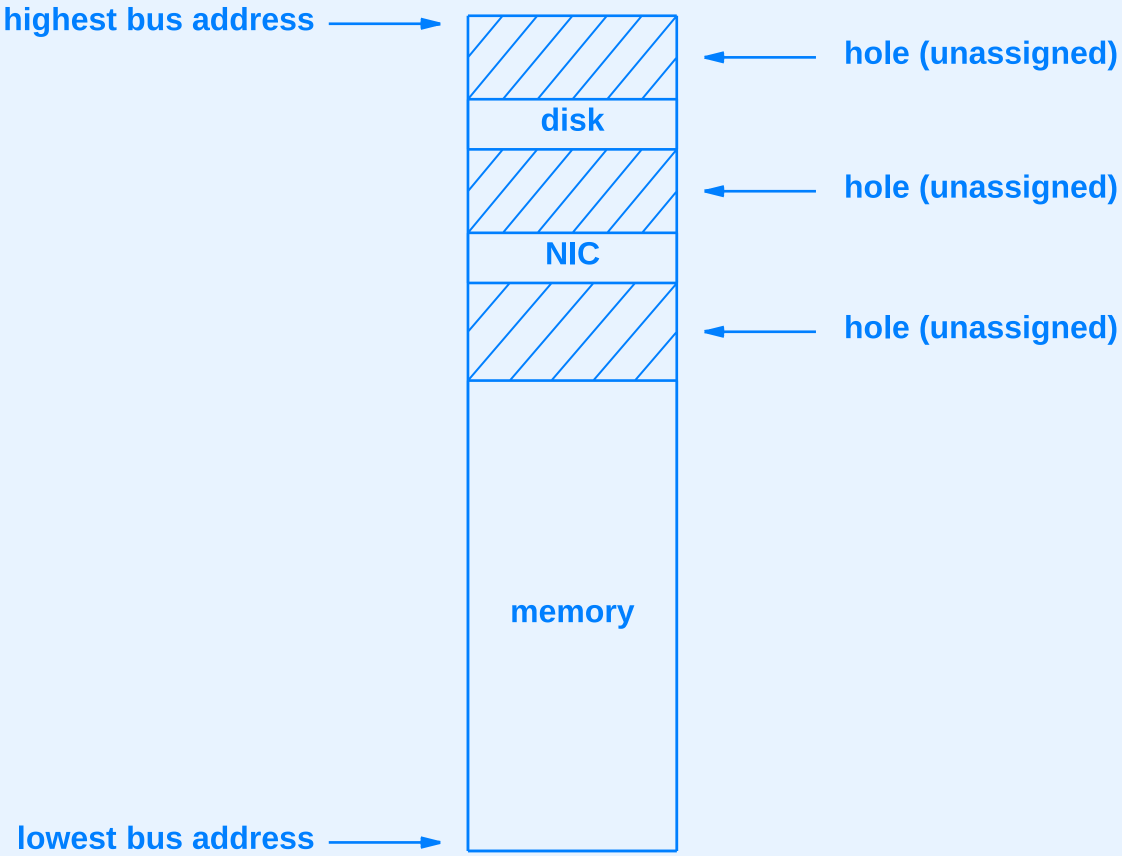
Bus Address Space

- Arbitrary hardware can be attached to bus
- K address lines result in 2^k possible bus addresses
- Address can refer to
  - Memory (e.g., RAM or ROM)
  - I/O device
- Arbitrary devices can be placed at arbitrary addresses
- Address space can contain ‘‘holes’’

Bus Address Terminology

- Device on bus known as *memory mapped* I/O
- Locations that correspond to nontransfer operations known as *Control and Status Registers (CSRs)*

Example Bus Address Space



Network I/O On
Conventional Hardware

- Network Interface Card (NIC)
  - Attaches between bus and network
  - Operates like other I/O devices
  - Handles electrical/optical details of network
  - Handles electrical details of bus
  - Communicates over bus with CPU or other devices

Making Network I/O Fast

- Key idea: migrate more functionality onto NIC
- Four techniques used with bus
  - Onboard address recognition & filtering
  - Onboard packet buffering
  - Direct Memory Access (DMA)
  - Operation and buffer chaining

Onboard Address Recognition And Filtering

- NIC given set of addresses to accept
  - Station’s unicast address
  - Network broadcast address
  - Zero or more multicast addresses
- When packet arrives, NIC checks destination address
  - Accept packet if address on list
  - Discard others

Onboard Packet Buffering

- NIC given high-speed local memory
- Incoming packet placed in NIC’s memory
- Allows computer’s memory/bus to operate slower than network
- Handles small packet bursts

Direct Memory Access (DMA)

- CPU
  - Allocates packet buffer in memory
  - Passes buffer address to NIC
  - Goes on with other computation
- NIC
  - Accepts incoming packet from network
  - Copies packet over bus to buffer in memory
  - Informs CPU that packet has arrived

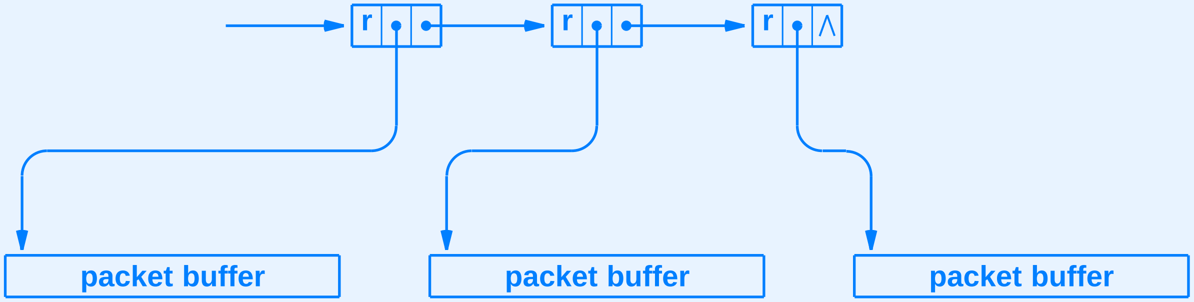
Buffer Chaining

- CPU
  - Allocates multiple buffers
  - Passes linked list to NIC
- NIC
  - Receives next packet
  - Divides into one or more buffers
- Advantage: a buffer can be smaller than a packet

Operation Chaining

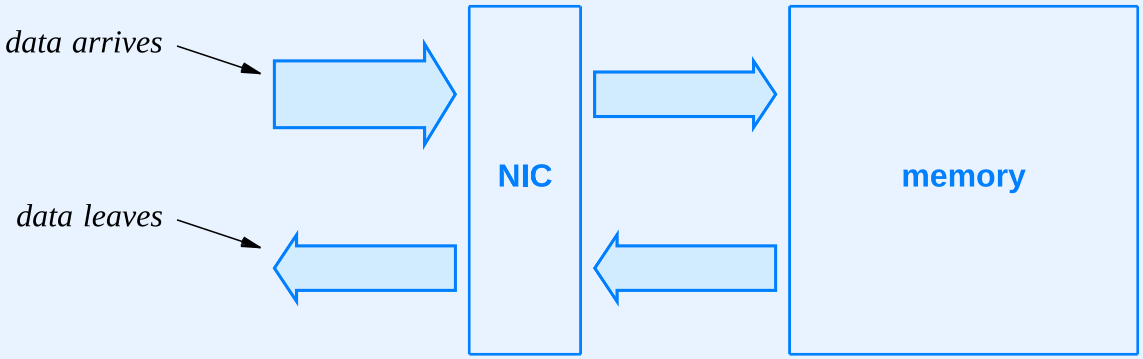
- CPU
  - Allocates multiple buffers
  - Builds linked list of operations
  - Passes list to NIC
- NIC
  - Follows list and performs instructions
  - Interrupts CPU after each operation
- Advantage: multiple operations proceed without CPU intervention

Illustration Of
Operation Chaining



- Optimizes movement of data to memory

Data Flow Diagram



- Depicts flow of data through hardware units
- Size of arrow represents throughput
- Used throughout the course and text

Summary

- Software-based network systems run on conventional hardware
  - Processor
  - Memory
  - I/O devices
  - Bus
- Network interface cards can be optimized to reduce CPU load

Questions?

V

Basic Packet Processing: Algorithms And Data Structures

Copying

- Used when packet moved from one memory location to another
- Expensive
- Must be avoided whenever possible
  - Leave packet in buffer
  - Pass buffer address among threads/layers

Possibilities For Buffer Allocation

- Fixed-sze buffers
  - Large enough for largest packet
    - Small, with bultiple buffers linked together for large packets
- Variable-size buffers

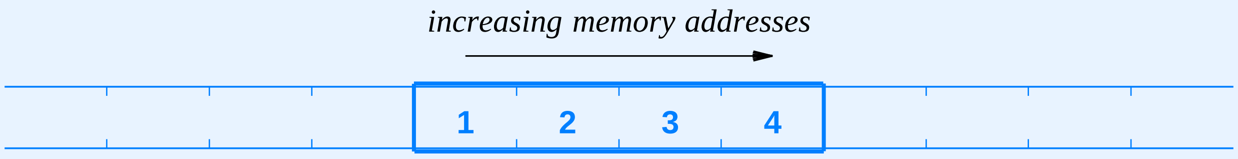
Buffer Addressing

- Buffer address must be resolvable in all contexts
- Easiest implementation: keep buffers in kernel space

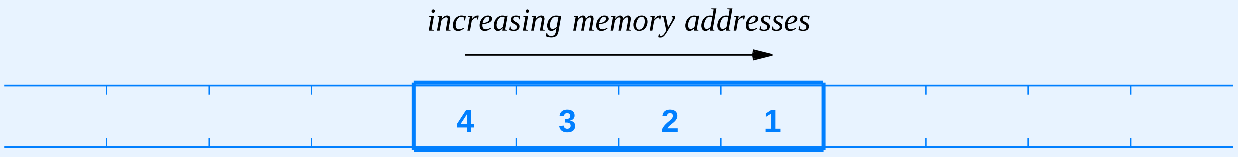
Integer Representation

- Two standards
  - Little endian (least-significant byte at lowest address)
  - Big endian (most-significant byte at lowest address)

Illustration Of Big And Little Endian Integers



little endian



big endian

Integer Conversion

- Needed when heterogeneous computers communicate
- Protocols define *network byte order*
- Computers convert to network byte order
- Typical library functions

Function	data size	Translation
ntohs	16 bits	Network byte order to host’s byte order
htons	16 bits	Host’s byte order to network byte order
ntohl	32 bits	Network byte order to host’s byte order
htonl	32 bits	Host’s byte order to network byte order

Examples Of Algorithms Implemented With Software-Based Systems

- Layer 2
  - Ethernet bridge
- Layer 3
  - IP forwarding
  - IP fragmentation and reassembly
- Layer 4
  - TCP connection recognition and splicing
- Other
  - Hash table lookup

Why Study These Algorithms?

Why Study These Algorithms?

- Provide insight to packet processing tasks

Why Study These Algorithms?

- Provide insight to packet processing tasks
- Reinforce concepts

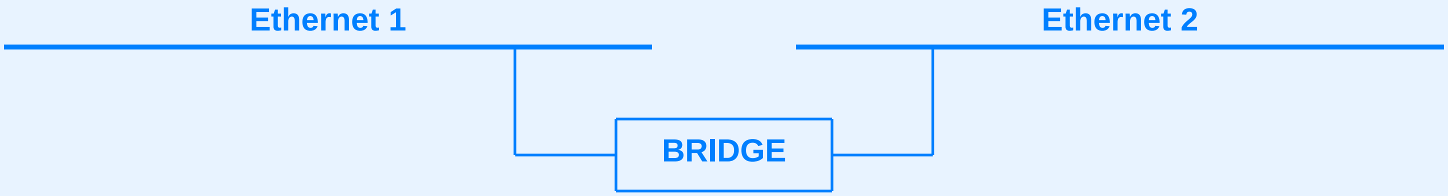
Why Study These Algorithms?

- Provide insight to packet processing tasks
- Reinforce concepts
- Help students recall protocol details

Why Study These Algorithms?

- Provide insight to packet processing tasks
- Reinforce concepts
- Help students recall protocol details
- You will need them in lab!

Ethernet Bridge



- Used between a pair of Ethernets
- Provides transparent, layer 2 connection
- Listens in promiscuous mode
- Forwards frames in both directions
- Uses addresses to filter

Bridge Filtering

- Uses source address in frames to identify computers on each network
- Uses destination address to decide whether to forward frame

Bridge Algorithm

```
Assume: two network interfaces each operating in promiscuous mode.
Create an empty list, L, that will contain pairs of values;
Do forever {
    Acquire the next frame to arrive;
    Set I to the interface over which the frame arrived;
    Extract the source address, S;
    Extract the destination address, D;
    Add the pair (S, I) to list L if not already present.
    If the pair (D, I) appears in list L {
        Drop the frame;
    } Else {
        Forward the frame over the other interface;
    }
}
```

Implementation Of Table Lookup

- Need high speed (more on this later)
- Software-based systems typically use *hashing* for table lookup

Hashing

- Optimizes number of *probes*
- Works well if table not full
- Practical technique: *double hashing*

Hashing Algorithm

```
Given: a key, a table in memory, and the table size N.
Produce: a slot in the table that corresponds to the key
    or an empty table slot if the key is not in the table.
Method: double hashing with open addressing.
Choose $P_1$ and $P_2$ to be prime numbers;
Fold the key to produce an integer, K;
Compute table pointer Q equal to $(P_1 \times K)$ modulo N;
Compute increment R equal to $(P_2 \times K)$ modulo N;
While (table slot Q not equal to K and nonempty) {
    $Q \leftarrow (Q + R)$ modulo N;
}
At this point, Q either points to an empty table slot or to the
    slot containing the key.
```

Address Lookup

- Computer can compare integer in one operation
- Network address can be longer than integer (e.g., 48 bits)
- Two possibilities
  - Use multiple comparisons per probe
  - Fold address into integer key

Folding

- Maps N-bit value into M-bit key, M < N
- Typical technique: exclusive or
- Potential problem: two values map to same key
- Solution: compare full value when key matches

IP Forwarding

- Used in hosts as well as routers
- Conceptual mapping

$$(\text{next hop, interface}) \leftarrow f(\text{datagram, routing table})$$

- Table driven

IP Routing Table

- One entry per destination
- Entry contains
  - 32-bit IP address of destination
  - 32-bit address mask
  - 32-bit next-hop address
  - N-bit interface number

Example IP Routing Table

Destination Address	Address Mask	Next-Hop Address	Interface Number
192.5.48.0	255.255.255.0	128.210.30.5	2
128.10.0.0	255.255.0.0	128.210.141.12	1
0.0.0.0	0.0.0.0	128.210.30.5	2

- Values stored in binary
- Interface number is for internal use only
- Zero mask produces *default* route

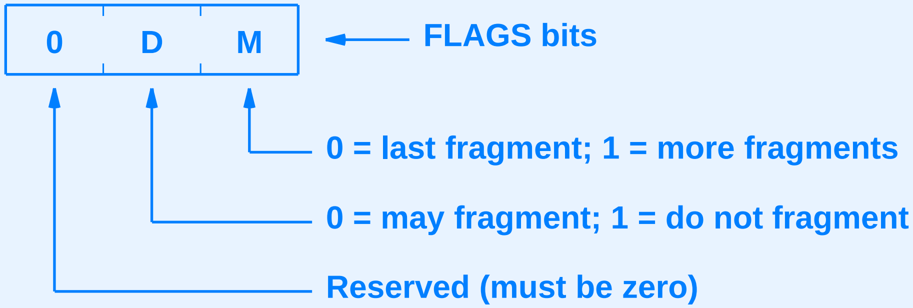
IP Forwarding Algorithm

```
Given: destination address A and routing table R.
Find: a next hop and interface used to route datagrams to A.
For each entry in table R  {
      Set MASK to the Address Mask in the entry;
      Set DEST to the Destination Address in the entry;
      If (A & MASK) == DEST  {
            Stop; use the next hop and interface in the entry;
      }
}
If this point is reached, declare error: no route exists;
```

- Note: algorithm assumes table is sorted in longest-prefix order

IP Fragmentation

- Needed when datagram larger than network MTU
- Divides IP datagram into *fragments*
- Uses FLAGS bits in datagram header


0
D
M
FLAGS bits
0 = last fragment; 1 = more fragments
0 = may fragment; 1 = do not fragment
Reserved (must be zero)

IP Fragmentation Algorithm (Part 1: Initialization)

```
Given: an IP datagram, D, and a network MTU.
Produce: a set of fragments for D.
If the DO NOT FRAGMENT bit is set  {
    Stop and report an error;
}
Compute the size of the datagram header, H;
Choose N to be the largest multiple of 8 such
    that  $H+N \le MTU$;
Initialize an offset counter, O, to zero;
```

IP Fragmentation Algorithm (Part 2: Processing)

```
Repeat until datagram empty {
    Create a new fragment that has a copy of D's header;
    Extract up to the next N octets of data from D and place the data in the fragment;
    Set the MORE FRAGMENTS bit in fragment header;
    Set TOTAL LENGTH field in fragment header to be $H+N$;
    Set FRAGMENT OFFSET field in fragment header to O;
    Compute and set the CHECKSUM field in fragment header;
    Increment O by $N/8$;
}
```

Reassembly

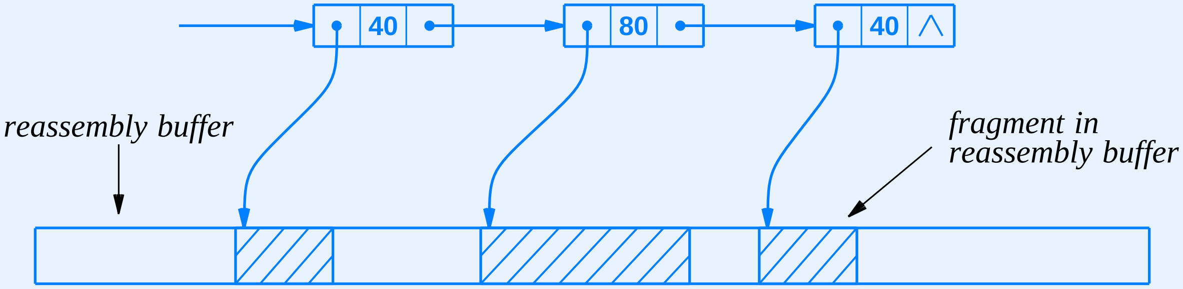
- Complement of fragmentation
- Uses IP SOURCE ADDRESS and IDENTIFICATION fields in datagram header to group related fragments
- Joins fragments to form original datagram

Reassembly Algorithm

```
Given: a fragment, F, add to a partial reassembly.
Method: maintain a set of fragments for each datagram.
Extract the IP source address, S, and ID fields from F;
Combine S and ID to produce a lookup key, K;
Find the fragment set with key K or create a new set;
Insert F into the set;
If the set contains all the data for the datagram  {
        Form a completely reassembled datagram and process it;
}
```

Data Structure For Reassembly

- Two parts
  - Buffer large enough to hold original datagram
  - Linked list of pieces that have arrived



TCP Connection

- Involves a pair of endpoints
- Started with SYN segment
- Terminated with FIN or RESET segment
- Identified by 4-tuple

(src addr, dest addr, src port, dest port)

TCP Connection Recognition Algorithm (Part 1)

```
Given: a copy of traffic passing across a network.
Produce: a record of TCP connections present in the traffic.
Initialize a connection table, C, to empty;
For each IP datagram that carries a TCP segment {
    Extract the IP source, S, and destination, D, addresses;
    Extract the source, $P_1$, and destination, $P_2$, port numbers;
    Use $(S,D,P_1,P_2)$ as a lookup key for table C and
        create a new entry, if needed;
```

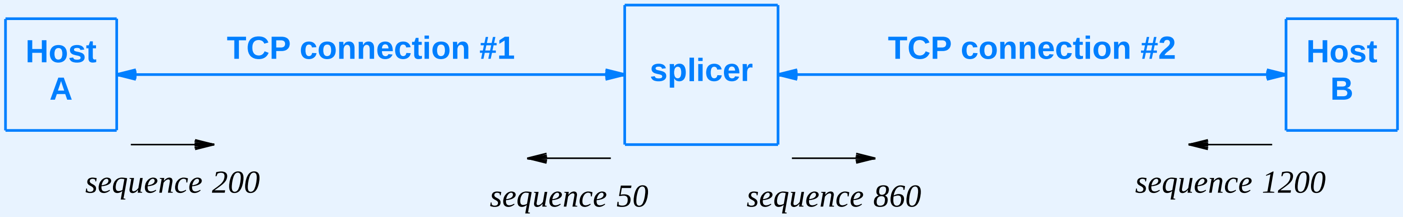
TCP Connection Recognition Algorithm (Part 2)

```
    If the segment has the RESET bit set, delete the entry;
    Else if the segment has the FIN bit set, mark the
connection
        closed in one direction, removing the entry from C if
        the connection was previously closed in the other;
    Else if the segment has the SYN bit set, mark the
connection as
        being established in one direction, making it completely
        established if it was previously marked as being
        established in the other;
}
```

TCP Splicing

- Join two TCP connections
- Allow data to pass between them
- To avoid termination overhead translate segment header fields
  - Acknowledgement number
  - Sequence number

Illustration Of TCP Splicing



Connection & Direction	Sequence Number	Connection & Direction	Sequence Number
Incoming #1	200	Incoming #2	1200
Outgoing #2	860	Outgoing #1	50
Change	660	Change	-1150

TCP Splicing Algorithm (Part 1)

```
Given: two TCP connections.
Produce: sequence translations for splicing the connection.
Compute D1, the difference between the starting sequences
    on incoming connection 1 and outgoing connection 2;
Compute D2, the difference between the starting sequences
    on incoming connection 2 and outgoing connection 1;
```

TCP Splicing Algorithm
(Part 2)

```
For each segment  {
    If segment arrived on connection 1  {
        Add D1 to sequence number;
        Subtract D2 from acknowledgement number;
    } else if segment arrived on connection 2  {
        Add D2 to sequence number;
        Subtract D1 from acknowledgement number;
    }
}
```

Summary

- Packet processing algorithms include
  - Ethernet bridging
  - IP fragmentation and reassembly
  - IP forwarding
  - TCP splicing
- Table lookup important
  - Full match for layer 2
  - Longest prefix match for layer 3

Questions?

For Hands-On Experience With

A Software-Based System:

Enroll in CS 636 !

VI

Packet Processing Functions

Goal

- Identify functions that occur in packet processing
- Devise set of operations sufficient for all packet processing
- Find an efficient implementation for the operations

Packet Processing Functions We Will Consider

- Address lookup and packet forwarding
- Error detection and correction
- Fragmentation, segmentation, and reassembly
- Frame and protocol demultiplexing
- Packet classification
- Queueing and packet discard
- Scheduling and timing
- Security: authentication and privacy
- Traffic measurement, policing, and shaping

Address Lookup And Packet Forwarding

- Forwarding requires address lookup
- Lookup is table driven
- Two types
  - Exact match (typically layer 2)
  - Longest-prefix match (typically layer 3)
- Cost depends on size of table and type of lookup

Error Detection And Correction

- Data sent with packet used as verification
  - Checksum
  - CRC
- Cost proportional to size of packet
- Often implemented with special-purpose hardware

An Important Note About Cost

The cost of an operation is proportional to the amount of data processed. An operation such as checksum computation that requires examination of all the data in a packet is among the most expensive.

Fragmentation, Segmentation, And Reassembly

- IP fragments and reassembles datagrams
- ATM segments and reassembles AAL5 packets
- Same idea; details differ
- Cost is high because
  - State must be kept and managed
  - Unreassembled fragments occupy memory

Frame And Protocol Demultiplexing

- Traditional technique used in layered protocols
- Type appears in each header
  - Assigned on output
  - Used on input to select ‘‘next’’ protocol
- Cost of demultiplexing proportional to number of layers

Packet Classification

- Alternative to demultiplexing
- Crosses multiple layers
- Achieves lower cost
- More on classification later in the course

Queueing And Packet Discard

- General paradigm is *store-and-forward*
  - Incoming packet placed in queue
  - Outgoing packet placed in queue
- When queue is full, choose packet to discard
- Affects throughput of higher-layer protocols

Queueing Priorities

- Multiple queues used to enforce priority among packets
- Incoming packet
  - Assigned priority as function of contents
  - Placed in appropriate priority queue
- *Queueing discipline*
  - Examines priority queues
  - Chooses which packet to send

Examples Of Queueing Disciplines

- Priority Queueing
  - Assign unique priority number to each queue
  - Choose packet from highest priority queue that is nonempty
  - Known as *strict priority* queueing
  - Can lead to starvation

Examples Of Queueing Disciplines
(continued)

- Weighted Round Robin (WRR)
  - Assign unique priority number to each queue
  - Process all queues round-robin
  - Compute N, max number of packets to select from a queue proportional to priority
  - Take up to N packets before moving to next queue
  - Works well if all packets equal size

Examples Of Queueing Disciplines
(continued)

- Weighted Fair Queueing (WFQ)
  - Make selection from queue proportional to priority
  - Use packet size rather than number of packets
  - Allocates priority to amount of data from a queue rather than number of packets

Scheduling And Timing

- Important mechanisms
- Used to coordinate parallel and concurrent tasks
  - Processing on multiple packets
  - Processing on multiple protocols
  - Multiple processors
- Scheduler attempts to achieve fairness

Security: Authentication And Privacy

- Authentication mechanisms
  - Ensure sender’s identity
- Confidentiality mechanisms
  - Ensure that intermediaries cannot interpret packet contents
- Note: in common networking terminology, *privacy* refers to confidentiality
  - Example: Virtual Private Networks

Traffic Measurement And Policing

- Used by network managers
- Can measure aggregate traffic or per-flow traffic
- Often related to Service Level Agreement (SLA)
- Cost is high if performed in real-time

Traffic Shaping

- Make traffic conform to statistical bounds
- Typical use
  - Smooth bursts
  - Avoid packet trains
- Only possibilities
  - Discard packets (seldom used)
  - Delay packets

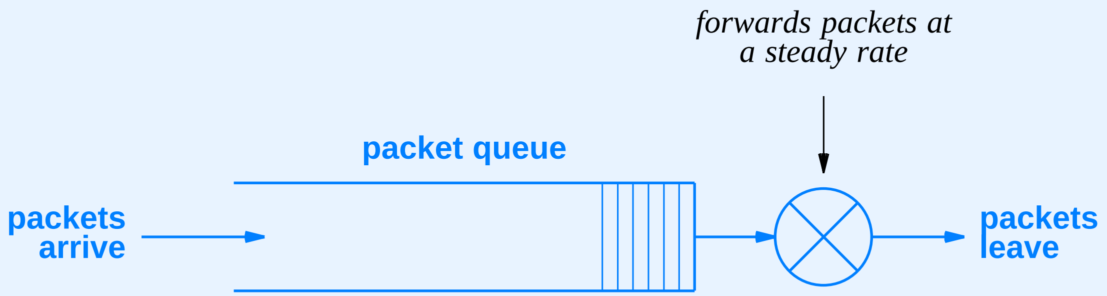
Example Traffic Shaping Mechanisms

- Leaky bucket
  - Easy to implement
  - Popular
  - Sends steady number of packets per second
  - Rate depends on number of packets waiting
  - Does not guarantee steady data rate

Example Traffic Shaping Mechanisms (continued)

- Token bucket
  - Sends steady number of bits per second
  - Rate depends on number of bits waiting
  - Achieves steady data rate
  - More difficult to implement

Illustration Of Traffic Shaper



- Packets
  - Arrive in bursts
  - Leave at steady rate

Timer Management

- Fundamental piece of network system
- Needed for
  - Scheduling
  - Traffic shaping
  - Other protocol processing (e.g., retransmission)
- Cost
  - Depends on number of timer operations (e.g., set, cancel)
  - Can be high

Summary

- Primary packet processing functions are
  - Address lookup and forwarding
  - Error detection and correction
  - Fragmentation and reassembly
  - Demultiplexing and classification
  - Queueing and discard
  - Scheduling and timing
  - Security functions
  - Traffic measurement, policing, and shaping

Questions?

VII

Protocol Software On A Conventional Processor

Possible Implementations Of Protocol Software

- In an application program
  - Easy to program
  - Runs as user-level process
  - No direct access to network devices
  - High cost to copy data from kernel address space
  - Cannot run at *wire speed*

Possible Implementations Of Protocol Software
(continued)

- In an embedded system
  - Special-purpose hardware device
  - Dedicated to specific task
  - Ideal for stand-alone system
  - Software has full control

Possible Implementations Of Protocol Software
(continued)

- In an embedded system
  - Special-purpose hardware device
  - Dedicated to specific task
  - Ideal for stand-alone system
  - Software has full control
  - You will experience this in lab!

Possible Implementations Of Protocol Software (continued)

- In an operating system kernel
  - More difficult to program than application
  - Runs with kernel privilege
  - Direct access to network devices

Interface To The Network

- Known as *Application Program Interface (API)*
- Can be
  - *Asynchronous*
  - *Synchronous*
- Synchronous interface can use
  - *Blocking*
  - *Polling*

Asynchronous API

- Also known as *event-driven*
- Programmer
  - Writes set of functions
  - Specifies which function to invoke for each event type
- Programmer has no control over function invocation
- Functions keep state in shared memory
- Difficult to program
- Example: function *f()* called when packet arrives

Synchronous API Using Blocking

- Programmer
  - Writes main flow-of-control
  - Explicitly invokes functions as needed
  - Built-in functions block until request satisfied
- Example: function *wait_for_packet()* blocks until packet arrives
- Easier to program

Synchronous API Using Polling

- Nonblocking form of synchronous API
- Each function call returns immediately
  - Performs operation if available
  - Returns error code otherwise
- Example: function *try_for_packet()* either returns next packet or error code if no packet has arrived
- Closer to underlying hardware

Typical Implementations And APIs

- Application program
  - Synchronous API using blocking (e.g., socket API)
  - Another application thread runs while an application blocks
- Embedded systems
  - Synchronous API using polling
  - CPU dedicated to one task
- Operating systems
  - Asynchronous API
  - Built on interrupt mechanism

Example Asynchronous API

- Design goals
  - For use with network processor
  - Simplest possible interface
  - Sufficient for basic packet processing tasks
- Includes
  - I/O functions
  - Timer manipulation functions

Example Asynchronous API (continued)

- Initialization and termination functions
  - on_startup()
  - on_shutdown()
- Input function (called asynchronously)
  - recv_frame()
- Output functions
  - new_fbuf()
  - send_frame()

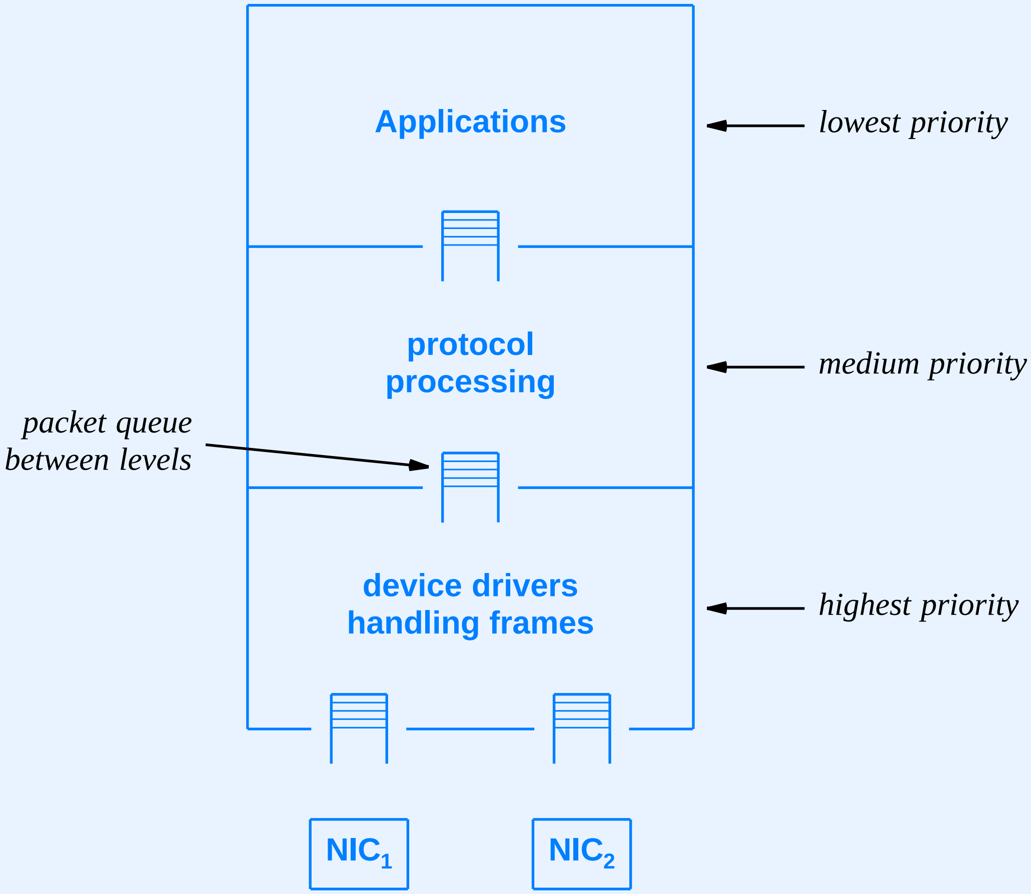
Example Asynchronous API (continued)

- Timer functions (called asynchronously)
  - delayed_call()
  - periodic_call()
  - cancel_call()
- Invoked by outside application
  - console_command()

Processing Priorities

- Determine which code CPU runs at any time
- General idea
  - Hardware devices need highest priority
  - Protocol software has medium priority
  - Application programs have lowest priority
- Queues provide buffering across priorities

Illustration Of Priorities



Implementation Of Priorities In An Operating System

- Two possible approaches
  - Interrupt mechanism
  - Kernel threads

Interrupt Mechanism

- Built into hardware
- Operates asynchronously
- Saves current processing state
- Changes processor status
- Branches to specified location

Two Types Of Interrupts

- *Hardware interrupt*
  - Caused by device (bus)
  - Must be serviced quickly
- *Software interrupt*
  - Caused by executing program
  - Lower priority than hardware interrupt
  - Higher priority than other OS code

Software Interrupts And Protocol Code

- Protocol stack operates as software interrupt
- When packet arrives
  - Hardware interrupts
  - Device driver raises software interrupt
- When device driver finishes
  - Hardware interrupt clears
  - Protocol code is invoked

Kernel Threads

- Alternative to interrupts
- Familiar to programmer
- Finer-grain control than software interrupts
- Can be assigned arbitrary range of priorities

Conceptual Organization

- Packet passes among multiple threads of control
- Queue of packets between each pair of threads
- Threads synchronize to access queues

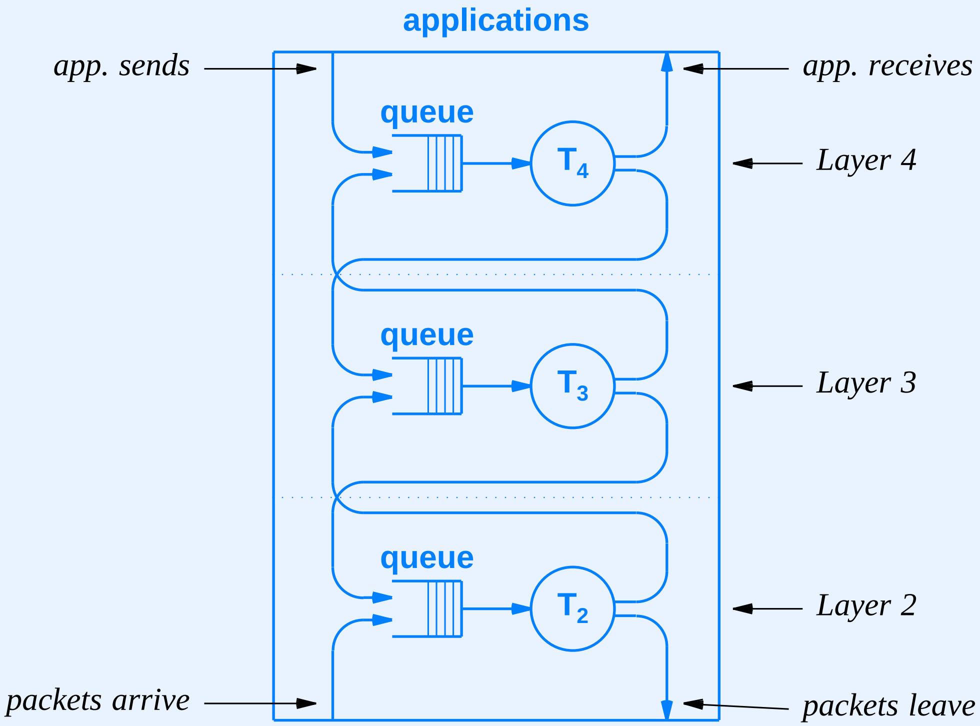
Possible Organization Of Kernel Threads For Layered Protocols

- One thread per layer
- One thread per protocol
- Multiple threads per protocol
- Multiple threads per protocol plus timer management thread(s)
- One thread per packet

One Thread Per Layer

- Easy for programmer to understand
- Implementation matches concept
- Allows priority to be assigned to each layer
- Means packet is enqueued once per layer

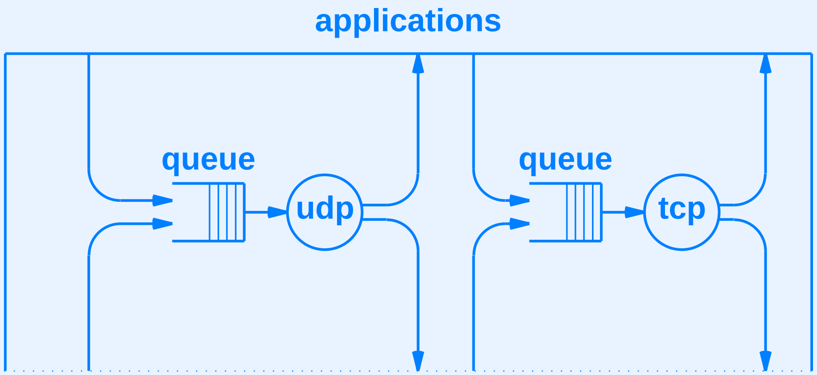
Illustration Of One Thread Per Layer



One Thread Per Protocol

- Like one thread per layer
  - Implementation matches concept
  - Means packet is enqueued once per layer
- Advantages over one thread per layer
  - Easier for programmer to understand
  - Finer-grain control
  - Allows priority to be assigned to each protocol

Illustration Of One Thread Per Protocol

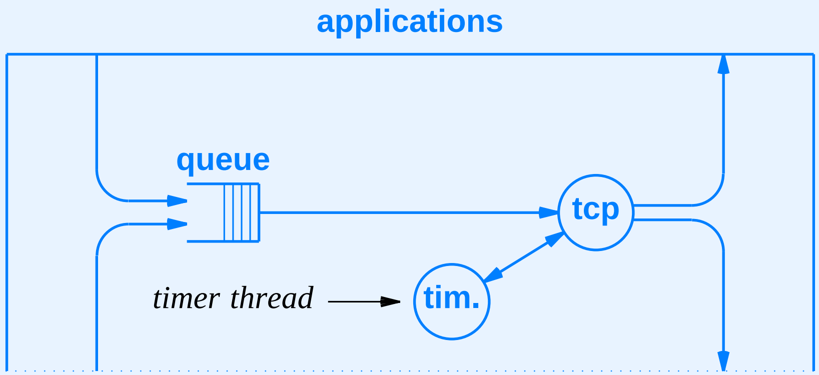


- TCP and UDP reside at same layer
- Separation allows priority

Multiple Threads Per Protocol

- Further division of duties
- Simplifies programming
- More control than single thread
- Typical division
  - Thread for incoming packets
  - Thread for outgoing packets
  - Thread for management/timing

Illustration Of Multiple Threads Used With TCP



- Separate timer makes programming easier

Timers And Protocols

- Many protocols implement timeouts
  - TCP
    - Retransmission timeout
    - 2MSL timeout
  - ARP
    - Cache entry timeout
  - IP
    - Reassembly timeout

Multiple Threads Per Protocol Plus Timer Management Thread(s)

- Observations
  - Many protocols each need timer functionality
  - Each timer thread incurs overhead
- Solution: consolidate timers for multiple protocols

Is One Timer Thread Sufficient?

- In theory
  - Yes
- In practice
  - Large range of timeouts (microseconds to tens of seconds)
  - May want to give priority to some timeouts
- Solution: two or more timer threads

Multiple Timer Threads

- Two threads usually suffice
- Large-granularity timer
  - Values specified in seconds
  - Operates at lower priority
- Small-granularity timer
  - Values specified in microseconds
  - Operates at higher priority

Thread Synchronization

- Thread for layer *i*
  - Needs to pass a packet to layer *i + 1*
  - Enqueues the packet
- Thread for layer *i + 1*
  - Retrieves packet from the queue

Thread Synchronization

- Thread for layer *i*
  - Needs to pass a packet to layer *i* + *1*
  - Enqueues the packet
- Thread for layer *i* + *1*
  - Retrieves packet from the queue
- Context switch required!

Context Switch

- OS function
- CPU passes from current thread to a waiting thread
- High cost
- Must be minimized

One Thread Per Packet

- Preallocate set of threads
- Thread operation
  - Waits for packet to arrive
  - Moves through protocol stack
  - Returns to wait for next packet
- Minimizes context switches

Summary

- Packet processing software usually runs in OS
- API can be synchronous or asynchronous
- Priorities achieved with
  - Software interrupts
  - Threads
- Variety of thread architectures possible

Questions?

VIII

Hardware Architectures For Protocol Processing And Aggregate Rates

A Brief History Of
Computer Hardware

- 1940s
  - Beginnings
- 1950s
  - Consolidation of von Neumann architecture
  - I/O controlled by CPU
- 1960s
  - I/O becomes important
  - Evolution of third generation architecture with interrupts

I/O Processing

- Evolved from after-thought to central influence
- Low-end systems (e.g., microcontrollers)
  - Dumb I/O interfaces
  - CPU does all the work (polls devices)
  - Single, shared memory
  - Low cost, but low speed

I/O Processing
(continued)

- Mid-range systems (e.g., minicomputers)
  - Single, shared memory
  - I/O interfaces contain logic for transfer and status operations
  - CPU
    - Starts device then resumes processing
  - Device
    - Transfers data to / from memory
    - Interrupts when operation complete

I/O Processing
(continued)

- High-end systems (e.g., mainframes)
  - Separate, programmable I/O processor
  - OS downloads code to be run
  - Device has private on-board buffer memory
  - Examples: IBM channel, CDC peripheral processor

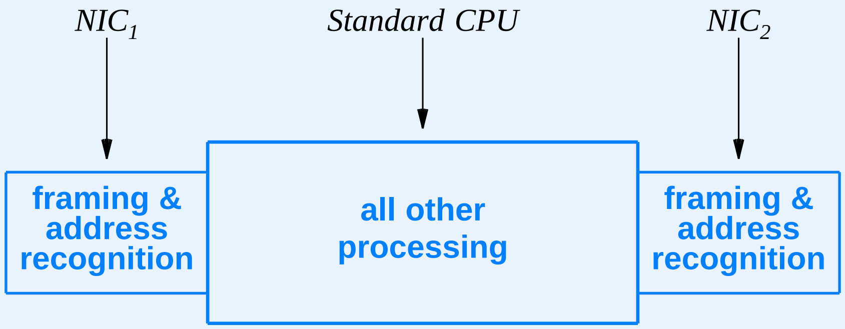
Networking Systems Evolution

- Twenty year history
- Same trend as computer architecture
  - Began with central CPU
  - Shift to emphasis on I/O
- Three main generations

First Generation Network Systems

- Traditional software-based router
- Used conventional (minicomputer) hardware
  - Single general-purpose processor
  - Single shared memory
  - I/O over a bus
  - Network interface cards use same design as other I/O devices

Protocol Processing In First Generation Network Systems



- General-purpose processor handles most tasks
- Sufficient for low-speed systems
- Note: we will examine other generations later in the course

How Fast Does A CPU Need To Be?

- Depends on
  - Rate at which data arrives
  - Amount of processing to be performed

Two Measures Of Speed

- Data rate (bits per second)
  - Per interface rate
  - Aggregate rate
- Packet rate (packets per second)
  - Per interface rate
  - Aggregate rate

How Fast Is A Fast Connection?

- Definition of fast data rate keeps changing
  - 1960: 10 Kbps
  - 1970: 1 Mbps
  - 1980: 10 Mbps
  - 1990: 100 Mbps
  - 2000: 1000 Mbps (1 Gbps)
  - 2004: 2400 Mbps

How Fast Is A Fast Connection?

- Definition of fast data rate keeps changing
  - 1960: 10 Kbps
  - 1970: 1 Mbps
  - 1980: 10 Mbps
  - 1990: 100 Mbps
  - 2000: 1000 Mbps (1 Gbps)
  - 2004: 2400 Mbps
  - Soon: 10 Gbps???

Aggregate Rate Vs.
Per-Interface Rate

- Interface rate
  - Rate at which data enters / leaves
- Aggregate
  - Sum of interface rates
  - Measure of total data rate system can handle
- Note: aggregate rate crucial if CPU handles traffic from all interfaces

A Note About System Scale

The aggregate data rate is defined to be the sum of the rates at which traffic enters or leaves a system. The maximum aggregate data rate of a system is important because it limits the type and number of network connections the system can handle.

Packet Rate Vs. Data Rate

- Sources of CPU overhead
  - Per-bit processing
  - Per-packet processing
- Interface hardware handles much of per-bit processing

A Note About System Scale

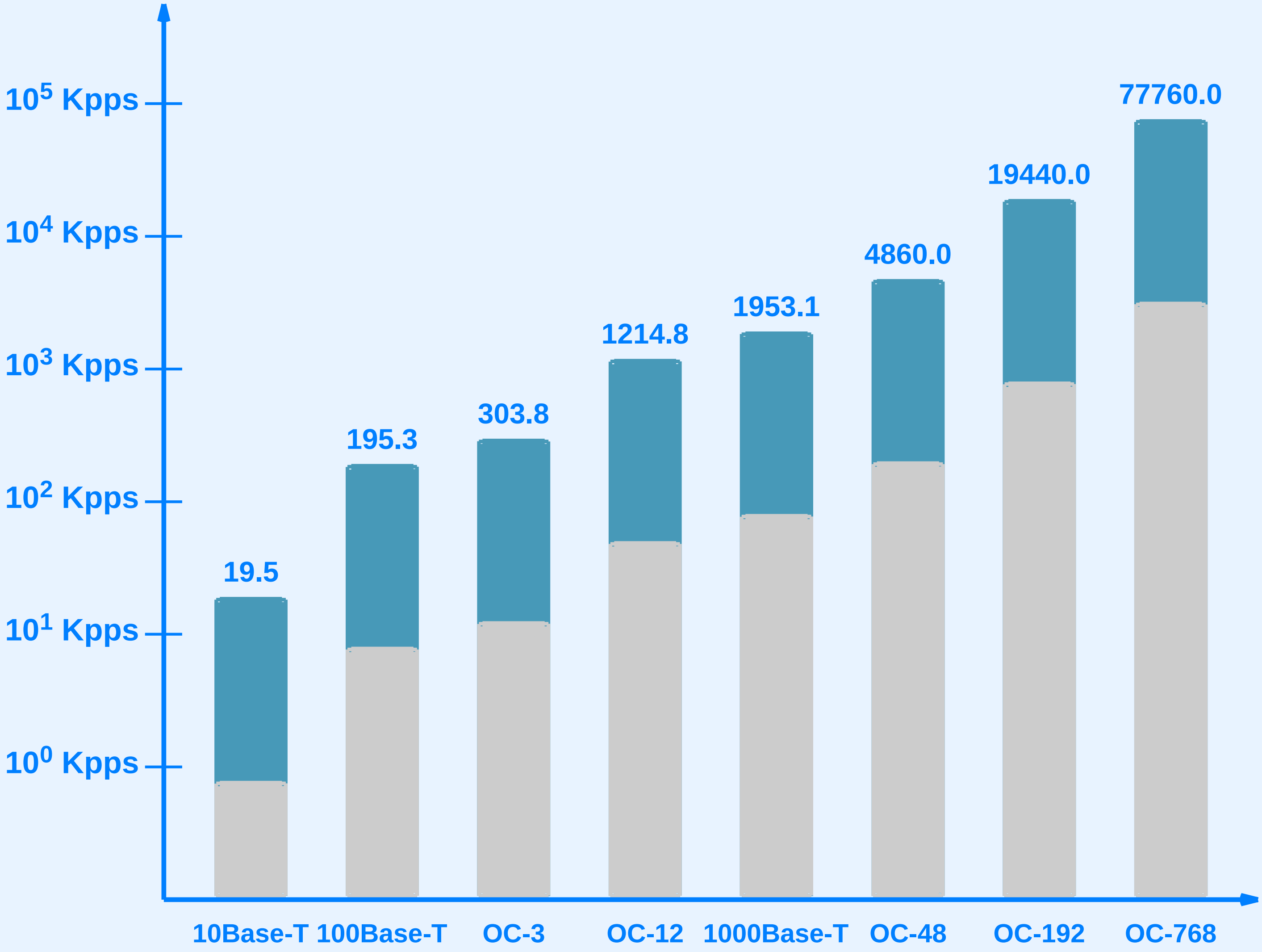
For protocol processing tasks that have a fixed cost per packet, the number of packets processed is more important than the aggregate data rate.

Example Packet Rates

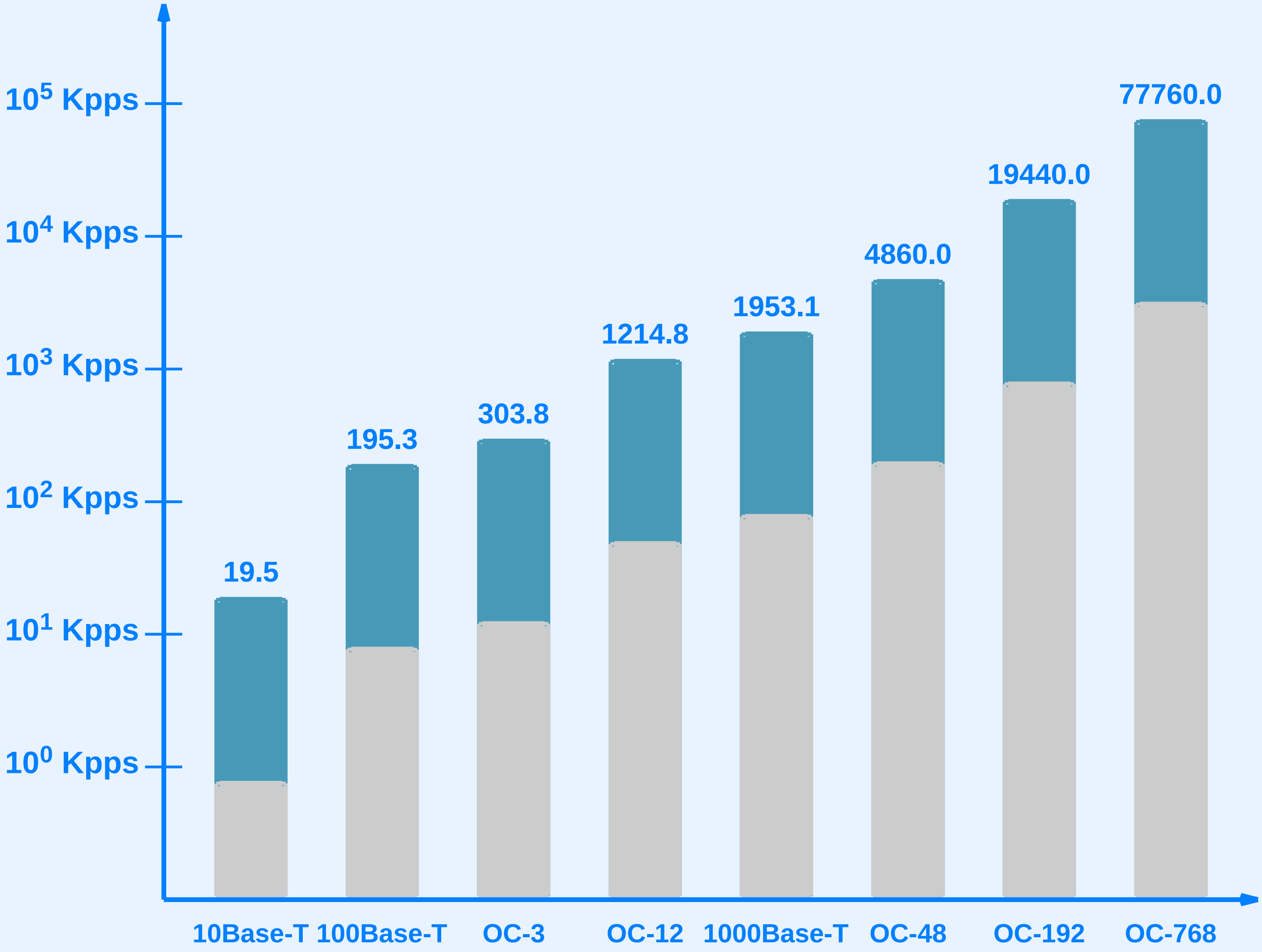
Technology	Network Data Rate In Gbps	Packet Rate For Small Packets In Kpps	Packet Rate For Large Packets In Kpps
10Base-T	0.010	19.5	0.8
100Base-T	0.100	195.3	8.2
OC-3	0.156	303.8	12.8
OC-12	0.622	1,214.8	51.2
1000Base-T	1.000	1,953.1	82.3
OC-48	2.488	4,860.0	204.9
OC-192	9.953	19,440.0	819.6
OC-768	39.813	77,760.0	3,278.4

- Key concept: maximum packet rate occurs with minimum-size packets

Bar Chart Of Example Packet Rates



Bar Chart Of Example Packet Rates



- Gray areas show rates for large packets

Time Per Packet

Technology	Time per packet for small packets (in µs)	Time per packet for large packets (in µs)
10Base-T	51.20	1,214.40
100Base-T	5.12	121.44
OC-3	3.29	78.09
OC-12	0.82	19.52
1000Base-T	0.51	12.14
OC-48	0.21	4.88
OC-192	0.05	1.22
OC-768	0.01	0.31

- Note: these numbers are for a single connection!

Conclusion

Software running on a general-purpose processor is an insufficient architecture to handle high-speed networks because the aggregate packet rate exceeds the capabilities of a CPU.

Possible Ways To Solve The CPU Bottleneck

- Fine-grain parallelism
- Symmetric coarse-grain parallelism
- Asymmetric coarse-grain parallelism
- Special-purpose coprocessors
- NICs with onboard processing
- Smart NICs with onboard stacks
- Cell switching
- Data pipelines

Fine-Grain Parallelism

- Multiple processors
- Instruction-level parallelism
- Example:
  - Parallel checksum: add values of eight consecutive memory locations at the same time
- Assessment: insignificant advantages for packet processing

Symmetric Coarse-Grain Parallelism

- Symmetric multiprocessor hardware
  - Multiple, identical processors
- Typical design: each CPU operates on one packet
- Requires coordination
- Assessment: coordination and data access means *N* processors cannot handle *N* times more packets than one processor

Asymmetric Coarse-Grain Parallelism

- Multiple processors
- Each processor
  - Optimized for specific task
  - Includes generic instructions for control
- Assessment
  - Same problems of coordination and data access as symmetric case
  - Designer must choose how many copies of each processor type

Special-Purpose Coprocessors

- Special-purpose hardware
- Added to conventional processor to speed computation
- Invoked like software subroutine
- Typical implementation: ASIC chip
- Choose operations that yield greatest improvement in speed

General Principle

To optimize computation, move operations that account for the most CPU time from software into hardware.

General Principle

To optimize computation, move operations that account for the most CPU time from software into hardware.

- Idea known as *Amdahl’s law* (performance improvement from faster hardware technology is limited to the fraction of time the faster technology can be used)

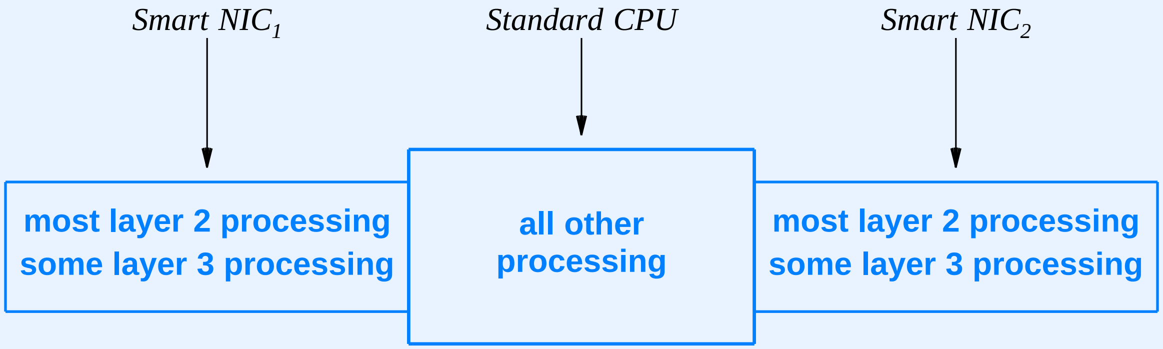
NICs And Onboard Processing

- Basic optimizations
  - Onboard address recognition and filtering
  - Onboard buffering
  - DMA
  - Buffer and operation chaining
- Further optimization possible

Smart NICs With Onboard Stacks

- Add hardware to NIC
  - Off-the-shelf chips for layer 2
  - ASICs for layer 3
- Allows each NIC to operate independently
  - Effectively a multiprocessor
  - Total processing power increased dramatically

Illustration Of Smart NICs With Onboard Processing



- NIC handles layers 2 and 3
- CPU only handles exceptions

Cell Switching

- Alternative to new hardware
- Changes
  - Basic paradigm
  - All details (e.g., protocols)
- Connection-oriented

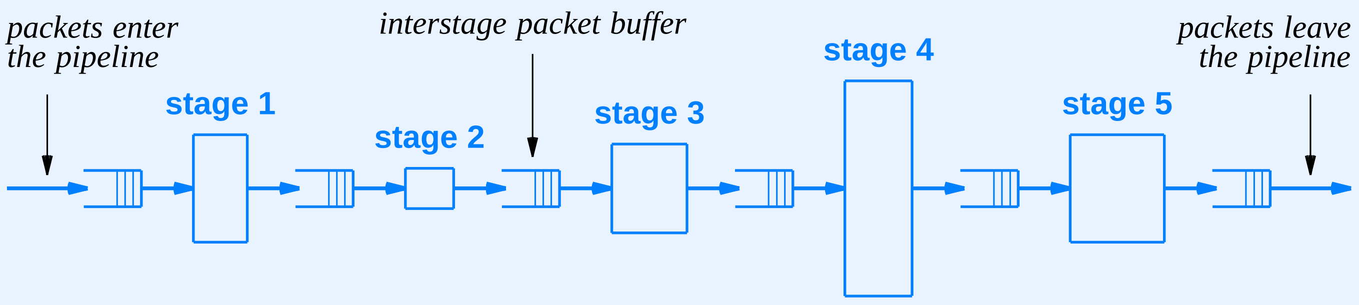
Cell Switching Details

- Fixed-size packets
  - Allows fixed-size buffers
  - Guaranteed time to transmit/receive
- Relative (connection-oriented) addressing
  - Smaller address size
  - Label on packet changes at each switch
  - Requires connection setup
- Example: ATM

Data Pipeline

- Move each packet through series of processors
- Each processor handles some tasks
- Assessment
  - Well-suited to many protocol processing tasks
  - Individual processor can be fast

Illustration Of Data Pipeline



- Pipeline can contain heterogeneous processors
- Packets pass through each stage

Summary

- Packet rate can be more important than data rate
- Highest packet rate achieved with smallest packets
- Rates measured per interface or aggregate
- Special hardware needed for highest-speed network systems
  - Smart NIC can include part of protocol stack
  - Parallel and pipelined hardware also possible

Questions?

IX

Classification And Forwarding

Recall

- Packet demultiplexing
  - Used with layered protocols
  - Packet proceeds through one layer at a time
  - On input, software in each layer chooses module at next higher layer
  - On output, type field in each header specifies encapsulation

The Disadvantage Of Demultiplexing

Although it provides freedom to define and use arbitrary protocols without introducing transmission overhead, demultiplexing is inefficient because it imposes sequential processing among layers.

Packet Classification

- Alternative to demultiplexing
- Designed for higher speed
- Considers all layers at the same time
- Linear in number of fields
- Two possible implementations
  - Software
  - Hardware

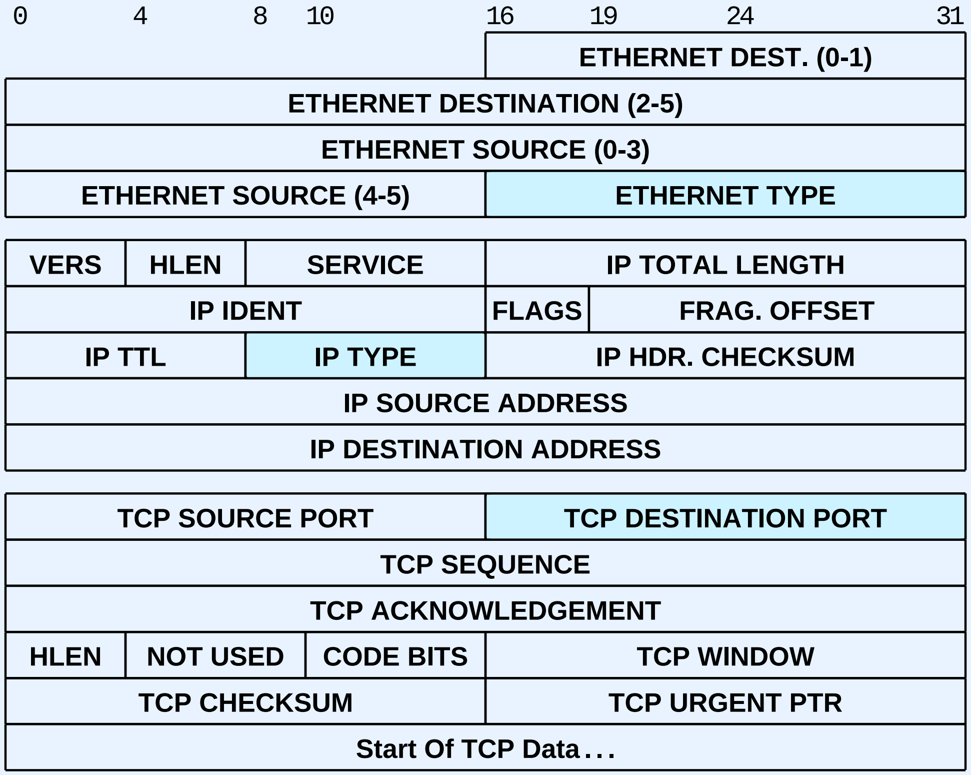
Example Classification

- Classify Ethernet frames carrying traffic to Web server
- Specify exact header contents in *rule set*
- Example
  - Ethernet type field specifies IP
  - IP type field specifies TCP
  - TCP destination port specifies Web server

Example Classification
(continued)

- Field sizes and values
  - 2-octet Ethernet type is 0800_{16}
  - 1-octet IP type is 6
  - 2-octet TCP destination port is 80

Illustration Of Encapsulated Headers



0 4 8 10 16 19 24 31

ETHERNET DEST. (0-1)

ETHERNET DESTINATION (2-5)

ETHERNET SOURCE (0-3)

ETHERNET SOURCE (4-5) | ETHERNET TYPE

VERS | HLEN | SERVICE | IP TOTAL LENGTH

IP IDENT | FLAGS | FRAG. OFFSET

IP TTL | IP TYPE | IP HDR. CHECKSUM

IP SOURCE ADDRESS

IP DESTINATION ADDRESS

TCP SOURCE PORT | TCP DESTINATION PORT

TCP SEQUENCE

TCP ACKNOWLEDGEMENT

HLEN | NOT USED | CODE BITS | TCP WINDOW

TCP CHECKSUM | TCP URGENT PTR

Start Of TCP Data . . .

- Highlighted fields are used for classification of Web server traffic

Software Implementation Of Classification

- Compare values in header fields
- Conceptually a *logical and* of all field comparisons
- Example

```
if ((frame type == 0x0800) && (IP type == 6) && (TCP port == 80))
        declare the packet matches the classification;
else
        declare the packet does not match the classification;
```

Optimizing Software Classification

- Comparisons performed sequentially
- Can reorder comparisons to minimize effort

Example Of Optimizing Software Classification

- Assume
  - 95.0% of all frames have frame type 0800_{16}
  - 87.4% of all frames have IP type 6
  - 74.3% of all frames have TCP port 80
- Also assume values 6 and 80 do not occur in corresponding positions in non-IP packet headers
- Reordering tests can optimize processing time

Example Of Optimizing Software Classification (continued)

```
if ((TCP port == 80) && (IP type == 6) && (frame type == 0x0800))
        declare the packet matches the classification;
else
        declare the packet does not match the classification;
```

- At each step, test the field that will eliminate the most packets

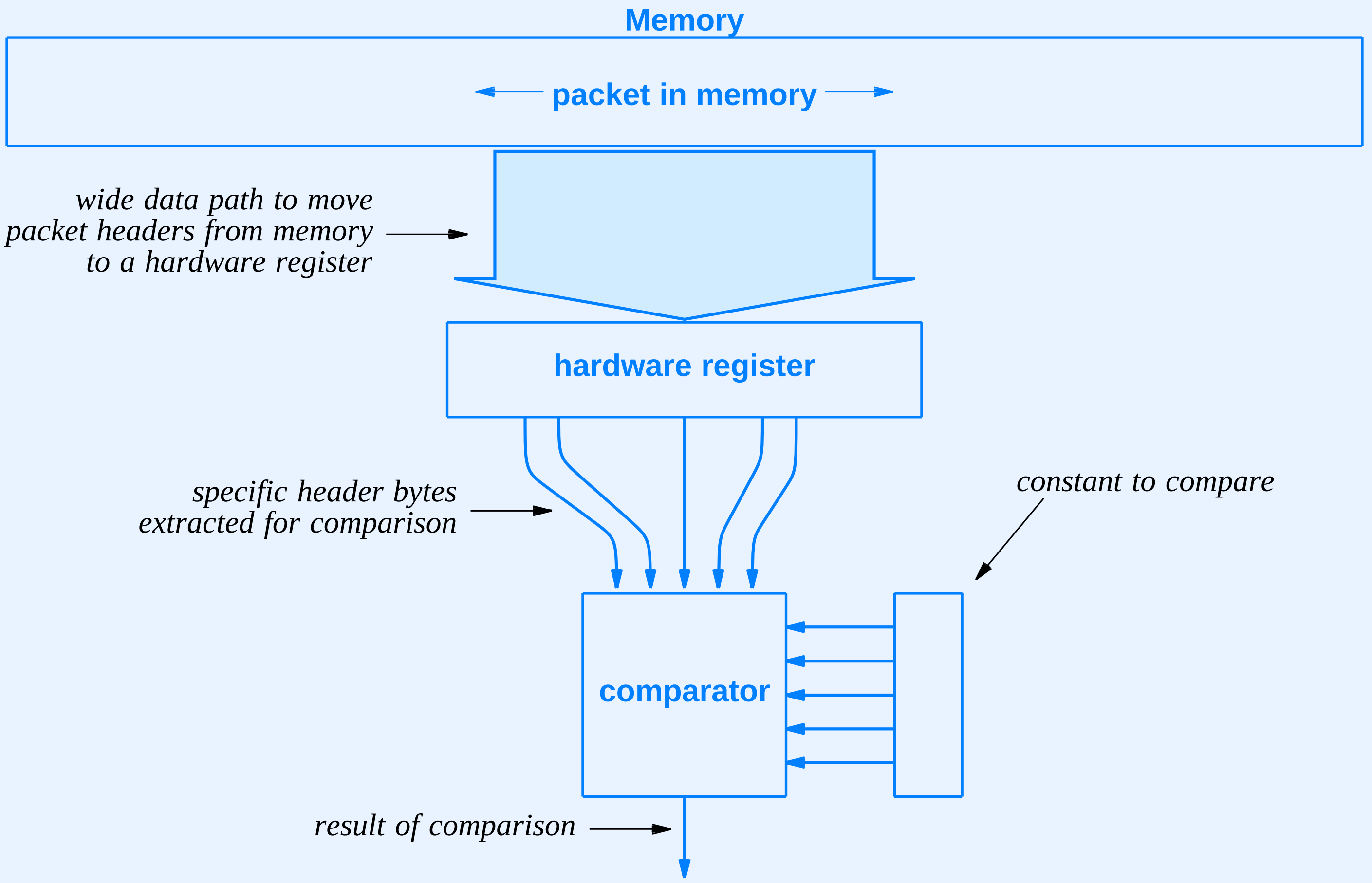
Note About Optimization

Although the maximum number of comparisons in a software classifier is fixed, the average number of comparisons is determined by the order of the tests; minimum comparisons result if, at each step, the classifier tests the field that eliminates the most packets.

Hardware Implementation Of Classification

- Can build special-purpose hardware
- Steps
  - Extract needed fields
  - Concatenate bits
  - Place result in register
  - Perform comparison
- Hardware can operate in parallel

Illustration Of Hardware Classifier


Memory
packet in memory
wide data path to move packet headers from memory to a hardware register
hardware register
specific header bytes extracted for comparison
constant to compare
comparator
result of comparison

- Constant for Web classifier is $08.00.06.00.50_{16}$

Special Cases Of Classification

- Multiple categories
- Variable-size headers
- Dynamic classification

In Practice

- Classification usually involves multiple categories
- Packets grouped together into *flows*
- May have a default category
- Each category specified with rule set

Example Multi-Category Classification

- Flow 1: traffic destined for Web server
- Flow 2: traffic consisting of ICMP echo request packets
- Flow 3: all other traffic (default)

Rule Sets

- Web server traffic
  - 2-octet Ethernet type is 0800_{16}
  - 1-octet IP type is 6
  - 2-octet TCP destination port is 80
- ICMP echo traffic
  - 2-octet Ethernet type is 0800_{16}
  - 1-octet IP type is 1
  - 1-octet ICMP type is 8

Software Implementation Of Multiple Rules

```
if  (frame type != 0x0800)  {
        send frame to flow 3;
}  else  if  (IP type == 6  &&  TCP destination port == 80)  {
        send packet to flow 1;
}  else  if  (IP type == 1  &&  ICMP type == 8)  {
        send packet to flow 2;
}  else  {
        send frame to flow 3;
}
```

- Further optimization possible

Variable-Size Packet Headers

- Fields not at fixed offsets
- Easily handled with software
- Finite cases can be specified in rules

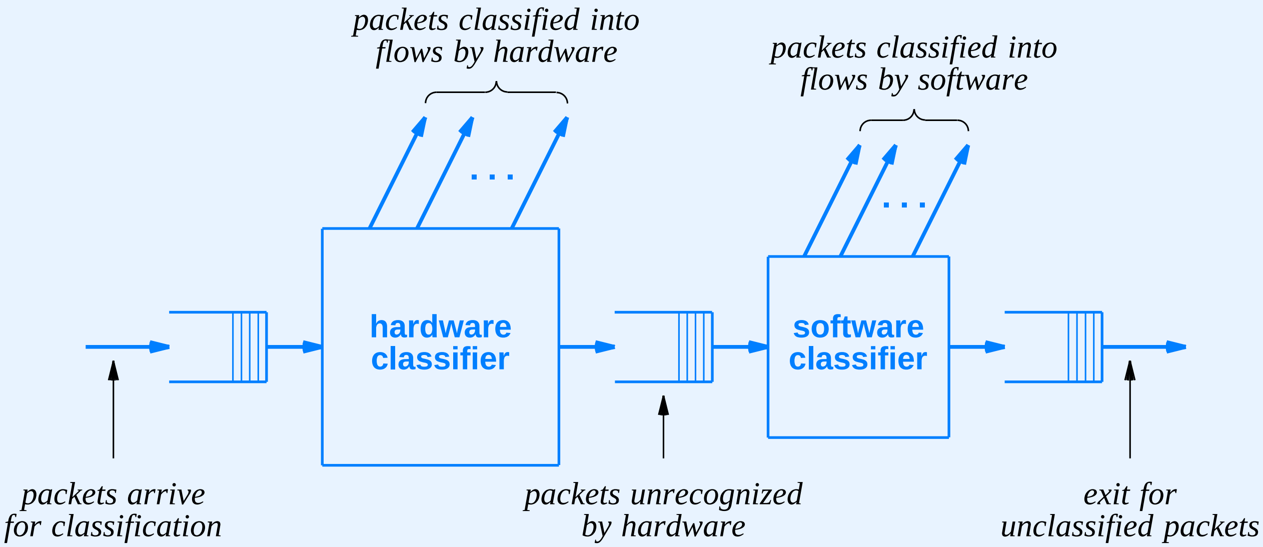
Example Variable-Size Header: IP Options

- Rule Set 1
  - 2-octet frame type field contains 0800_{16}
  - 1-octet field at the start of the datagram contains 45_{16}
  - 1-octet type field in the IP datagram contains 6
  - 2-octet field 22 octets from start of the datagram contains 80
- Rule Set 2
  - 2-octet frame type field contains 0800_{16}
  - 1-octet field at the start of the datagram contains 46_{16}
  - 1-octet type field in the IP datagram contains 6
  - 2-octet field 26 octets from the start of datagram contains 80

Effect Of Protocol Design On Classification

- Fixed headers fastest to classify
- Each variable-size header adds one computation step
- In worst case, classification no faster than demultiplexing
- Extreme example: IPv6

Hybrid Classification



- Combines hardware and software mechanisms
  - Hardware used for standard cases
  - Software used for exceptions
- Note: software classifier can operate at slower rate

Two Basic Types Of Classification

- Static
  - Flows specified in rule sets
  - Header fields and values known a priori
- Dynamic
  - Flows created by observing packet stream
  - Values taken from headers
  - Allows fine-grain flows
  - Requires state information

Example Static Classification

- Allocate one flow per service type
- One header field used to identify flow
  - IP TYPE OF SERVICE (TOS)
- Use DIFFSERV interpretation
- Note: Ethernet type field also checked

Example Dynamic Classification

- Allocate flow per TCP connection
- Header fields used to identify flow
  - IP source address
  - IP destination address
  - TCP source port number
  - TCP destination port number
- Note: Ethernet type and IP type fields also checked

Implementation Of Dynamic Classification

- Usually performed in software
- State kept in memory
- State information created/updated at wire speed

Two Conceptual Bindings

classification: packet → *flow*

forwarding: flow → *packet disposition*

- Classification binding is usually 1-to-1
- Forwarding binding can be 1-to-1 or many-to-1

Flow Identification

- Connection-oriented network
  - Per-flow SVC can be created on demand
  - Flow ID equals connection ID
- Connectionless network
  - Flow ID used internally
  - Each flow ID mapped to *(next hop, interface)*

Relationship Of Classification And Forwarding In A Connection-Oriented Network

In a connection-oriented network, flow identifiers assigned by classification can be chosen to match connection identifiers used by the underlying network. Doing so makes forwarding more efficient by eliminating one binding.

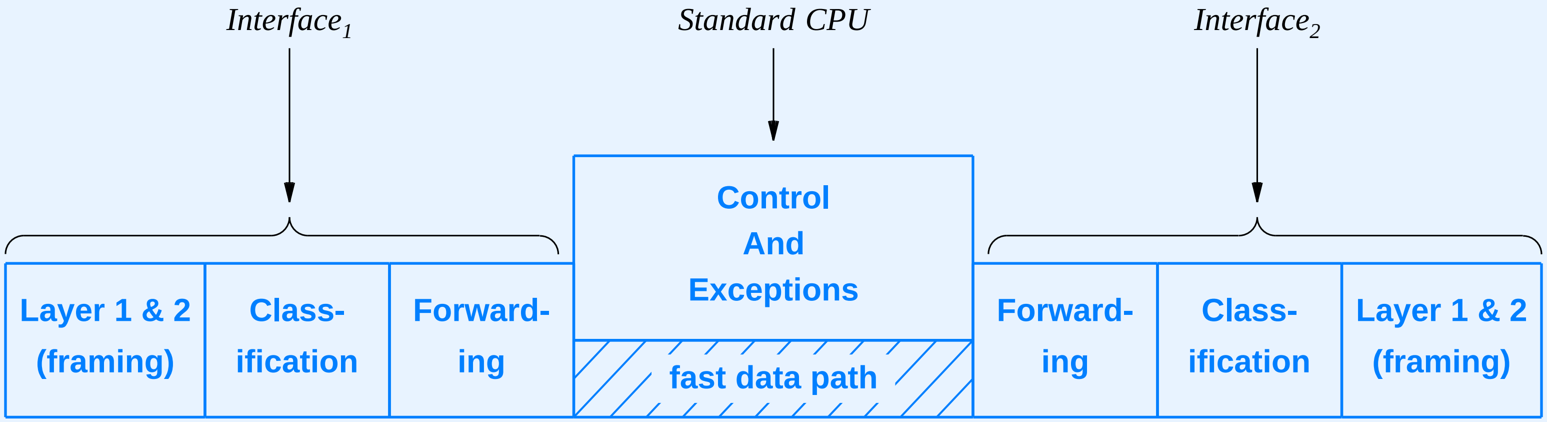
Forwarding In A Connectionless Network

- Route for flow determined when flow created
- Indexing used in place of route lookup
- Flow identifier corresponds to index of entry in forwarding cache
- Forwarding cache must be changed when route changes

Second Generation Network Systems

- Designed for greater scale
- Use classification instead of demultiplexing
- Decentralized architecture
  - Additional computational power on each NIC
  - NIC implements classification and forwarding
- High-speed internal interconnection mechanism
  - Interconnects NICs
  - Provides *fast data path*

Illustration Of Second Generation Network Systems Architecture



Classification And Forwarding Chips

- Sold by vendors
- Implement hardware classification and forwarding
- Typical configuration: rule sets given in ROM

Summary

- Classification faster than demultiplexing
- Can be implemented in hardware or software
- Dynamic classification
  - Uses packet contents to assign flows
  - Requires state information

Questions?

XI

Network Processors: Motivation And Purpose

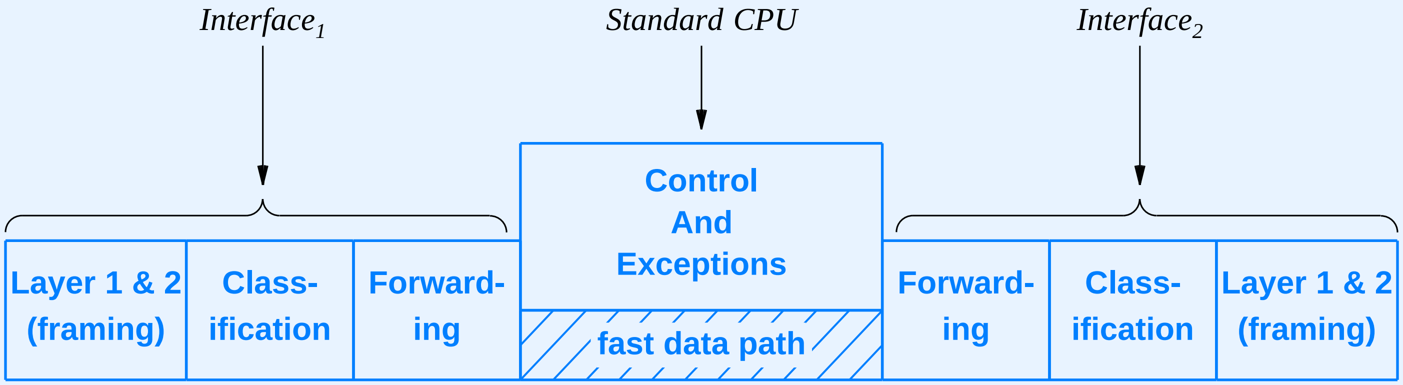
Second Generation Network Systems

- Concurrent with ATM development (early 1990s)
- Purpose: scale to speeds faster than single CPU capacity
- Features
  - Use classification instead of demultiplexing
  - Decentralized architecture to offload CPU
  - Design optimized for fast data path

Second Generation Network Systems (details)

- Multiple network interfaces
  - Powerful NIC
  - Private buffer memory
- High-speed hardware interconnects NICs
- General-purpose processor only handles exceptions
- Sufficient for medium speed interfaces (100 Mbps)

Reminder: Protocol Processing In Second Generation Network Systems



- NIC handles most of layers 1 - 3
- Fast-path forwarding avoids CPU completely

Third Generation Network Systems

- Late 1990s
- Functionality partitioned further
- Additional hardware on each NIC
- Almost all packet processing off-loaded from CPU

Third Generation Design

- NIC contains
  - ASIC hardware
  - Embedded processor plus code in ROM
- NIC handles
  - Classification
  - Forwarding
  - Traffic policing
  - Monitoring and statistics

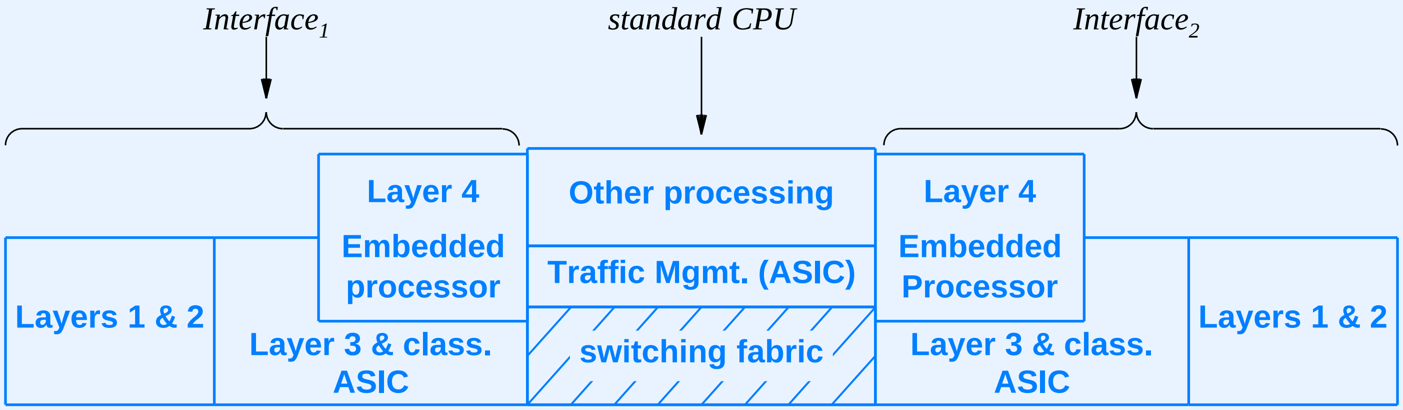
Embedded Processor

- Two possibilities
  - Complex Instruction Set Computer (CISC)
  - Reduced Instruction Set Computer (RISC)
- RISC used often because
  - Higher clock rates
  - Smaller
  - Lower power consumption

Purpose Of Embedded Processor In Third Generation Systems

Third generation systems use an embedded processor to handle layer 4 functionality and exception packets that cannot be forwarded across the fast path. An embedded processor architecture is chosen because ease of implementation and amenability to change are more important than speed.

Protocol Processing In Third Generation Systems



- Special-purpose ASICs handle lower layer functions
- Embedded (RISC) processor handles layer 4
- CPU only handles low-demand processing

Are Third Generation Systems Sufficient?

Are Third Generation Systems Sufficient?

- Almost

Are Third Generation Systems Sufficient?

- Almost … but not quite.

Problems With Third Generation Systems

- High cost
- Long time to market
- Difficult to simulate/test
- Expensive and time-consuming to change
  - Even trivial changes require silicon respin
  - 18-20 month development cycle
- Little reuse across products
- Limited reuse across versions

Problems With Third Generation Systems (continued)

- No consensus on overall framework
- No standards for special-purpose support chips
- Requires in-house expertise (ASIC designers)

A Fourth Generation

- Goal: combine best features of first generation and third generation systems
  - Flexibility of programmable processor
  - High speed of ASICs
- Technology called *network processors*

Definition Of A Network Processor

A network processor *is a special-purpose, programmable hardware device that combines the low cost and flexibility of a RISC processor with the speed and scalability of custom silicon (i.e., ASIC chips). Network processors are building blocks used to construct network systems.*

Network Processors: Potential Advantages

- Relatively low cost
- Straightforward hardware interface
- Facilities to access
  - Memory
  - Network interface devices
- Programmable
- Ability to scale to higher
  - Data rates
  - Packet rates

Network Processors: Potential Advantages

- Relatively low cost
- Straightforward hardware interface
- Facilities to access
  - Memory
  - Network interface devices
- Programmable
- Ability to scale to higher
  - Data rates
  - Packet rates

The Promise Of Programmability

- For producers
  - Lower initial development costs
  - Reuse software in later releases and related systems
  - Faster time-to-market
  - Same high speed as ASICs
- For consumers
  - Much lower product cost
  - Inexpensive (firmware) upgrades

Choice Of Instruction Set

- Programmability alone insufficient
- Also need higher speed
- Should network processors have
  - Instructions for specific protocols?
  - Instructions for specific protocol processing tasks?
- Choices difficult

Instruction Set

- Need to choose one instruction set
- No single instruction set best for all uses
- Other factors
  - Power consumption
  - Heat dissipation
  - Cost
- More discussion later in the course

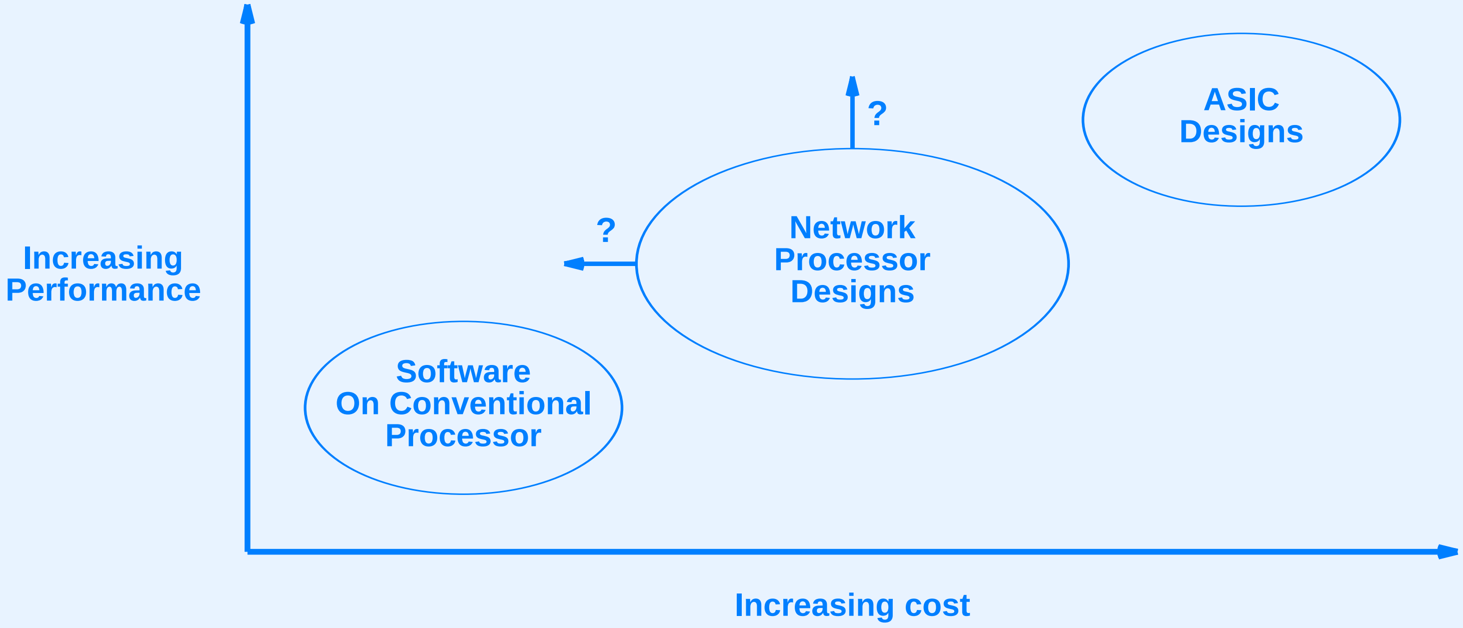
Scalability

- Two primary techniques
  - Parallelism
  - Data pipelining
- Questions
  - How many processors?
  - How should they be interconnected?
- More discussion later

Costs And Benefits Of Network Processors

- Currently
  - More expensive than conventional processor
  - Slower than ASIC design
- Where do network processors fit?
  - Somewhere in the middle

Where Network Processors Fit



- Network processors: the middle ground

Achieving Higher Speed

- What is known
  - Must partition packet processing into separate functions
  - To achieve highest speed, must handle each function with separate hardware
- What is unknown
  - Exactly what functions to choose
  - Exactly what hardware building blocks to use
  - Exactly how building blocks should be interconnected

Variety Of Network Processors

- Economics driving a gold rush
  - NPs will dramatically lower production costs for network systems
  - A good NP design potentially worth lots of $$
- Result
  - Wide variety of architectural experiments
  - Wild rush to try yet another variation

An Interesting Observation

- System developed using ASICs
  - High development cost ($1M)
  - Lower cost to replicate
- System developed using network processors
  - Lower development cost
  - Higher cost to replicate
- Conclusion: amortized cost favors ASICs for most high-volume systems

Summary

- Third generation network systems have embedded processor on each NIC
- Network processor is programmable chip with facilities to process packets faster than conventional processor
- Primary motivation is economic
  - Lower development cost than ASICs
  - Higher processing rates than conventional processor

Questions?

XII

The Complexity Of Network Processor Design

How Should A Network Processor Be Designed?

- Depends on
  - Operations network processor will perform
  - Role of network processor in overall system

Goals

- Generality
  - Sufficient for all protocols
  - Sufficient for all protocol processing tasks
  - Sufficient for all possible networks
- High speed
  - Scale to high bit rates
  - Scale to high packet rates
- Elegance
  - Minimality, not merely comprehensiveness

The Key Point

A network processor is not designed to process a specific protocol or part of a protocol. Instead, designers seek a minimal set of instructions that are sufficient to handle an arbitrary protocol processing task at high speed.

Network Processor Design

- To understand network processors, consider problem to be solved
  - Protocols being implemented
  - Packet processing tasks

Packet Processing Functions

- Error detection and correction
- Traffic measurement and policing
- Frame and protocol demultiplexing
- Address lookup and packet forwarding
- Segmentation, fragmentation, and reassembly
- Packet classification
- Traffic shaping
- Timing and scheduling
- Queueing
- Security: authentication and privacy

Questions

- Does our list of functions encompass all protocol processing?
- Which function(s) are most important to optimize?
- How do the functions map onto hardware units in a typical network system?
- Which hardware units in a network system can be replaced with network processors?
- What minimal set of instructions is sufficiently general to implement all functions?

Division Of Functionality

- Partition problem to reduce complexity
- Basic division into two parts
- Functions applied when packet arrives known as

  ingress processing

- Functions applied when packet leaves known as

  egress processing

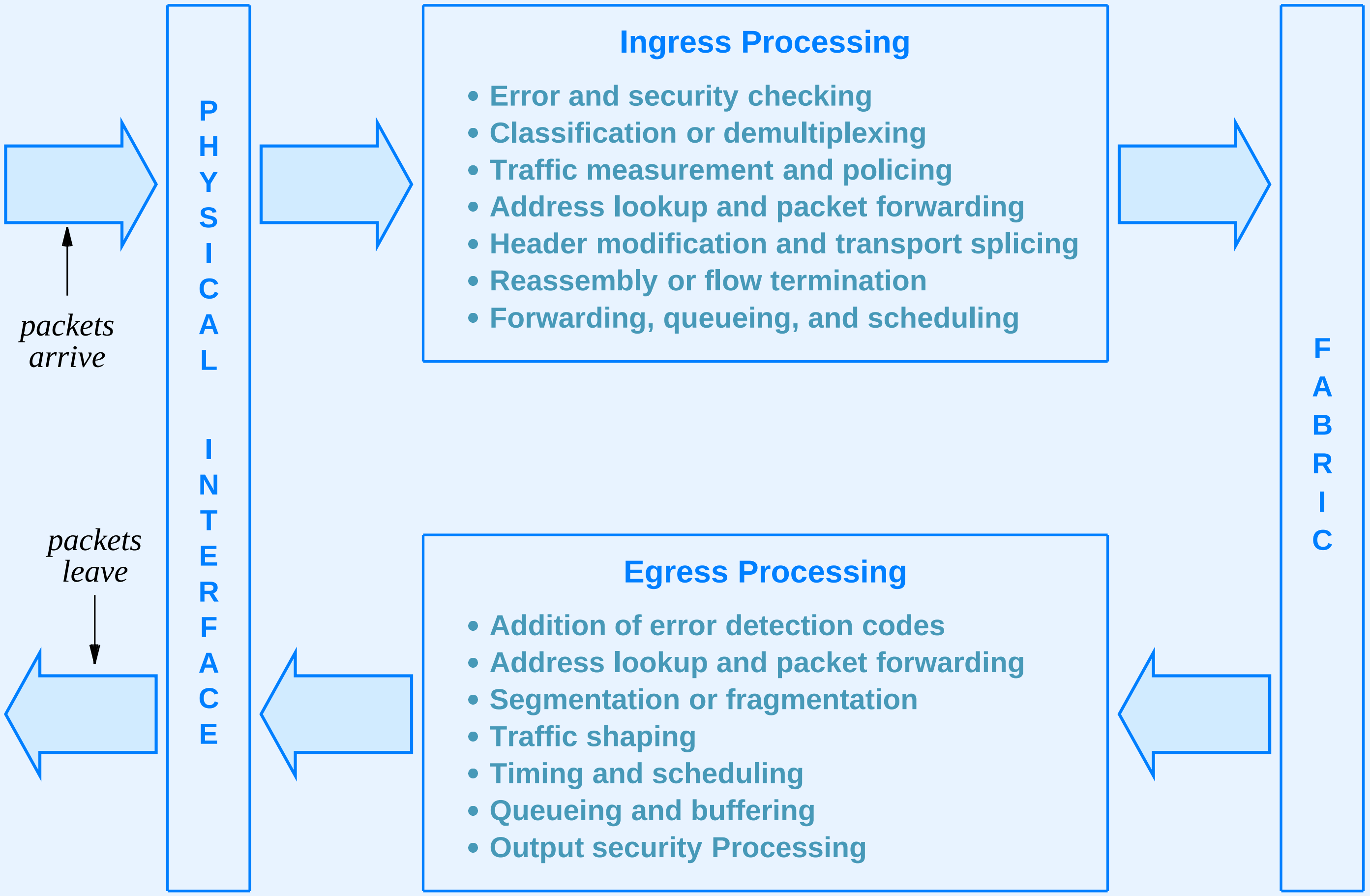
Ingress Processing

- Security and error detection
- Classification or demultiplexing
- Traffic measurement and policing
- Address lookup and packet forwarding
- Header modification and transport splicing
- Reassembly or flow termination
- Forwarding, queueing, and scheduling

Egress Processing

- Addition of error detection codes
- Address lookup and packet forwarding
- Segmentation or fragmentation
- Traffic shaping
- Timing and scheduling
- Queueing and buffering
- Output security processing

Illustration Of Packet Flow



packets arrive

PHYSICAL INTERFACE

Ingress Processing

- Error and security checking
- Classification or demultiplexing
- Traffic measurement and policing
- Address lookup and packet forwarding
- Header modification and transport splicing
- Reassembly or flow termination
- Forwarding, queueing, and scheduling

FABRIC

Egress Processing

- Addition of error detection codes
- Address lookup and packet forwarding
- Segmentation or fragmentation
- Traffic shaping
- Timing and scheduling
- Queueing and buffering
- Output security Processing

packets leave

A Note About Scalability

Unlike a conventional processor, scalability is essential for network processors. To achieve maximum scalability, a network processor offers a variety of special-purpose functional units, allows parallel or pipelined execution, and operates in a distributed environment.

How Will Network Processors Be Used?

- For ingress processing only?
- For egress processing only?
- For combination?

How Will Network Processors Be Used?

- For ingress processing only?
- For egress processing only?
- For combination?
- Answer: No single role

Potential Architectural Roles For Network Processor

- Replacement for a conventional CPU
- Augmentation of a conventional CPU
- On the input path of a network interface card
- Between a network interface card and central interconnect
- Between central interconnect and an output interface
- On the output path of a network interface card
- Attached to central interconnect like other ports

An Interesting Potential Role For Network Processors

In addition to replacing elements of a traditional third generation architecture, network processors can be attached directly to a central interconnect and used to implement stages of a macroscopic data pipeline. The interconnect allows forwarding among stages to be optimized.

Conventional Processor Design

- Design an instruction set, *S*
- Build an emulator/simulator for *S* in software
- Build a compiler that translates into *S*
- Compile and emulate example programs
- Compare results to
  - Extant processors
  - Alternative designs

Network Processor Emulation

- Can emulate low-level logic (e.g., Verilog)
- Software implementation
  - Slow
  - Cannot handle real packet traffic
- FPGA implementation
  - Expensive and time-consuming
  - Difficult to make major changes

Network Processor Design

- Unlike conventional processor design
- No existing code base
- No prior hardware experience
- Each design differs

Hardware And Software Design

Because a network processor includes many low-level hardware details that require specialized software, the hardware and software designs are codependent; software for a network processor must be created along with the hardware.

Summary

- Protocol processing divided into ingress and egress operations
- Network processor design is challenging because
  - Desire generality and efficiency
  - No existing code base
  - Software designs evolving with hardware

Questions?

XIII

Network Processor Architectures

Architectural Explosion

An excess of exuberance and a lack of experience have produced a wide variety of approaches and architectures.

Principle Components

- Processor hierarchy
- Memory hierarchy
- Internal transfer mechanisms
- External interface and communication mechanisms
- Special-purpose hardware
- Polling and notification mechanisms
- Concurrent and parallel execution support
- Programming model and paradigm
- Hardware and software dispatch mechanisms

Processing Hierarchy

- Consists of hardware units
- Performs various aspects of packet processing
- Includes onboard and external processors
- Individual processor can be
  - Programmable
  - Configurable
  - Fixed

Typical Processor Hierarchy

Level	Processor Type	Programmable?	On Chip?
8	General purpose CPU	yes	possibly
7	Embedded processor	yes	typically
5	I/O processor	yes	typically
6	Coprocessor	no	typically
4	Fabric interface	no	typically
3	Data transfer unit	no	typically
2	Framer	no	possibly
1	Physical transmitter	no	possibly

Memory Hierarchy

- Memory measurements
  - Random access latency
  - Sequential access latency
  - Throughput
  - Cost
- Can be
  - Internal
  - External

Typical Memory Hierarchy

Memory Type	Rel. Speed	Approx. Size	On Chip?
Control store	100	10^3	yes
G.P. Registers†	90	10^2	yes
Onboard Cache	40	10^3	yes
Onboard RAM	7	10^3	yes
Static RAM	2	10^7	no
Dynamic RAM	1	10^8	no

Internal Transfer Mechanisms

- Internal bus
- Hardware FIFOs
- Transfer registers
- Onboard shared memory

External Interface And Communication Mechanisms

- Standard and specialized bus interfaces
- Memory interfaces
- Direct I/O interfaces
- Switching fabric interface

Example Interfaces

- System Packet Interface Level 3 or 4 (SPI-3 or SPI-4)
- SerDes Framer Interface (SFI)
- CSIX fabric interface

Note: The Optical Internetworking Forum (OIF) controls the SPI and SFI standards.

Polling And Notification Mechanisms

- Handle asynchronous events
  - Arrival of packet
  - Timer expiration
  - Completion of transfer across the fabric
- Two paradigms
  - Polling
  - Notification

Concurrent Execution Support

- Improves overall throughput
- Multiple threads of execution
- Processor switches context when a thread blocks

Support For Concurrent Execution

- Embedded processor
  - Standard operating system
  - Context switching in software
- I/O processors
  - No operating system
  - Hardware support for context switching
  - Low-overhead or zero-overhead

Concurrent Support Questions

- Local or global threads (does thread execution span multiple processors)?
- Forced or voluntary context switching (are threads preemptable)?

Hardware And Software Dispatch Mechanisms

- Refers to overall control of parallel operations
- Dispatcher
  - Chooses operation to perform
  - Assigns to a processor

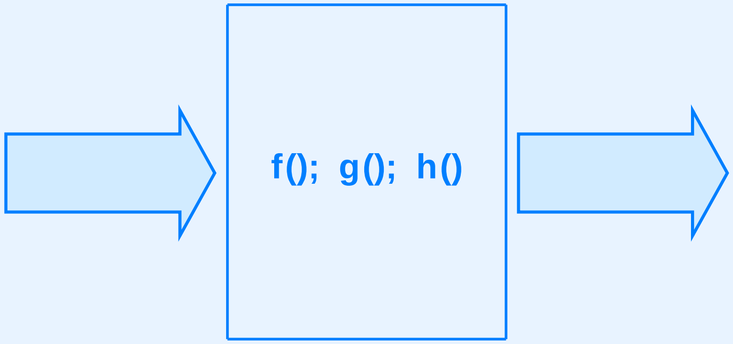
Implicit And Explicit Parallelism

- Explicit parallelism
  - Exposes parallelism to programmer
  - Requires software to understand parallel hardware
- Implicit parallelism
  - Hides parallel copies of functional units
  - Software written as if single copy executing

Architecture Styles, Packet Flow, And Clock Rates

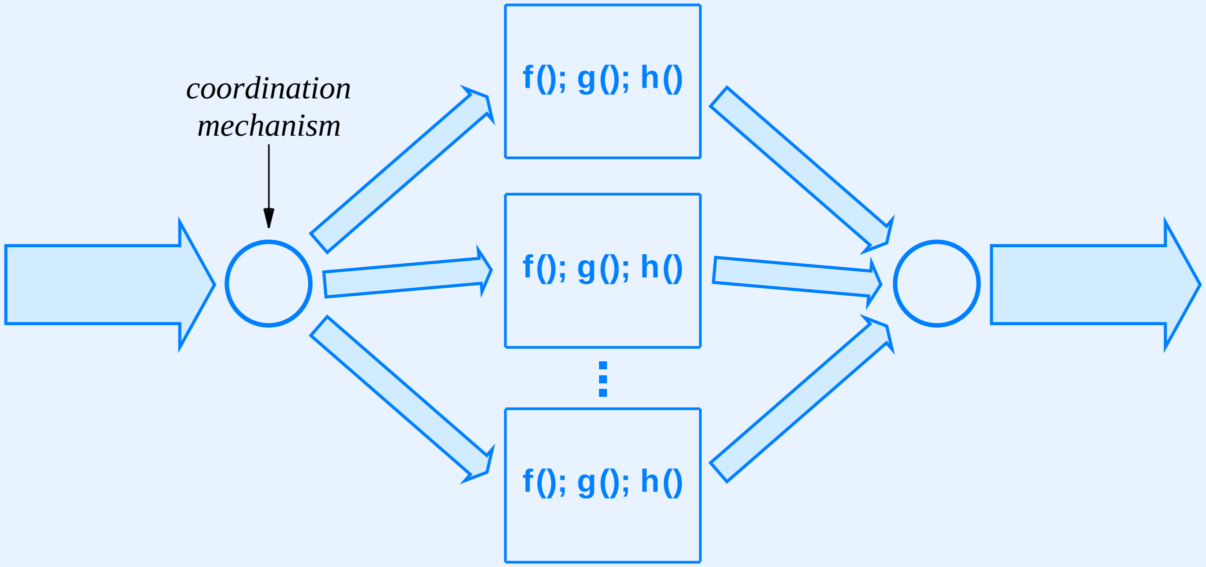
- Embedded processor plus fixed coprocessors
- Embedded processor plus programmable I/O processors
- Parallel (number of processors scales to handle load)
- Pipeline processors
- Dataflow

Embedded Processor Architecture



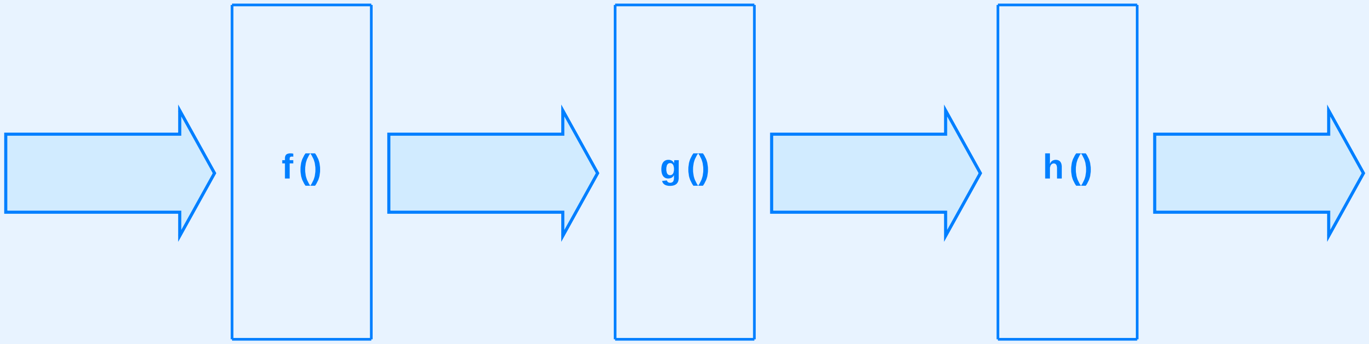
- Single processor
  - Handles all functions
  - Passes packet on
- Known as run-to-completion

Parallel Architecture



- Each processor handles 1/N of total load

Pipeline Architecture



- Each processor handles one function
- Packet moves through ‘‘pipeline’’

Clock Rates

- Embedded processor runs at > wire speed
- Parallel processor runs at < wire speed
- Pipeline processor runs at wire speed

Software Architecture

- Central program that invokes coprocessors like subroutines
- Central program that interacts with code on intelligent, programmable I/O processors
- Communicating threads
- Event-driven program
- RPC-style (program partitioned among processors)
- Pipeline (even if hardware does not use pipeline)
- Combinations of the above

Example Uses Of Programmable Processors

General purpose CPU

- Highest level functionality
- Administrative interface
- System control
- Overall management functions
- Routing protocols

Embedded processor

- Intermediate functionality
- Higher-layer protocols
- Control of I/O processors
- Exception and error handling
- High-level ingress (e.g., reassembly)
- High-level egress (e.g., traffic shaping)

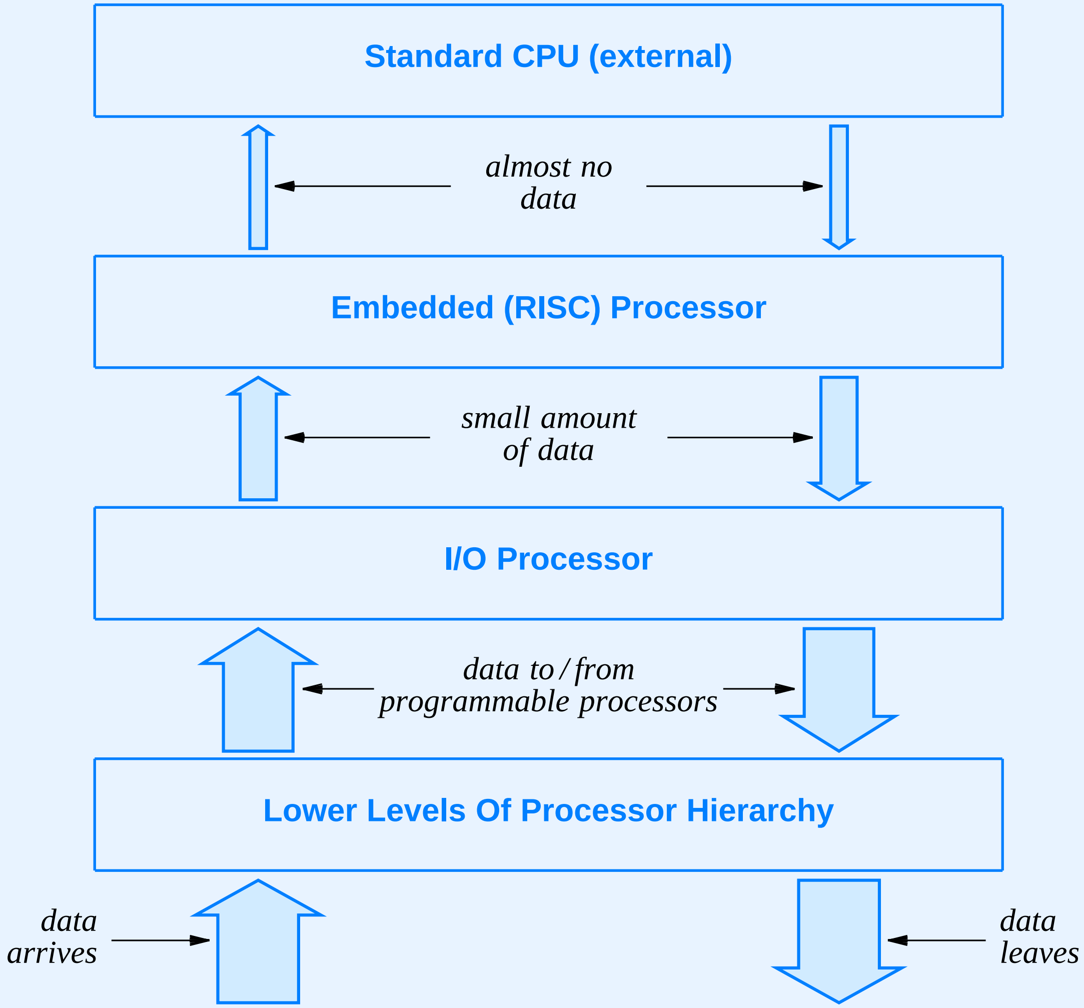
I/O processor

- Basic packet processing
- Classification
- Forwarding
- Low-level ingress operations
- Low-level egress operations

Using The Processor Hierarchy

To maximize performance, packet processing tasks should be assigned to the lowest level processor capable of performing the task.

Packet Flow Through The Hierarchy



Standard CPU (external)

almost no data

Embedded (RISC) Processor

small amount of data

I/O Processor

data to / from programmable processors

Lower Levels Of Processor Hierarchy

data arrives

data leaves

Summary

- Network processor architectures characterized by
  - Processor hierarchy
  - Memory hierarchy
  - Internal buses
  - External interfaces
  - Special-purpose functional units
  - Support for concurrent or parallel execution
  - Programming model
  - Dispatch mechanisms

Questions?

XVII

Overview Of The Agere Network Processor

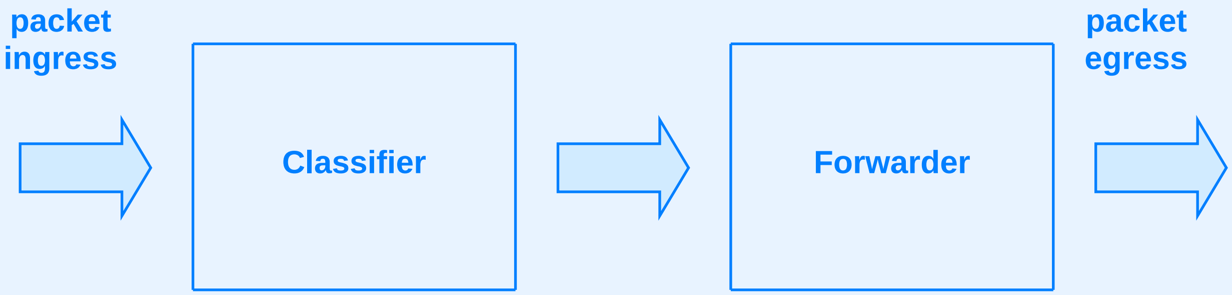
An Example Network Processor

- We will
  - Choose one example
  - Examine the hardware
  - Understand the programming model
  - Consider the capabilities and limitations
- Our choice for this course: Agere Systems APP550

Agere Hardware Organization

- Conceptual pipeline
- Three major blocks
  - Classifier
  - Forwarder
  - State Engine

Illustration Of Hardware Pipeline



- All packets flow through classifier and forwarder

Classifier

- Classifies packets or cells
- Implemented with pattern matching engine
- Passes packet to forwarder along with classification
- On fast path

Classifier

- Classifies packets or cells
- Implemented with pattern matching engine
- Passes packet to forwarder along with classification
- On fast path

State Engine

- Invoked by classifier
- Provides host interface
- Collects statistics
- Not on fast path

Forwarder

- Receives packets from the classifier
- Performs traffic management and shaping
- Handles scheduling and packet queueing
- Modifies the packet if necessary
- Programmed via scripts
- On fast path

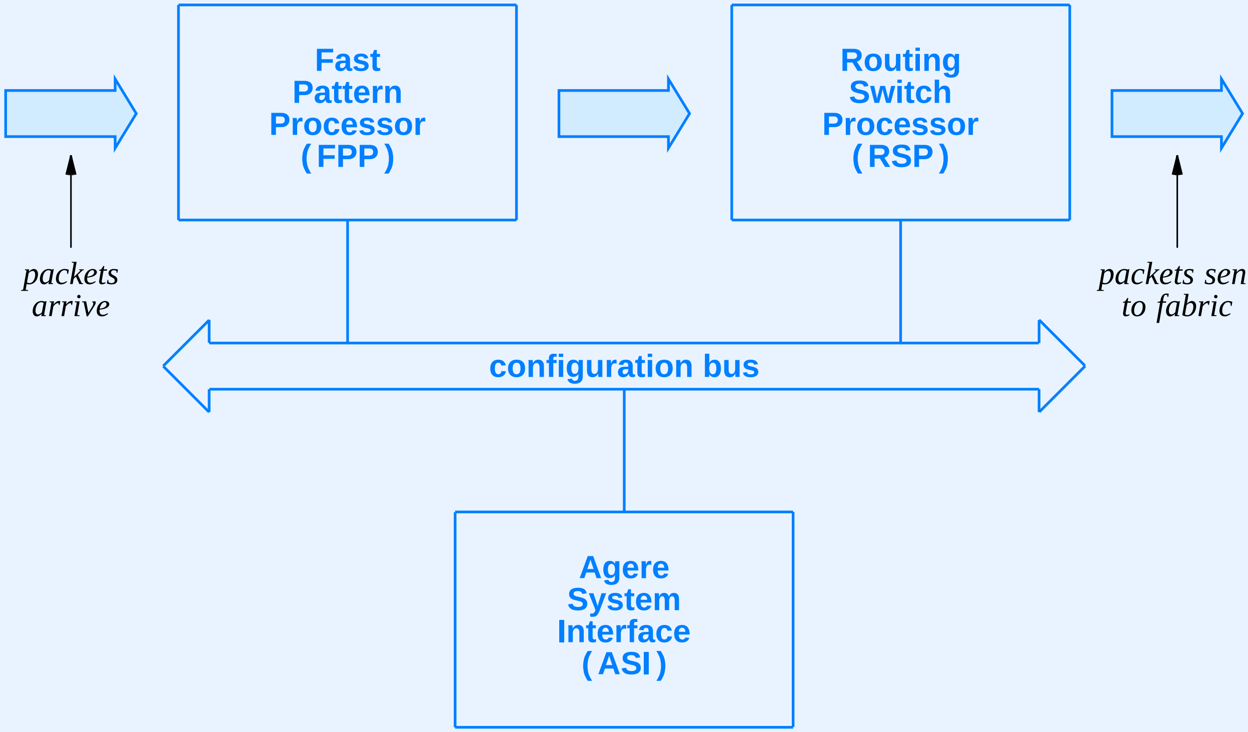
Forwarder

- Receives packets from the classifier
- Performs traffic management and shaping
- Handles scheduling and packet queueing
- Modifies the packet if necessary
- Programmed via scripts
- On fast path

Original Agere Design

- Among the first network processors
- Named *PayloadPlus 2.5*
- Handled 2.5 Gbps
- Three separate chips
  - Fast Pattern Processor (FPP)
  - Routing Switching Processor (RSP)
  - Agere System Interface (ASI)

Illustration Of First-Generation Design



Second Generation Agere Design

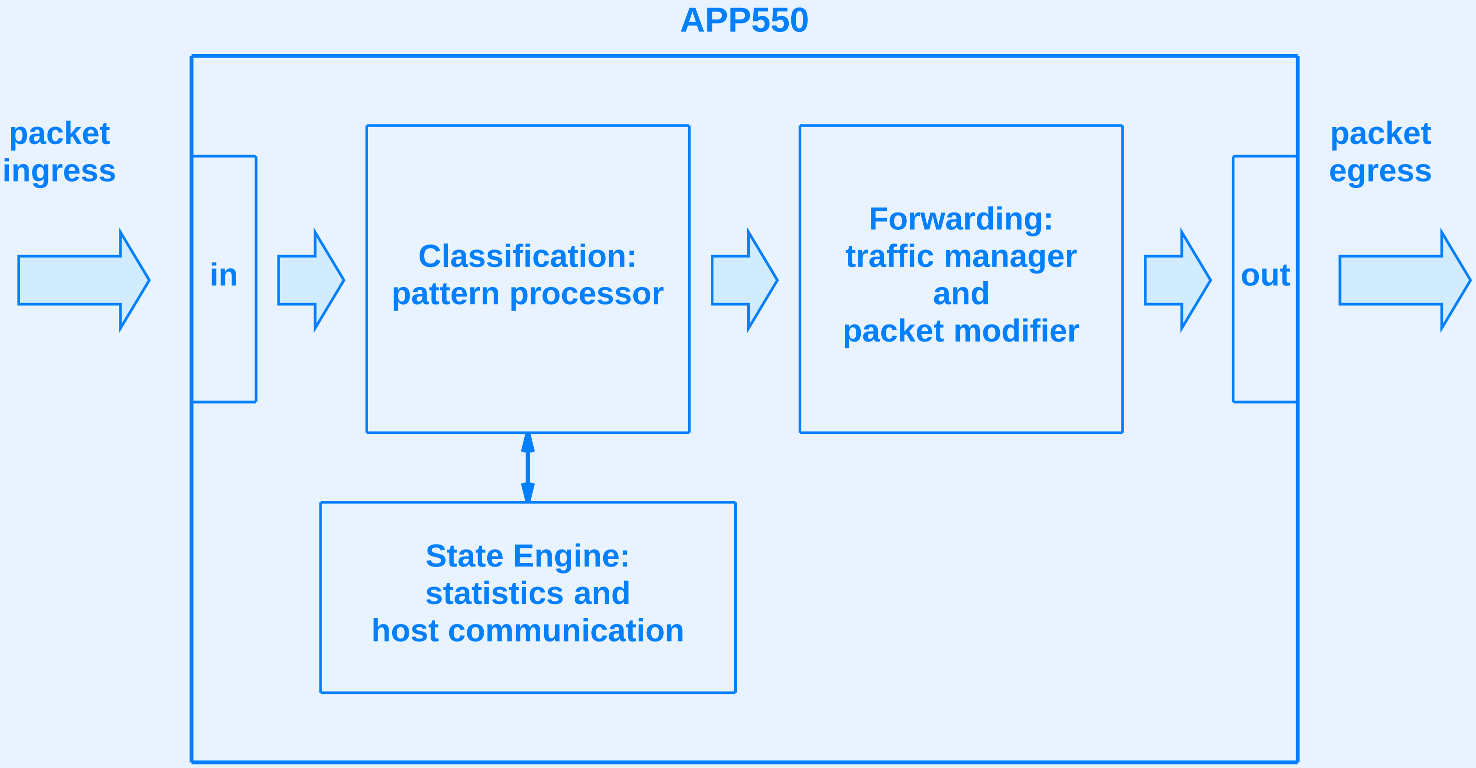
- Same basic architecture as first generation
- All three functions combined onto single chip
- Internal communication paths provide faster pipeline
- Designed to handle 10 Gbps
- Multiple models

Agere Second Generation Models

Model	Throughput	Features
APP520	5+ Gbps	2 GigE ports, no external memory
APP530	2.5 Gbps	Slower speed version of the 550
APP530TM	2.5 Gbps	530 plus traffic management software
APP540	5+ Gbps	Packet traffic with no external reassembly
APP550	5+ Gbps	4 GigE ports, full capability
APP750	10.0 Gbps	Higher speed than a 550

- Models 520 and 540 provide traffic management only
- We will focus on the APP550

Illustration Of APP550 Architecture


APP550
packet ingress
in
Classification: pattern processor
Forwarding: traffic manager and packet modifier
out
packet egress
State Engine: statistics and host communication

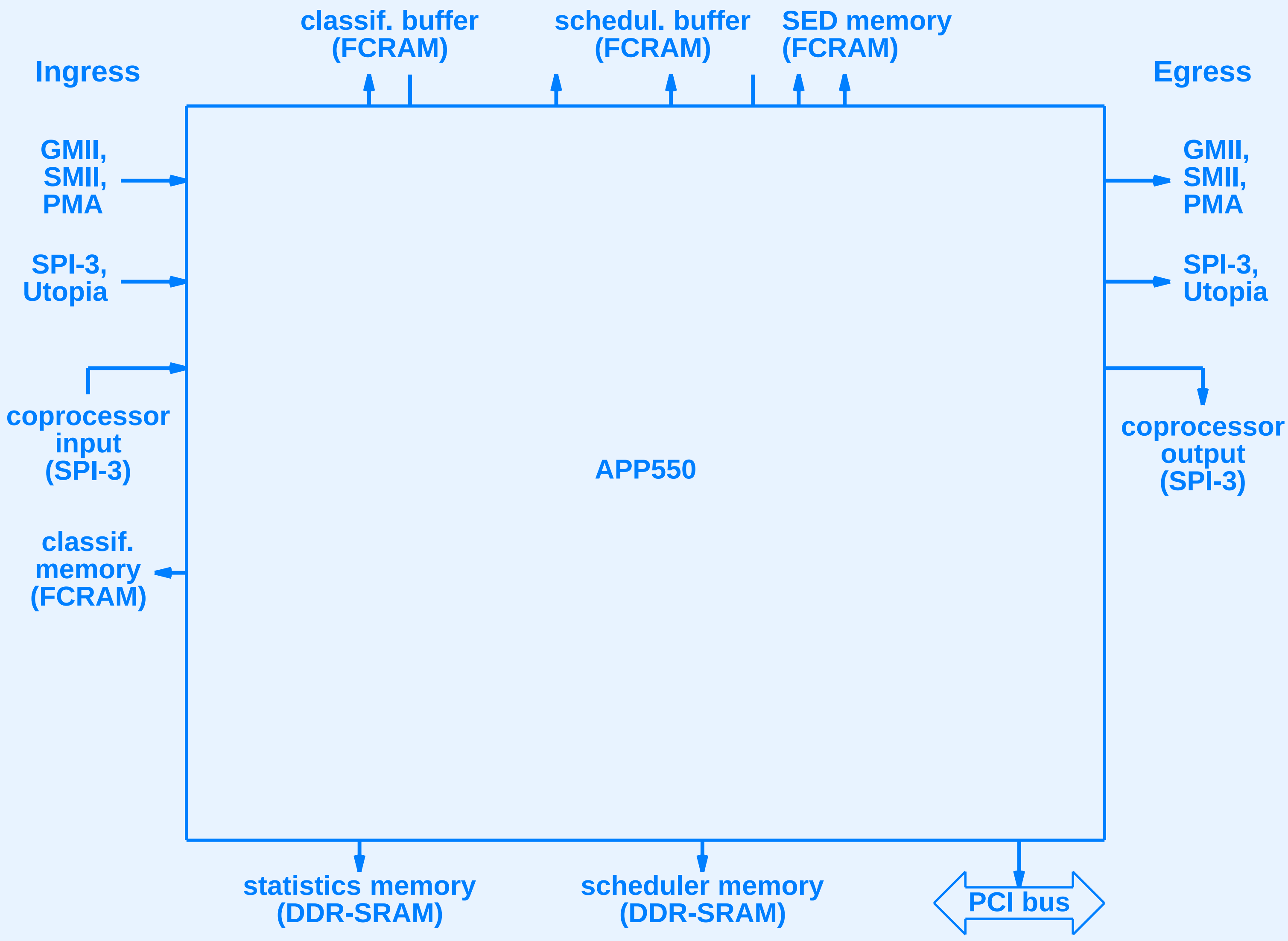
APP550 Features

- Unconventional design with special-purpose hardware
- Programmable using high-level languages
- Specialized onboard engines for protocol processing tasks
- Connection for external coprocessor
- Hardware support for classification, scheduling, policing, shaping, and packet modification
- Interfaces for various physical media

APP550 External Interfaces

- Multiple memory interfaces
- Coprocessor interface
- Multiple packet I/O interfaces
- Control interfaces
  - External scheduling
  - Host processor (PCI bus)

Illustration Of APP550 External Connections



Purpose Of External Connections

Interface To	Purpose
memory	Access to storage for packet buffers, queues, instructions, and other parameters
media	Packet or cell ingress from physical network or egress to physical network
switching fabric	Packet transfer to an output port
PCI bus	Allows host computer to control the APP550
scheduler	Access to external scheduler
coprocessor	Access to external coprocessor hardware

External Media Interface Hardware

- Divides packet into 64-byte blocks
- Delivers one block at a time
- Sends additional information
  - Bit to indicate first block of packet
  - Bit to indicate last block of packet
  - Integer to indicate size of block (64 except for final block)
  - Integer to indicate interface over which block arrived
- Note: if packet fits into single block, bits indicate both ‘‘first’’ and ‘‘last’’ block of packet

APP550 Media Interfaces

Standard Name	Meaning
GMII	Gigabit Media Independent Interface
PMII	Physical Media Independent Interface
SMII	Serial Media Independent Interface
PMA	Physical Medium Attachment
SPI-3	System Packet Interface Level 3
Utopia	Universal Test and Operations PHY Interface for ATM

APP550 External Memory Uses

Engine	Memory	Use
Classifier	FCRAM	Packet buffer memory
	FCRAM	Program memory (patterns)
	FCRAM	Control memory
State Engine	DDR-SRAM	Flow statistics and profile memory
	DDR-SRAM	OAM data memory
Stream Editor	DDR-SRAM	Context memory
	FCRAM	Parameter memory
Reorder Buffer and Shaper	FCRAM	Scheduler buffer memory
	DDR-SRAM	Scheduler linked list memory
	DDR-SRAM	Scheduler parameter memory
	DDR-SRAM	Scheduler queue memory

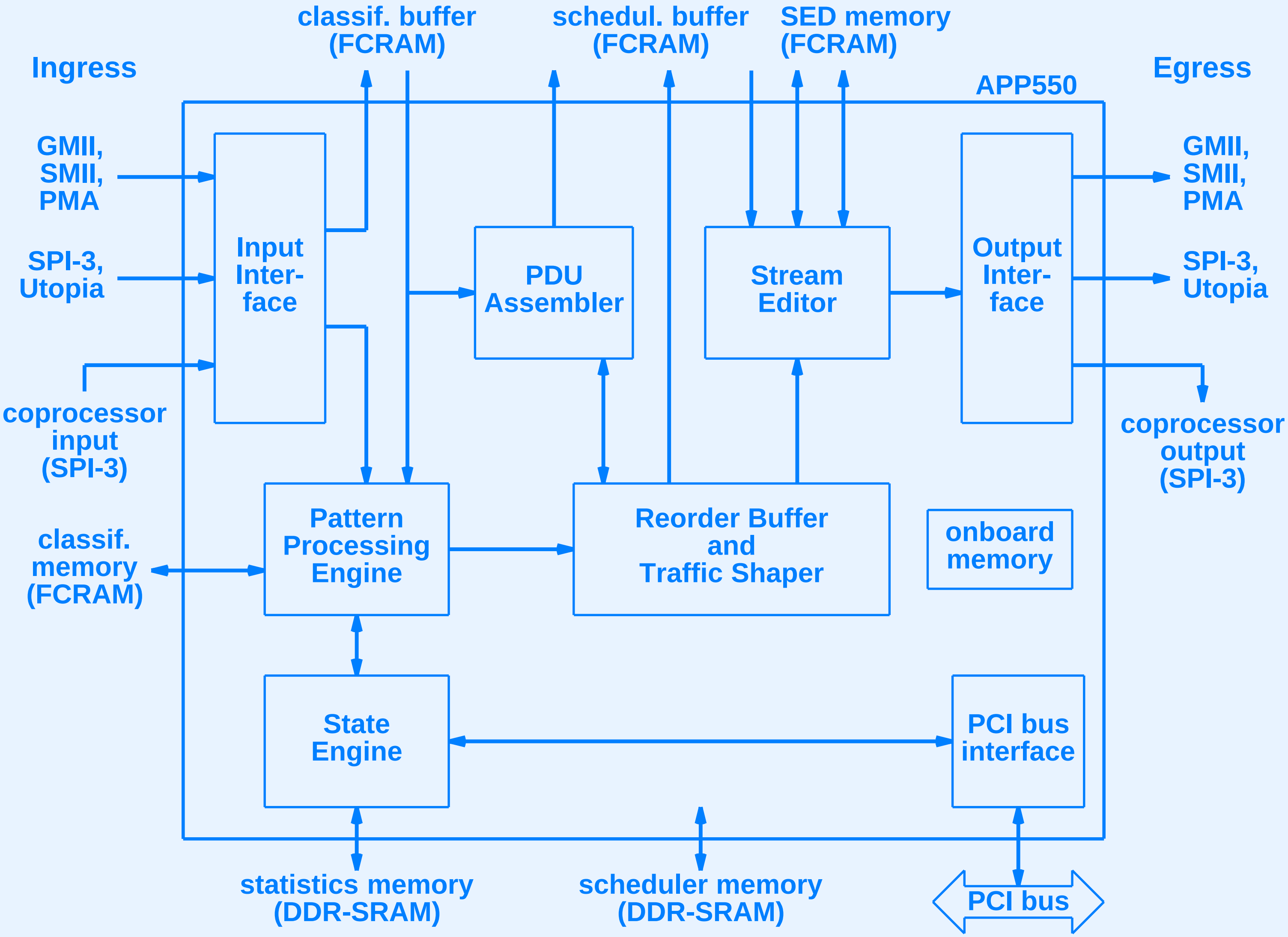
APP550 External Memory Technologies

- Fast Cycle Random Access Memory (FCRAM)
  - Low cycle time allows rapid storage of sequential bytes
  - Used for packet storage
- Double Data Rate Static Random Access Memory (DDR-SRAM)
  - Low latency for random data access
  - Used for table lookup

APP550 Internal Architecture

- Multiple onboard engines
  - Some programmable
  - Some configurable
- I/O interface units handle
  - Cells
  - Frames
- External memory interface units
- Onboard memory

Illustration Of APP550 Internal Architecture



- Many internal connections not shown

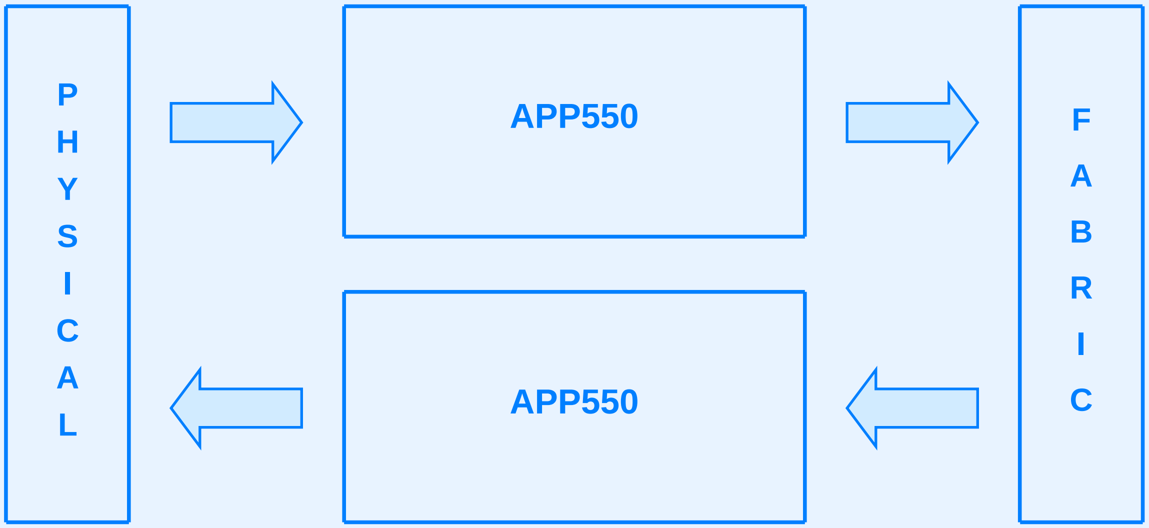
Example Engines On The APP550

Engine	Purpose
Pattern Processing Engine	Classification
State Engine	Gathering state information for scheduling and verifying flow is within bounds
Reorder Buffer Manager	Ensure packet order
PDU Assembler	Collect all blocks of a frame
Traffic Manager	Schedule packets and shape traffic flow
Stream EDitor (SED)	Modify outgoing packet

Full-Duplex Operation

- Single APP550 does not have sufficient capacity to handle flow in two directions
- Solution: use two APP550s per physical interface
  - One handles ingress
  - One handles egress

Illustration Of Two APP550s Used For Full Duplex



- Note: egress processor does not need classification

Questions?

XVIII

Functional Units
On
The Agere APP550

Major Functional Units On the APP550

Unit	Programming Language	Purpose
Buffer Manager	C-NP	Store or drop packets
Input Interface	none	Interface to network devices
Pattern Processor	FPL	Classification
PDU Assembler	none	Collect blocks of a frame
Policing/OAM Engine	C-NP	Gather statistics and administrative functions
Reorder Buffer	none	Ensure packet order
Stream Editor	C-NP	Modify outgoing packet
Traffic Shaper	C-NP	Shape outgoing flows

Input Interface

- Receives packets from
  - Up to four high-speed sources
  - Coprocessor
  - Fabric
  - host via PCI bus
- Passes packets to classifier

Input Interface
(continued)

- Configurable for
  - SPI-3
  - UTOPIA 2 or UTOPIA 3
  - PLATO
- Up to 256 MultiPHY (MPHY) addresses

Pattern Processing Engine (PPE)

- Used for classification
- Unconventional architecture (no fetch-execute)
- Implements form of pattern matching
- Programmable using FPL (more later)
- Uses implicit parallelism
- Invoked automatically when packet arrives

Pattern Processing Engine (PPE)
(continued)

- Can access multiple memories
  - Classifier PDU Buffer (CPDUB)
  - Classifier Program Memory (CPM)
  - Classifier Control Memory (CCM)

PDU Assembler

- Accepts blocks of data from Pattern Processing Engine
- Combines blocks to form a complete packet

Data Flow Through Classifier Block

- Ingress storage
  - Packet divided into blocks
  - Blocks written to Classification PDU buffer (CPDUB)
- First pass of classification
  - Blocks pass through PPE
  - PPE runs root program on each block

Data Flow Through Classifier Block (continued)

- Second pass of classification
  - Packet passes through PPE
  - PPE runs replay program on packet
- Reassembly and handoff
  - Needed for implicit parallelism
  - Packets emitted in order
  - ReOrder Buffer (*ROB*) used

A Key Idea

Each packet passes through the PPE twice: in the first pass, a root program is invoked once for each sixty-four byte block of the packet, and in the second pass a replay program is invoked once for the entire packet.

State Engine

- Invoked by Pattern Processing Engine
- Collects statistics needed for policing
- Provide host interface
- Configures and controls other functional units
- Operates like coprocessor (function call)
- Collects statistics for policing and traffic management functions
- Programmable
- Programming language is ASL, a.k.a. C-NP (more later)

Reorder Buffer

- Associated with buffer manager
- Needed because classifier has
  - Implicit parallelism
  - Variable processing time (depends on the packet)
- Purpose: guarantee packet order unchanged
  - Required for some protocols (e.g., ATM)
  - Beneficial for other protocols (e.g., TCP)

Reorder Buffer
(continued)

- Invoked after classification
- Not programmable

Function Of the Reorder Buffer

The Reorder Buffer Manager provides transfer between the Classifier and Traffic Manager. The Traffic Manager extracts packets from the ROB table in sequential order, waiting for the Classifier to finish processing a packet, even if later packets are already finished.

Traffic Manager Block

- On the fast path
- Programmable with scripts
- TM script invoked once per packet
- Receives packets from the Classifier
- Polices traffic
- Queues packets
- Schedules and shapes traffic
- Modifies outgoing packets

Traffic Manager Block
(continued)

- Transmits each packet on the appropriate output port
- Several subsystems
- Uses two types of scripts (programs)
  - *Buffer manager script* invoked when packet arrives
  - *Scheduler script* invoked to select packets for egress

Buffer Manager

- Part of Traffic Manager block
- Receives packets from classifier
- Decides to enqueue or discard packet
- Decision based on
  - Memory use
  - Thresholds
  - Classification and / or policing results

Traffic Shaper

- Part of Traffic Manager block
- Invoked for outgoing packet
- Programmable
- Programming language is ASL, a.k.a. C-NP (more later)
- Handles hierarchy of output queues
- Scheduling based on
  - Time
  - Current congestion
  - Classification results
  - Priorities

Stream editor (SED)

- Part of Traffic Manager block
- Invoked for outgoing packet
- Handles packet modification / update
- Not a conventional processor
- Changes specified with parameters
- Example: compute checksum

Stream editor (SED)
(continued)

- Capabilities
  - Create up to 127 bytes of frame header, and prepend to frame
  - Create up to 20 bytes of cell header, and prepend to block
  - Update items in the packet such as the TTL or checksum
  - Change an MPHY address.
- Parameters used to control processing
  - HeaderDeltaParameters
  - SED parameters

Programming, Performance, And Global Pulse Rate

- To process packets at wire speed, cannot spend too long on any packet
- Clock rate of APP550 is fixed at 266 MHz
- For given interface speed, can compute maximum cycles available per packet
- Agere uses term *Global Pulse*
- Example: for OC-48 interface, global pulse is 23 instructions
- Compiler flags program that exceeds global pulse

Global Pulse And SED Engine

- SED engine
  - Exception to global pulse
  - Can execute twice as many instructions as other engines
- Reason: hardware provides two copies of the SED engine (implicit parallelism)

Other Functional Units

- External Scheduling Interface
- Configuration Bus Interface (CBI)
- Packet Generation Engine
- Output Interface

Summary

- APP550 contains many functional units
- Some units are programmable; others are not
- Classifier, Traffic Manager, and State Engine handle basic packet processing functions
- All packets proceed through the Pattern Processing Engine
- PPE uses the State Engine to collect statistics
- Disposition decisions made in the Traffic Manager

Questions?

XIX

Reference Platform And Simulator (HDS, SPA)

Reference System

- Provided by vendor
- Targeted at potential customers
- Usually includes
  - Hardware testbed
  - Development software
  - Simulator or emulator
  - Download and bootstrap software
  - Reference implementations

Simulation Vs. Emulation

- Simulator
  - Software that mimics external actions of a network processor
  - Usually runs on conventional computer (e.g., PC)
  - Takes program and sequence of packets as input
- Emulator
  - Software that mimics internal actions of a network processor
  - Attempts to be *cycle accurate*
  - Usually runs on conventional computer (e.g., PC)
  - Takes program and sequence of packets as input

Simulation Vs. Emulation
(continued)

- Simulator
  - Not as accurate, but faster
- Emulator
  - Not as fast, but more accurate

Agere Reference System

- Software Development Environment (SDE)
- Hardware Development System (HDS)
- Run Time Environment (RTE)

Software Development Environment (SDE)

- Used by programmer to prepare and test software
- Runs on conventional computer

Agere SDE Components

- FPL compiler
- FPL source code optimizer
- FPL debugger
- C-NP compiler
- Configuration generator
- Simulator
- Traffic generator
- Traffic analyzer and plotter

System Performance Analyzer (SPA)

- Graphical User Interface (GUI) for tools in the SDE
- Permits programmer to compile, test, and debug APP550 software
- Can invoke the simulator, generate traffic, and allow a programmer to monitor the results

Hardware Development System (HDS)

- Hardware testbed
- Manufactured as a stand-alone system (chassis)
- Composed of three boards
- Includes cross-development and downloading facilities

The Three Boards In Agere's HDS

Port card

An Agere APP550 with associated memory and a connection to the HDS bus over which the APP550 can access I/O ports.

I/O card

Facilities for packet input and output: four 1-Gbps optical Ethernet connections and an OC-48 TADM connection that can be configured as an OC-48C connection or a mixture of up to four OC-12 ATM and/or Packet Over SONET connections.

CPU card

A PowerPC processor used to control the APP550, RAM, ROM, and PROM memories, a serial port, an Ethernet connection, and a debugging port.

HDS Bootstrap And Operation

- HDS does contain
  - Conventional processor (PowerPC)
  - Ethernet interface
- HDS does not contain
  - Stable storage (e.g., a disk)
- Can still run conventional OS

Five-Step HDS Bootstrap Procedure

1. CPU card runs an initial bootstrap program from ROM.
2. Boot program obtains address of TFTP server either from PROM or via BOOTP, and runs TFTP to obtain an operating system image (embedded Linux or VxWorks), which is loaded into memory. All communication proceeds over the Ethernet interface on the CPU card.
3. When it boots, the operating system creates a RAM disk. The operating system uses NFS to mount a remote file system. Once the operating system is running, a user can log in and receive a shell prompt for command input.

Five-Step HDS Bootstrap Procedure

4. The operating system proceeds to load a set of libraries and functions that comprise the API (VxWorks) or a kernel module that can load libraries on demand (Linux).

5. One of the API functions downloads the *Agere Configuration Image* onto the Port card. The configuration image contains parameters for the APP550 as well as compiled code from FPL and C-NP (i.e., data and instructions for the APP550).

5. Another of the API functions downloads values into the FPGA on the I/O card.

6. The programmer uses the API (from a command or from a program) to interact with the APP550.

Testing Paradigms

- APP550 does not
  - Provide convenient, efficient host interface
  - Support for instrumentation of code
- Hardware testbed allows programmer to
  - Test at wire speed
  - Measure with actual traffic

Testing Paradigms
(continued)

- Simulator allows programmer to
  - Control input
  - Step through a program
  - Generate arbitrary flows (e,g, to test queueing or scheduling)

Summary

- Reference systems
  - Provided by vendor
  - Targeted at potential customers
  - Usually include
    - Hardware testbed
    - Cross-development software
    - Download and bootstrap software
    - Reference implementations

Summary
(continued)

- Agere offers
  - Hardware Development System (HDS)
  - Software Development Environment
- System Performance Analyzer (SPA) is graphical interface for reference platforms

Questions?

XXI

State Engine And Scripting Language (C-NP)

We Will Discuss

- State Engine role and interface
- C-NP programming language

State Engine Functionality

- Contains memory used to store statistics
- Performs computations needed for traffic policing
- Provides interface to external host processor

Memory For Statistics

- State Engine provides
  - Interface to memory
  - Memory access functions in FPL (e.g., store or increment)
- Intended to be used with classifier
- Notes
  - Statistics needed for monitoring and control
  - Classifier hardware has no memory except for packets
  - FPL cannot have static variables

Computations For Traffic Policing

- Determine whether flow exceeds *performance profile*
- Results passed to Traffic Manager for drop decision
- Performed by *Policing Engine*

Policing Engine

- Part of the State Engine
- Programmable via scripts
- Language is *C for Network Processors* (*C-NP*), which was formerly known as *Agere Scripting Language* (*ASL*)

Interface To Host Processor

- External host necessary
- Functionality
  - Overall control
  - Chip configuration and initialization
  - Dynamic updates to runtime data structures
  - Handling traffic on the slow path
    - Routing protocols
    - Exceptions

Interface To Host Processor
(continued)

- Physical interconnection to external host
  - *Peripheral Component Interconnect* (*PCI*) bus
  - Terminated by State Engine hardware
- Logical interconnection
  - Internal *Configuration Bus Interconnect* (*CBI*) connects State Engine to Classifier and Traffic Manager
  - External (PCI) and internal (CBI) buses are mapped

Communication Paradigm

- Standard bus paradigm
- Host issues fetch or store operation
- APP550 hardware provides large set of *Control and Status Registers* (*CSRs*)
- Semantics of each CSR defined independently
  - Meaning of fetch
  - Meaning of store

CSRs On The APP550

- Separate groups of CSRs for
  - State Engine
  - Classifier
  - Traffic Manager
  - Internal Memory
  - MAC interfaces
- State Engine maps requests and responses between PCI and CBI buses

State Engine Memory

- Up to
  - 32 MBytes of external DDR-SRAM
  - 2.6 Mbytes of internal memory
- Integrated into single address space along with CSRs
- Address space defined by CBI
- Byte addressable
- Divided into four-byte units known as *registers*
  - 135 registers of address space for control functions
  - 1280 registers of address space for physical memory

Policing Scripts

- Programs used by Policing Engine
- Stored in State Engine memory
- Memory holds
  - 1024 standard scripts
  - 256 user-defined scripts

Two Types Of Policing Scripts

- Void script
  - Does not return a value
  - Typical use: accumulate statistics
- Value-returning script
  - Computes and returns a value
  - Typical use: notify FPL of a policing decision

ASI Functions

- Defined by Agere
- Called from FPL program
- Performed by State Engine
- Typical use: access memory
- Name retained from first-generation chip in which State Engine was called *Agere System Interface*
- Functionality offered
  - Memory access
  - Arithmetic operations
  - Logical operations

Example ASI Function

- Name *asiWrite*
- Used to store values in memory
- Two arguments: 24-bit memory address and 32-bit value
- Example:

asiWrite(0x4:24,949:32)

- Stores integer 949 into memory location 4

Memory For Policing Scripts

- Invoked from FPL
- Run by Policing Engine
- Use on-chip data store known as *policing database*

Policing Database

- Implemented as array in memory
- Values persist across multiple packets
- Can be used to accumulate statistics for entire flow
- Up to 512K entries, one entry per flow
- Indexed by integer *flow ID*
- Occupies up to 32 Mbytes of memory

Policing Database
(continued)

- Each entry in database contains 64 bytes
- Policing script decides how to use content of entry
- Examples
  - Count of packets on flow
  - Count of bytes on flow
  - Record size of burst
- Note: to optimize performance, policing database items are cached in *register file*

Policing Scripts

- Written in *C-NP*
- Up to 16 scripts
- Each script given name
- Compiler produces .aso file for script
- Invoked via generic functions

Generic Policing Functions

- Called from FPL
- Flow ID is part of name
- Two forms (*N* denotes flow ID):
  - Function *asiPoliceN* does not return value
  - Function *asiPoliceEOFN* returns a value
- Example call for flow ID 3:

asiPolice3

Performance

- Functions invoked with *asiPolice* (no return value) performed in parallel with FPL
- Programmer can optimize performance by
  - Minimizing *asiPoliceEOF* calls
  - Starting computation with *asiPolice* early

Binding Script Names

- FPL uses generic functions such as *asiPolice0*
- Programmer gives each script a descriptive name
- Therefore, must bind generic function to specific script
- Binding specified as flow ID → script file
- Example of binding ID zero:

```
ID  0  myscript.aso
```

Binding Script Names
(continued)

- Binding
  - Controlled by programmer
  - Specified before chip configured
  - Does not change during execution
- Two possibilities
  - Use SPA to specify
  - Edit XML configuration file

Function Prototypes In FPL

- Used to declare external functions
- FPL statement is *SETUP PROTO*
- Specifies number and size of arguments
- Example:

```
SETUP PROTO (asiPoliceEOF3, 24, 16, 24);
```

- Specifies that function *asiPoliceEOF3* takes three arguments that are 24, 16, and 24 bits long

Example Organization Of Policing Script

```
#include "np5.fpl"
#include "np5asi.fpl"

   // Note: other initialization code goes here

SETUP PROTO(asiPoliceEOF0, 24, 16, 24);

   // Note: classification code goes here

/* Assume flow ID has been placed in variable FID and that      */
/* variable currlength contains the current length of the packet.  */

outcome = asiPoliceEOF0($FID:24, $currlength:16, 0:24);

   // Note: code to place outcome in tm_flags goes here

fTransmit(0:1, 0:1, $DID:20, 0:16, 0:5, $tm_flags:10, $info:24);
```

Policing Database Initialization

- Entry in database must be initialized before APP550 begins
- XML configuration file used
- Values given as pairs:

(flow ID, 64 bytes of parameters for the flow)

State Engine Register File

- Hardware mechanism
- Much faster than memory access
- Used when policing script runs
- Needed because memory too slow
- Essentially a preloaded cache

State Engine Register File

- Before script for flow ID *N* runs
  - State Engine automatically preloads values from policing database entry into register file
- During script execution
  - All memory references refer to values in register file
  - Execution does not wait for (slow) memory
- After script completes
  - State Engine copies values from register file back into policing database entry
- Generic script names provide flow ID to hardware

Example Policing Function

- Single token bucket
- Test whether flow is over sustained rate
- Adds tokens to bucket for elapsed time
- Compares tokens in bucket to packet rate

Policing Script Details

- Hardware *param_block* variable preloaded before script runs
- Result stored in hardware *predicate* register
  - Sixteen bit value
  - Bit fifteen known as *OutOfProfile* bit
- Code written in C-NP

C-NP Language Overview

- Language called *C for Network Processors*
- Formerly known as *Agere Scripting Language*
- We will discuss
  - Lexical conventions
  - Data declarations
  - Expressions
  - Statements
  - Preprocessor directives
  - Script structure

C-NP Lexical Conventions

- Comments
  - C++ or C style comments
- Four forms of numeric constants

Type	Syntax
Binary	Starts with 0b followed by binary digits
Decimal	Digits not starting with zero
Hexadecimal	Starts with 0x followed by hex digits
Octal	Starts with 0 followed by digits 0 through 7

C-NP Lexical Conventions

- Reserved words

  block
  boolean
  else
  false
  if
  inout
  input
  littleEndian
  output
  script
  signed
  swap
  true
  unsigned

- Register file references

Syntactic Form	Meaning
@[*X*] *type*	Reference the X^{th} byte (*X* is between 0 and 127)
@[*X* : *Y*] *type*	Reference a string of bytes from byte *X* to byte *Y*

C-NP Lexical Conventions

- Predicate word reference

$[*X* **]** *type*

- Indirect reference
  - Permits reference of bytes within a variable
  - Example

```
block          entire_packet  @[0:63];
block          ip_header  packet_data[14:33];
unsigned srcIPip_header[12:15];
unsigned destIPip_header[16:19];
```

- Note: default is big endian byte order; programmer can override

C-NP Expressions

- Integers are *signed* or *unsigned*
- Arithmetic Operators are + - * /   unary -
- Type casts are permitted
- Logical operators are < > >= <= == !=
- Bitwise shift operators are << >>
- Bitwise Logical Operators are & | ^ ~

Statements

- Only four statements types

Statement	General Form
assignment	*identifier* **=** *expression*

conditional with optional else	**if (** *expression* **)** *statement*
	else *statement*

selection	**switch (** *expression* **)** *case statements*
compound	*statements separated by semicolons*

- Note: no iteration such as *for* or *while*

Preprocessor Directives

#define
#elif
#else
#endif
#error
#if
#ifdef
#ifndef
#include
#pragma
#undef

- Only one #pragma directive: *multiplySupport* for multiplication

Standard Header Files

- Define constants and variables

File	Pertinent Engine
policeNp5.h	Policing Engine
tmNp5.h	Traffic Manager
tsNp5.h	Traffic Scheduler
sedNp5.h	Stream Editor

Example Header File Contents

- File *policeNp5.h* contains declaration

```
unsigned  current_time  @[8:11]  input;
```

Alignment And Timing

- Aligning values in register file speeds access
- Think of register file as 2-dimensional array
  - Row corresponds to four-byte *register*
  - Column called a *slice*
- Example bytes 0, 4, 8,... lie in first slice
- Access optimized when both operands lie in the same slice

Script Structure

optional preprocessor directives

data declarations

script *script name*

script body

Example Policing Script

- Compare arriving traffic to token bucket
- Code does not explicitly return a value: uses *predicate* instead
- Script can set 15 of 16 bits in the predicate
- Example code used bit 15 for *OutOfProfile*

Example Policing Script (part 1)

```
#include "policeNp5.h"

/* Token Bucket parameters in param_block are        */
/* initialized at configuration                      */

/* Bit Rate (initialized to RTC ticks per byte) */
unsigned BR param_block[0:1] input;

/* Burst Size (initialized to Bytes times BR) */
unsigned BS param_block[2:9] input;

/* Previous arrival time (initialized to zero) */
unsigned last_pdu_arrival param_block[10:13] inout;

/* Token counter (initialized to BS) */
unsigned tokens param_block[14:21] inout;
```

Example Policing Script (part 2)

```
/* Packet length (passed as argument from FPL) */
unsigned pdu_length fpp_args[0:1] input;

unsigned(4) delta_t;               /* RTC ticks since last packet arrived */

unsigned(4) pdu_len_t;             /* Packet size in RTC ticks */

boolean OutOfProfile $[15] output;  /* results of policing */

script tokbucket {                 /* will be bound to asiPoliceEOF0 */
  delta_t = current_time - last_pdu_arrival;
  pdu_len_t = pdu_length * BR;
  tokens = tokens + delta_t;       /* update bucket length */
  if (tokens > BS)  {
      tokens = BS;
  }
```

Example Policing Script (part 3)

```
  OutOfProfile = tokens < pdu_len_t;  /* compute result */

  if (!OutOfProfile)  {
      tokens = tokens - pdu_len_t;           /* update bucket depth */
  }

  last_pdu_arrival = current_time;           /* update timestamp */
}
```

Summary

- State Engine provides
  - Memory
  - Host interface
  - Support for policing computation
- Memory accessed via ASI functions
- Host interface
  - Uses PCI bus
  - Supports fetch-store paradigm
  - Defines CSRs

Summary
(continued)

- Traffic policing
  - Performed by Policing Engine
  - Programmable via C-NP scripts
  - Decides whether flow is within performance profile
  - Result returned to classifier
- C-NP
  - Scripting language derived from C
  - Limited expressions
  - No iteration

Questions?

XXII

Traffic Manager (TM)

Traffic Management

- Generic term
- Usually includes
  - Bandwidth allocation
  - Enforcement of priorities
- May also include
  - Traffic policing
  - Buffering, queueing, and memory management

Agere Traffic Manager Block

- Performs five main functions
  - Buffer management
  - Completion of flow policing and packet discard
  - Traffic shaping
  - Bandwidth allocation
  - Packet modification

Buffer Management

- Needed for store-and-forward paradigm
- Critical decision: what happens when memory exhausted?
  - Tail drop: wait until memory full and then discard
  - Random Early Detection (RED): probabilistically discard as memory fills

Tail Drop

- Trivial to implement
- Leads to global synchronization of TCP streams

Random Early Detection (RED)

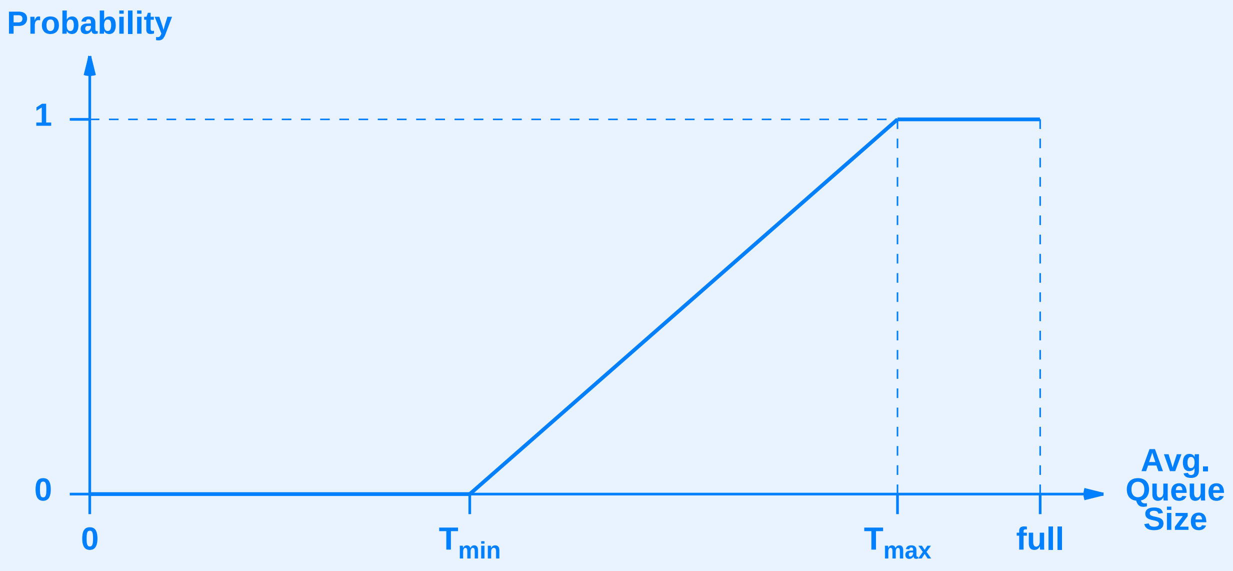
- Alternative to tail drop
- More difficult to implement
- Defined for floating point calculation

RED Algorithm

- Define two thresholds
  - T_{min} at which RED starts
  - T_{max} beyond which all packets are dropped
- Vary probability of drop linearly when average queue size, Q_{avg}, lies between the thresholds:

$$P = \frac{Q_{avg} - T_{min}}{T_{max} - T_{min}}$$

Illustration Of RED Probability



- Probability varies linearly over the range from T_{min} to T_{max}

RED Queue Size Computation

- Uses smoothed average
  - Avoids quick response to packet bursts
  - Weights exact queue size and long-term average
- Computation of exact size

$$Q = Q + N$$

- Computation of long term average

$$Q_{avg} = \alpha\ Q + (1 - \alpha)\ Q_{avg}$$

- Note: α is a fraction ($0 \leq \alpha \leq 1$) that weights the new queue size (typical value is 0.2)

Completion Of Flow Policing

- Performed by *Policing Engine*
- Determines whether flow is within *profile*
- Marks each packet for later processing

Example Flow Policing

- VBR-style profile
- Four parameters
  - Sustained Bit Rate (SBR)
  - Peak Bit Rate (SBR)
  - Sustained Burst Size (SBS)
  - Peak Burst Size (PBS)
- Implemented with dual token bucket

Dual Token Bucket

- First bucket monitors Peak Bit Rate (PBR)
- Second bucket monitors Sustained Bit Rate (SBR)
- Result is *tri-color labeling*:
  - Green: flow is less than the SBR
  - Yellow: flow is less then the PBR but exceeds the SBR
  - Red: flow exceeds the PBR

Discard Decision

- Also called *drop decision*
- Important concept: single decision handles both buffering and flow policing
- Input parameters
  - Current average queue size
  - Tri-color label
- Algorithm is *Weighted RED* (WRED)

Weighted RED

- Probabilistic packet discard
- Same basic algorithm as RED
- Three independent sets of parameters
  - T_{min} and T_{max} for green packest
  - T_{min} and T_{max} for yellow packets
  - T_{min} and T_{max} for red packets

The Point About Discard

A single algorithm, known as Weighted RED, *handles both buffer management and flow policing.* WRED *requires the policer to label each packet with one of three colors, and uses the color to compute a probability of discard.*

WRED On The APP550

- Cannot use floating point
  - Too slow
  - No floating point hardware on the APP550
- Solution
  - Use integer arithmetic
  - Make α an inverse power of two
  - Replace multiplication and division with bit shifting
  - Replace linear probability with set of intervals

Approximation

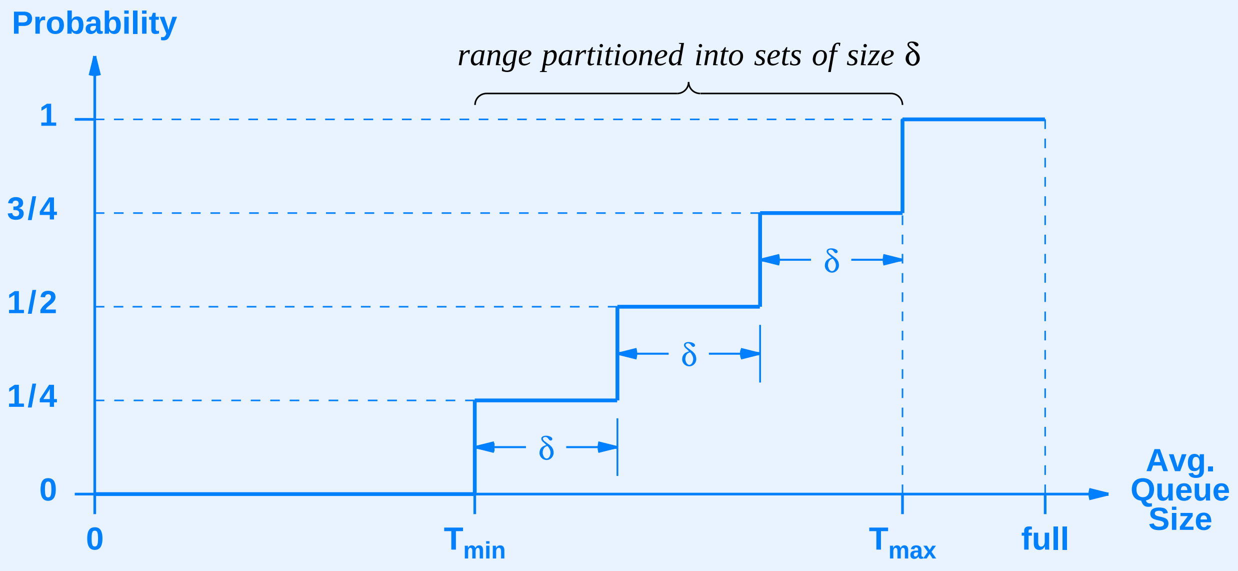
- Divide the interval

  T_{min} to T_{max}

  into a set of equal-size groups

- Assign each group fixed value of probability

Illustration Of Using Intervals



- Value of δ can be chosen at compile time
- Eliminates multiplication and division

RED Without Multiplication Or Division

```
if  (packet color is red)  {
        tmin = red_tmin
        tmax = red_tmax
        delta = red_delta
}  else if  (packet color is yellow)  {
        tmin = yellow_tmin
        tmax = yellow_tmax
        delta = yellow_delta
}  else  {
        tmin = green_tmin
        tmax = green_tmax
        delta = green_delta
}
invoke  RED(tmin, tmax, delta)
```

- Color constants specified at compile time

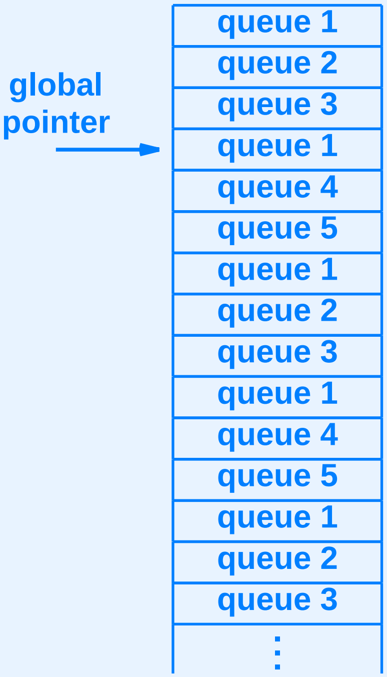
Scheduling And Traffic Shaping

- Two problems
  - Allocation of bandwidth among flows
  - Shape traffic on each flow
- Single hierarchical traffic scheduler solves both problems
- Lowest level of hierarchy
  - Programmable via C-NP script
  - Known as a *scheduler*

Scheduler Mechanism

- Circular time slot table plus global pointer
- On each time slot, global pointer moves and entry is used
- Each entry points to a queue
- Specific queue can be repeated in table
  - Used for queue priorities
  - Repetition gives queue higher frequency of selection

Illustration Of Time Slot Table



global pointer

queue 1
queue 2
queue 3
queue 1
queue 4
queue 5
queue 1
queue 2
queue 3
queue 1
queue 4
queue 5
queue 1
queue 2
queue 3
⋮

- Pointer moves at a fixed rate
- Entries in table specified by programmer

CBR Shaping

- Straightforward with time slot table
- Given queue appears at regular interval throughout the table
- Example
  - Time slot table contains six entries
  - Output interface rate is OC-12
  - Each table slot corresponds to OC-2 rate
  - A CBR queue, call it X, needs OC-4 rate
  - Queue X appears twice in table

A Note About Bandwidth Allocation

In a packet switching system, the allocation of bandwidth is closely interrelated with traffic shaping because both are achieved by selecting packets from multiple sources.

Bandwidth Allocation

- Two forms
  - Fixed allocation
  - Proportional allocation
- Can apply at multiple levels
- Given network system may need both

Fixed Allocation

- Specified amount of bandwidth allocated to each source
- Form of Time Division Multiplexing
- Often used to subdivide physical channel
- Actual bandwidth cannot exceed allocation
- If source is idle, its allocation goes unused
- Consequence: output may be idle even though packets are waiting

Fixed Allocation
(continued)

- Example: can be used to divide an OC-48 line into:
  - Four channels that operate at OC-12
  - Two channels that operate at OC-12 and eight channels that operate at OC-3
  - Other combinations that sum to OC-48

The Point Of Fixed Bandwidth Allocation

Fixed bandwidth allocation is used to partition bandwidth into isolated channels. The isolation guarantees that a channel cannot encroach on the bandwidth allocated to another channel, but also means that bandwidth can remain unused if a channel is idle.

Implementation Of Fixed Allocation

- For cells
  - Especially easy
  - Similar to CBR scheme
- For variable-size frames
  - Not as simple, but still straightforward
  - Instead of counting packets, keep track of size of packets sent

Proportional Allocation

- Each source is assigned percentage of total capacity
- If a source does not have traffic to send, its bandwidth is allocated to remaining sources proportional to their percentages
- Often used to partition capacity among flows
- Consequences
  - Output not idle provided at least one source has packets to send
  - Given source can exceed its percentage

Proportional allocation
(continued)

- When all channels busy, resembles fixed allocation
- When only one channel busy, channel receives 100% of the bandwidth

The Point Of Proportional Allocation

Proportional bandwidth allocation is used to provide controlled sharing among a set of channels. When all channels compete, bandwidth is divided as in a fixed allocation scheme; when some channels use less than the amount allocated to them, the excess bandwidth is divided among the remaining channels proportional to their overall share.

Example Of Proportional Allocation

Channel	Proportional Share	Effective Bandwidth
1	25.0%	OC-6
2	12.5%	OC-3
3	50.0%	OC-12
4	12.5%	OC-3

- If channel 3 becomes idle
  - Channel 1 receives equivalent of OC-12
  - Channel 2 receives equivalent of OC-6
  - Channel 4 receives equivalent of OC-6

Analysis Of Proportional Allocation

- Let p_i denote percentage of bandwidth devoted to channel i, $(1 \leq i \leq N)$
- Total allocation sums to 100%, so

$$\sum_{i=1}^{N} p_i \;=\; 100$$

- If channel k is idle, channel i receives effective bandwidth of

$$e_i \;=\; \frac{p_i}{100 - p_k}$$

Analysis Of Proportional Allocation (continued)

- Let a_i represent actual use of Channel *i* measured as percentage of total bandwidth
- Percentage of unused bandwidth is:

$$A = \sum_{i=1}^{N} (p_i - a_i)$$

- Effective bandwidth allocated to Channel *i* is:

$$e_i = \frac{p_i}{100 - A}$$

Implementation Of Proportional Allocation

- Theoretically optimum algorithm is *Generalized Processor Sharing* (*GPS*)
- Too expensive in practice
  - Cannot use floating point
  - Cannot use integer multiplication or division

Algorithms For Proportional Allocation

- Weighted Fair Queueing (WFQ)
  - Too inefficient
  - Does not scale well for large numbers of queues
- Weighted Round Robin (WRR)
  - Efficient
  - Handles variable-size frames
  - Scales
  - Close to optimal performance

Algorithms For Proportional Allocation
(continued)

- Smoothed Deficit Weighted Round Robin (SDWRR)
  - Variant of WRR
  - Supported by Agere hardware
- Generalized Processor Sharing (GPS)
  - Theoretical optimal
  - Impractical for packet switching systems
  - Used to assess other algorithms

SDWRR

- Uses four FIFO lists (0 through 3)
- Each FIFO list contains a set of packet queues
- Hardware services FIFOs round-robin
- Assigns three limits to each queue
- Uses limits to determine whether queue should move to new FIFO list
- Computes deficit between amount of data sent and amount that should have been sent

SDWRR
(continued)

- If deficit exceeds limit 1, move queue forward one FIFO list
- If deficit exceeds limit 2, move queue forward two FIFO lists
- If deficit exceeds limit 2, move queue forward three FIFO lists
- Moving essentially postpones service because the queue has already exceeded bandwidth

APP550 Traffic Management Mechanisms

- Five queueing mechanisms
- Arranged in a hierarchy
  - Port Manager
  - Logical Port
  - Scheduler
  - QoS Queue
  - CoS Queue

Top Level Port Managers

- Correspond to physical output ports
- Multiple managers can be assigned to given output port
- Configurable, but not programmable
- Configured to provide fixed allocation
- Port manager configurable but not programmable

Second Level Logical Ports

- Sources of data for a port manager
- One or more Logical Ports assigned to Port Manager
- Configurable, but not programmable

Third Level Schedulers

- Programmable scheduler
  - Spans three lower levels of hierarchy
  - Selected at the third level
  - Makes decisions about fourth and fifth levels
  - Multiple schedulers (up to four) can be assigned to Logical Port

Fourth Level QoS Queues

- Up to sixty-four thousand QoS queues per scheduler
- Handle per-flow Quality Of Service
- Scheduler selects among queues
- Typical scheduler algorithm: *Smoothed Deficit Weighted Round Robin* (*SDWRR*)
- Can have proportional bandwidth scheduling
- Can be lowest level

Fifth Level CoS Queues

- Optional
- Handles *Class of Service* withing a QoS queue
- Up to sixteen CoS queues per QoS queue
- Default: priority of CoS queue *i* higher than priority of CoS queue *i+1*

Summary Of Traffic Manager Hierarchy

Level	Mechanism	Number
1	Port Manager	256 total
2	Logical Port	1024 total
3	Scheduler	4 per Logical Port
4	QoS Queue	64K per Scheduler
5	CoS Queue	16 per QoS Queue

Summary

- APP550 Traffic Manager performs
  - Buffer management
  - Completion of flow policing
  - Packet discard
  - Traffic shaping
  - Bandwidth allocation
  - Packet modification
- Weighted RED used for buffer management

Summary
(continued)

- Bandwidth allocation can be
  - Fixed
  - Proportional
- Traffic manager has five-level scheduling hierarchy
  - Port Manager
  - Local Ports
  - Schedulers
  - QoS Queues
  - Cos Queues

Questions?

XXIII

Host Interface And Control Functions

Motivation For An External Processor

- No general-purpose processor on the APP550
- APP550 hardware is highly specialized
- Insufficient computational power on APP550 for other tasks
- Conclusion: external host needed

Role Of An External Host Processor

- Initial configuration of an APP550
- Dynamic update of data structures
- Retrieval and update of status information and statistics
- Slow-path packet processing

Dynamic Update Of Data Structures

- Host can alter
  - Patterns used in classification
  - The set of Destination IDs (DIDs)
  - Logical ports used by the Traffic Manager

Retrieval And Update Of Status Information And Statistics

- Performed in conjunction with State Engine
- Allows external host to monitor or reset statistics while APP550 runs

Physical Interconnection To An External Host

- Peripheral Component Interconnect bus (PCI bus)
- APP550 defines
  - Set of hardware registers
  - Bus address for each
- State Engine block provides bus interface

Packet Exchange And The Concept Of Pseudo Interface

- Hardware defines *pseudo interface*
  - Appears to be packet interface
  - Allows packet traffic to pass over bus to external host
- Known as the *Management Path Interface*

Application Program Interface (API) For External Hosts

- Defines communication between APP550 and host
- Consists of functions that host uses to
  - Interrogate APP550
  - Control APP550

Two Levels Of API

- Device level
  - Implement low-level communication between the host and APP550
  - Messages sent over PCI bus
- Object level
  - Higher-level interface functions
  - Invoke device-level functions
  - Example: allow host to change data structure for a Destination ID

Programming Paradigm And Handles

- Used with object-level functions
- Host program
  - Create a *handle* for specific object
  - Use handle in series of calls to interrogate or modify the object
- Example
  - Create handle for complex data object
  - Call functions that build the object
  - Pass handle to function that uses object

Items For Which A Handle Can Be Defined

Chipset Originally used to refer to the entire set of three chips, the name is now used for functions that span one or more APP550s.

APP550 Used for functions that manipulate the APP550 hardware configuration.

asi Used for functions that refer to the State Engine, which was known as the ASI in the previous generation.

fpp Used for functions that refer to the Classification block and the Pattern Processing Engine, which was known as the FPP in the previous generation.

rsp Used for functions that refer to the Traffic Manager block, which was known as the RSP in the previous generation.

Example Declarations Used For Handles

```
ag_chipset_t    chipset_handle;
ag_np5_t        np_handle;
ag_fpp_t        fpp_handle;
ag_rsp_t        rsp_handle;
ag_asi_t        asi_handle;
```

Initialization Functions

- Used to initialize data structure associated with a handle
- Set of functions provided by Agere
- Invoked in top-down order to initialize
  - Chipset
  - APP550
  - Fpp
  - Rsp
  - Asi

Initialization Functions (Part 1)

ag_chipset_init

Purpose: initialize a data structure that ties together all secondary data structures.
Notable argument: &chipset_handle

ag_chipset_config

Purpose: extract hardware configuration parameters from an image generated by Agere's SPA or a command line tool.
Notable argument: chipset_handle

ag_chipset_hdl_get

Purpose: initialize the data structures used with the APP550, and link them into the chipset.
Notable arguments: chipset_handle, &np_handle

Initialization Functions (Part 2)

ag_fpp_hdl_get

Purpose: initialize data structures associated with the Classification block, and link them into data structures for the chip.

Notable arguments: np_handle, &fpp_handle

ag_rsp_hdl_get

Purpose: initialize data structures associated with the Traffic Manager block, and link them into data structures for the chip.

Notable arguments: np_handle, &rsp_handle

ag_asi_hdl_get

Purpose: initialize data structures associated with the State Engine block, and link them into data structures for the chip.

Notable arguments: np_handle, &asi_handle

Initialization Functions

- Note: functions initialize items on the host; communication with APP550 deferred until later

Examples Of Object Functions

- API offers many functions host software can invoke to
  - Interrogate values
  - Modify parameters to control behavior of the APP550
- A few examples include (not a comprehensive list):

Function	Purpose
ag_fpp_learn	Learn a pattern (add to a tree function)
ag_fpp_list_ptns	List all patterns in a tree function
ag_fpp_unlearn	Unlearn a pattern (delete from a tree function)
ag_rsp_did_add	Add a new DID to the set
ag_rsp_did_get	Obtain information about a DID
ag_rsp_queue_add	Add a queue to the Traffic Manager

External Host Capability

- In addition to administrative tasks, external host can change data structures such as
  - IP routing table (forwarding)
  - Firewall rules (filtering)
  - Classification (queueing)
- Dynamic classification significantly more powerful than static classification
  - Permits flow-based classification
  - Per-flow scheduling

A Dynamic Classification Example

- Assume FPL program contains a tree function named *Network*

```
Network: 192.168.0.* fReturn(0);
Network: 192.168.1.* fReturn(1);
```

- FPL assigns tree function a unique internal identifier
- External host uses identifier to update tree function

Tree Identifiers

- External host cannot determine tree function identifier at run-time
- Instead, programmer can use *SETUP ASSIGN* statement to specify explicit identifier
- Assignable range is 3072 through 4095
- Example: to assign 3073 as ID for function *Network*

```
SETUP ASSIGN(Network, TREE, 3073)
```

Tree Identifiers
(continued)

- To ensure classification code and external host application code uses same constant for tree identifier, declare constant in header file and include in both programs
- Note: FPL and C use same syntax for symbolic constant declaration
- Example: place the following in file *example.h*

```
#define NETWORK_TREE_ID      3073
```

Tree Identifiers
(continued)

- To use constant from file *example.h* in an FPL program

```
#include "example.h"

SETUP ASSIGN(Network, TREE, NETWORK_TREE_ID);

Network: 192.168.0.* fReturn(0);
Network: 192.168.1.* fReturn(1);
```

Example Of Dynamic Tree Update

- To add the following to a dynamic tree function

Network: 192.168.2.* fReturn (2);

- Steps are
  - Initialize handles
  - Invoke function *ag_fpp_learn*
  - Use arguments to specify item to be added

Code To Insert Item Into A Tree Function

```
/* Initialization of handles goes here */

#include "example.h"

unsigned int        netaddr = 0xC0A80200;     /* 192.168.2.0 */
ag_fpp_ptn_t        fppPattern;
ag_fpp_action_t     fppAction;

fppPattern.data = &netaddr;                   /* pointer to data */
fppPattern.noDataBits = 24;                   /* 24 significant bits */
fppPattern.noWildCardBits = 8;                /* last 8 bits - wildcard */

fppAction.type = ag_fpp_action_type_return;   /* fReturn() action */
fppAction.value = 2;                          /* fReturn() value */

ag_fpp_learn(fpp_handle, NETWORK_TREE_ID,
        &fppPattern, &fppAction);
```

Constants And Host Byte Order

- Constant 0xC0A80200 depends on host byte order
- Example code assumes host is big-endian
- Library functions do not perform conversions

Example Of Slow Path Packet Transfer (Part 1)

```
/* Example code that runs on an external host and obtains          */
/* packets from an APP550.  The external host is only used for     */
/* the slow path.          */

#define BUF_SIZE 2048
#include <agere_np5.h>

int main(int argc, char *argv[])  {

    ag_st_t              rc;
    ag_np5_dev_hdl_t     devHandle;
    unsigned char        pdu_buf[BUF_SIZE];
    ag_uint32_t          pdu_buf_size = BUF_SIZE, pdu_size, devNum;

    if (argc < 2)  {
        fprintf(stderr,"\nUsage: %s <device number>\n",argv[0]);
        return(-1);
    }
```

Example Of Slow Path Packet Transfer (Part 2)

```
/* get device number from command line */
devNum = atoi(argv[1]);

/* Open NP5 device */
rc=ag_np5_dev_open(devNum, 0, &devHandle);
if (rc != AG_ST_SUCCESS)  {
    fprintf(stderr,"\nError: Cannot open device number %i.\n",devNum);
    return(-1);
}
/* read packets sent from the APP550 */

while(1)  {  /* do forever */

    /* read packet from ASI receive queue (block if queue is empty) */

    rc = ag_np5_dev_pdu_read(devHandle, pdu_buf, pdu_buf_size,
             &pdu_size);

    /* use return code to determine processing */
```

Example Of Slow Path Packet Transfer (Part 3)

```
switch (rc)  {

case AG_ST_DEV_INVALID_HANDLE:
    fprintf(stderr,"\nError: Invalid device handle! Exiting.\n");
    return(-1);

case AG_ST_DEV_INVALID_BUFFER:
    fprintf(stderr,"\nError: Invalid PDU buffer! Exiting.\n");
    return(-1);

case AG_ST_SUCCESS:

    /* At this point, a packet has been loaded into pdu_buf.      */
    /* Additional code should be inserted here to handle the      */
    /* packet.         */

    break;
```

Example Of Slow Path Packet Transfer (Part 4)

```
        default:
            fprintf(stderr,"\nUnknown return code: %u. Exiting.\n",rc);
            return(-1);

        }   /* end switch */

    }   /* end while */

}   /* end main program */
```

Summary

- External host is required to
  - Initialize chip
  - Update dynamic data structures
  - Provide slow-path packet processing
- Agere provides API that software on external host uses to communicate with APP550
  - Device-level functions handle bus interface
  - Object-level functions permit host to control and manage FPL data structures

Summary
(continued)

- Paradigm: host software
  - Initializes a handle
  - Makes a set of function calls to build and modify data structure
  - Calls functions to download the resulting data structure onto the APP550

XXIV

An Example System

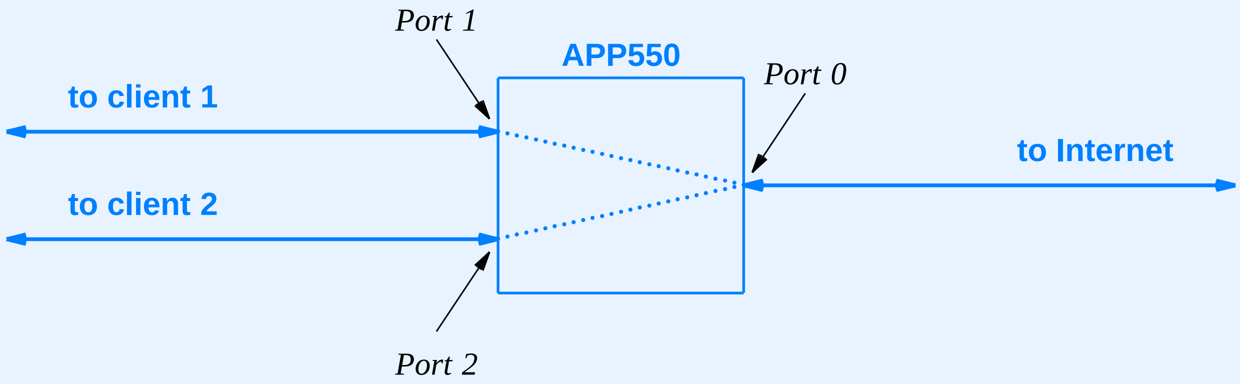
ISP Access Node

- Example system
  - Operates between ISP and clients
  - Uses policing and scheduling
  - Monitors traffic
  - Enforces *Service Level Agreement* (*SLA*) between customer and ISP

Example Access System Functions

- Classification of packets arriving from client, according to SLA
- Policing of client traffic to ensure that the traffic follows the SLA
- Scheduling of traffic in both directions according to the SLA

Illustration Of Example System



Basic Functionality

- Example access system implements *Differentiated Services* (*DiffServ*)
- Traffic divided into five *classes*
  - Four classes are *assured forwarding* (*AF*) for normal traffic
  - One class is *expedited forwarding* (*EF*) for network management traffic
- A *dropping precedence* is appended to class values

Binary Encoding Of DiffServ Values

- Known as *codepoint*

Encoding	Name	Encoding	Name
001 010	AF11	011 010	AF31
001 100	AF12	011 100	AF32
001 110	AF13	011 110	AF33
010 010	AF21	100 010	AF41
010 100	AF22	100 100	AF42
010 110	AF23	100 110	AF43
		101 110	EF

Mapping SLA Requirements To DiffServ Classes

- Assume four flows given priority

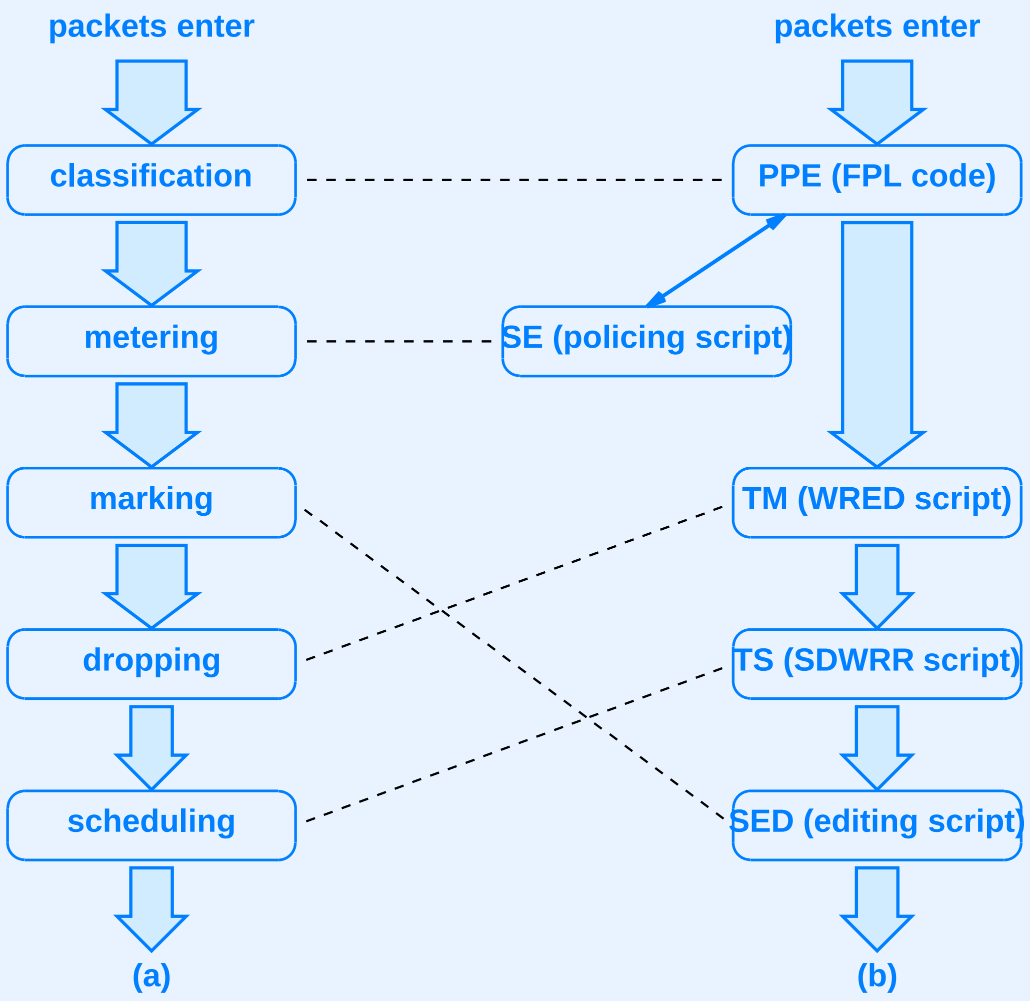
Client	IP addr.	Flow	Profile	Class
1	10.*.*.*	Video	100 mbps	AF4
1	10.*.*.*	Audio	10 mbps	AF3
2	128.211.*.*	Audio	5 mbps	AF3
2	128.10.*.*	Audio	2 mbps	AF3

- Note: all other flows assigned to DiffServ class AF1.

Conceptual DiffServ Pipeline

- Classification
- Metering
- Marking
- Dropping
- Scheduling

Correspondence Between Model And Implementation



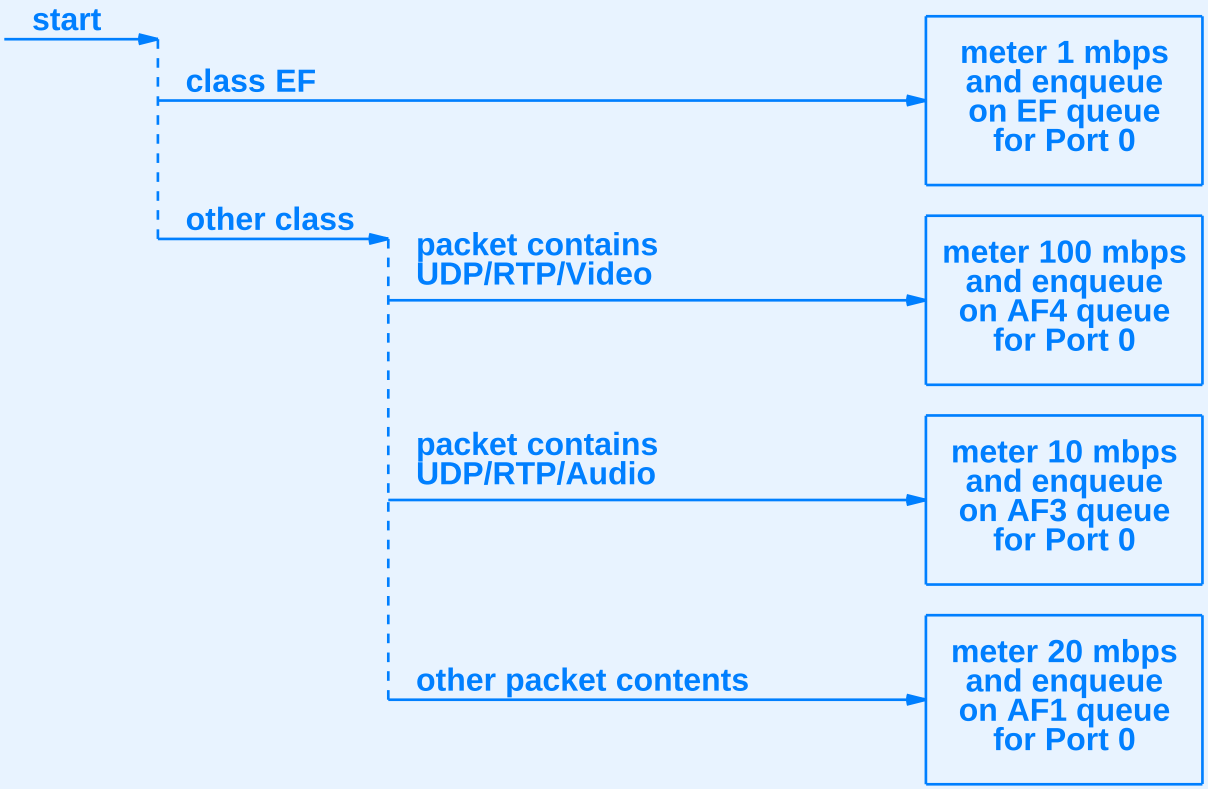
Queues And Destination IDs

DID	Port	Class
1	0	EF
2	0	AF1
3	0	AF2
4	0	AF3
5	0	AF4
6	1	EF
7	1	AF1
8	1	AF2
9	1	AF3
10	1	AF4
11	2	EF
12	2	AF1
13	2	AF2
14	2	AF3
15	2	AF4

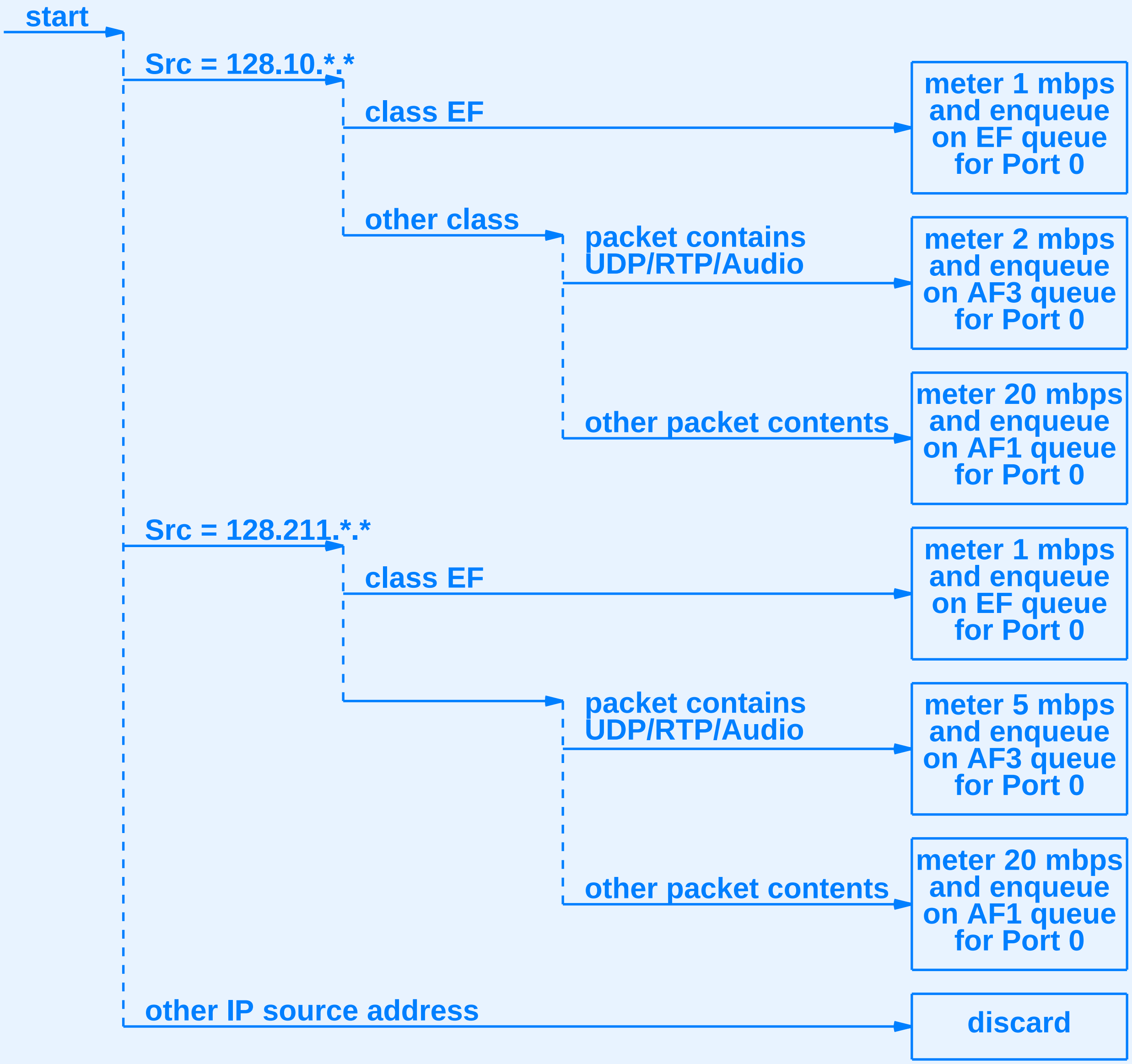
Classification Algorithm

```
if (codepoint specified expedited forwarding) {
   Run the policing script for EF packets;
   Enqueue packet on the EF queue for Port 0
} else if (packet carries UDP/RTP/video) {
   Run the policing script for 100 Mbps video;
   Enqueue packet on the AF4 queue for Port 0
} else if (packet carries UDP/RTP/audio) {
   Run the policing script for 10 Mbps audio;
   Enqueue packet on the AF3 queue for Port 0
} else {
   Run the policing script for best-effort traffic
   Enqueue packet on the AF1 queue for Port 0.
}
```

Decision Tree For Packet From Port 1


start
class EF
meter 1 mbps and enqueue on EF queue for Port 0
other class
packet contains UDP/RTP/Video
meter 100 mbps and enqueue on AF4 queue for Port 0
packet contains UDP/RTP/Audio
meter 10 mbps and enqueue on AF3 queue for Port 0
other packet contents
meter 20 mbps and enqueue on AF1 queue for Port 0

Decision Tree For Packet From Port 2



start

Src = 128.10.*.*

class EF

meter 1 mbps and enqueue on EF queue for Port 0

other class

packet contains UDP/RTP/Audio

meter 2 mbps and enqueue on AF3 queue for Port 0

other packet contents

meter 20 mbps and enqueue on AF1 queue for Port 0

Src = 128.211.*.*

class EF

meter 1 mbps and enqueue on EF queue for Port 0

packet contains UDP/RTP/Audio

meter 5 mbps and enqueue on AF3 queue for Port 0

other packet contents

meter 20 mbps and enqueue on AF1 queue for Port 0

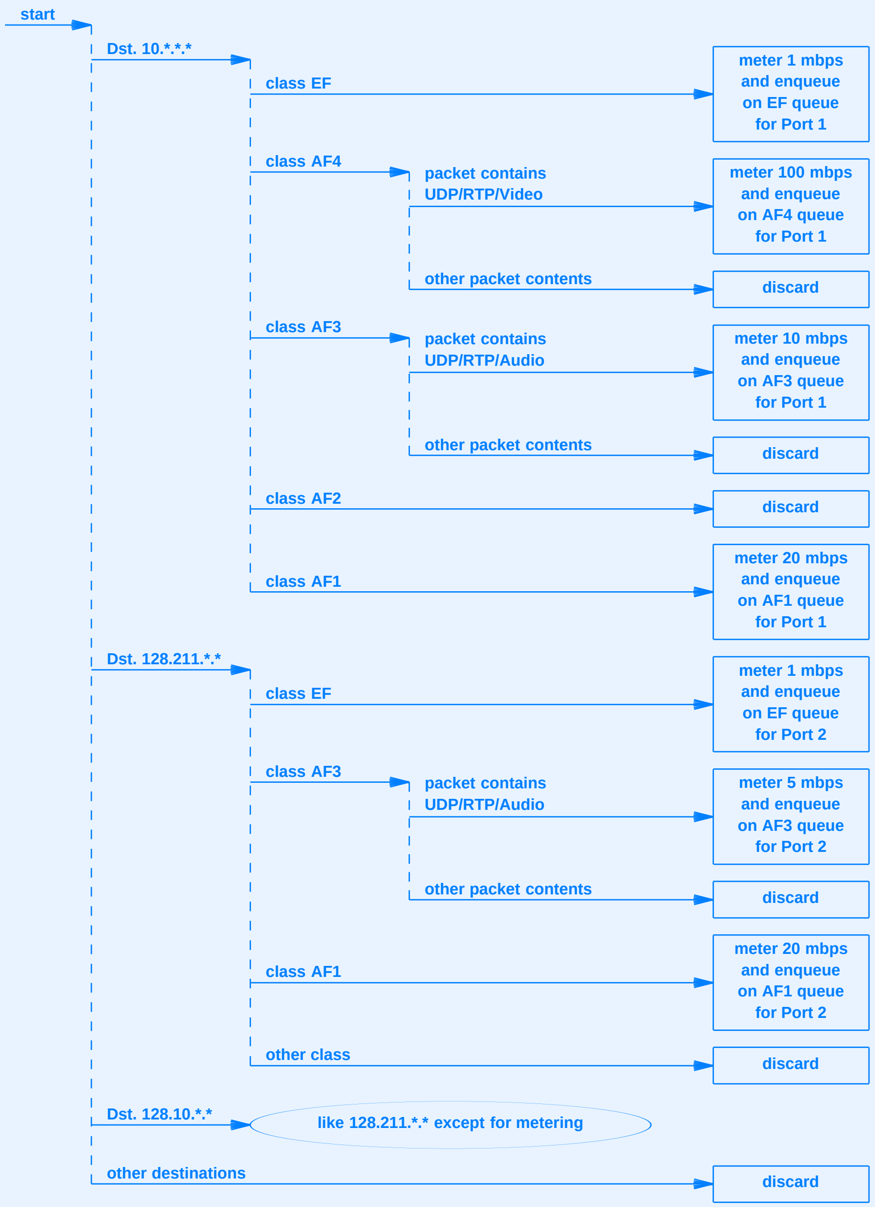
other IP source address

discard

Examples Of Decisions

- Client 2 sends a packet with an arbitrary source address
  - Packet will be discarded
- Client 1 sends a packet with an arbitrary source address
  - Packet will be forwarded to Port 0

Decision Tree For Packet From Port 0



start

Dst. 10.*.*.*

class EF

meter 1 mbps and enqueue on EF queue for Port 1

class AF4

packet contains UDP/RTP/Video

meter 100 mbps and enqueue on AF4 queue for Port 1

other packet contents

discard

class AF3

packet contains UDP/RTP/Audio

meter 10 mbps and enqueue on AF3 queue for Port 1

other packet contents

discard

class AF2

discard

class AF1

meter 20 mbps and enqueue on AF1 queue for Port 1

Dst. 128.211.*.*

class EF

meter 1 mbps and enqueue on EF queue for Port 2

class AF3

packet contains UDP/RTP/Audio

meter 5 mbps and enqueue on AF3 queue for Port 2

other packet contents

discard

class AF1

meter 20 mbps and enqueue on AF1 queue for Port 2

other class

discard

Dst. 128.10.*.*

like 128.211.*.* except for metering

other destinations

discard

Policing, Coloring, And Flow IDs

- Example code uses *dual token bucket*
- Traffic measured against
  - Sustained rate bucket
  - Peak rate bucket
- Results is *color* assigned to packet
  - Red: exceeds both rates
  - Yellow: only exceeds sustained rate
  - Green: does not exceed either

Policing, Coloring, And Flow IDs

- Note: if packet already has DiffServ classification, can use classification to precolor packet
  - Drop precedence 1 green
  - Drop precedence 2 yellow
  - Drop precedence 3 red
- Policing can move up (green toward red), but never down

Flow IDs Used In The Example Code

- For traffic forwarded to clients 1 or 2

Flow ID	Destination IP Address	Profile	Class
1	10.*.*.*	None	EF
2	128.10.*.*	None	EF
2	128.211.*.*	None	EF
3	10.*.*.*	None	AF1
4	128.211.*.*	None	AF1
4	128.10.*.*	None	AF1
5	10.*.*.*	10 mbps	AF3
6	128.211.*.*	5 mbps	AF3
7	128.10.*.*	2 mbps	AF3
8	10.*.*.*	100 mbps	AF4

Flow IDs Used In The Example Code

- For traffic received from clients 1 or 2

Flow ID	Source IP Address	Profile	Class
9	10.*.*.*	None	EF
9	128.211.*.*	None	EF
9	128.10.*.*	None	EF
10	10.*.*.*	None	AF1
10	128.211.*.*	None	AF1
10	128.10.*.*	None	AF1
11	10.*.*.*	10 mbps	AF3
12	128.211.*.*	5 mbps	AF3
13	128.10.*.*	2 mbps	AF3
14	10.*.*.*	100 mbps	AF4

- Entry *None* corresponds to cases where SLA does not specify a value

Example Classification Code (part 1)

```
/* ds_classifier.fpl - classification for the example DiffServ node     */

/*
 * ---------------------------------------------------------------------
 * DiffServ classifier for a boundary node.
 * Supported classes are: EF, AF1x, AF2x, AF3x, AF4x
 * Policing is performed for packets from clients; packets to clients are
 * already policed and marked with a DiffServ codepoint
 * All frames are Ethernet.
 * ---------------------------------------------------------------------
 */

#include "np5.fpl"
#include "np5asi.fpl"

/* Setup error handler */
SETUP ERROR(MyError);

/* ---------------------------------------------------------------------
 * Input Ports
 */
#define IN_PORT 0 /* traffic from the DS network arrives over Port 0 */
#define C1_PORT 1 /* Traffic from client 1 arrives over Port 1       */
#define C2_PORT 2 /* Traffic from client 2 arrives over Port 2       */
```

Example Classification Code (part 2)

```
/* ---------------------------------------------------------------------
 * Protocol constants
 */
#define IPT_UDP 17
#define RTP_PORT 5004

/* ---------------------------------------------------------------------
 * Data types
 */
#define UNK   0  /* Unknown */
#define AUD   1  /* Audio   */
#define VID   2  /* Video   */

/* ---------------------------------------------------------------------
 * DiffServ classes
 */
#define EF  0
#define AF1 1
#define AF2 2
#define AF3 3
#define AF4 4

/* ---------------------------------------------------------------------
 * Clients
 */
#define CL1    0 /* client 1       */
#define CL2IP1 1 /* client 2, IP 1 */
#define CL2IP2 2 /* client 2, IP 2 */
```

Example Classification Code (part 3)

```
/* ----------------------------------------------------------------------
 * Client networks
 */
#define IP_C1  10.*.*.*
#define IP1_C2 128.211.*.*
#define IP2_C2 128.10.*.*

/* ----------------------------------------------------------------------
 * Directions
 */
#define IN  0
#define OUT 1

/* ----------------------------------------------------------------------
 * Flow IDs
 * (flow ID 0 is dummy flow)
 */
#define C1_IN_EF_FID      1
#define C2IP1_IN_EF_FID   2
#define C2IP2_IN_EF_FID   2

#define C1_IN_AF1_FID     3
#define C2IP1_IN_AF1_FID  4
#define C2IP2_IN_AF1_FID  4

#define C1_IN_AF2_FID     0
#define C2IP1_IN_AF2_FID  0
#define C2IP2_IN_AF2_FID  0
```

Example Classification Code (part 4)

```
#define C1_IN_AF3_FID     5
#define C2IP1_IN_AF3_FID  6
#define C2IP2_IN_AF3_FID  7

#define C1_IN_AF4_FID     8
#define C2IP1_IN_AF4_FID  0
#define C2IP2_IN_AF4_FID  0

#define C1_OUT_EF_FID     9
#define C2IP1_OUT_EF_FID  9
#define C2IP2_OUT_EF_FID  9

#define C1_OUT_AF1_FID    10
#define C2IP1_OUT_AF1_FID 10
#define C2IP2_OUT_AF1_FID 10

#define C1_OUT_AF2_FID    0
#define C2IP1_OUT_AF2_FID 0
#define C2IP2_OUT_AF2_FID 0

#define C1_OUT_AF3_FID    11
#define C2IP1_OUT_AF3_FID 12
#define C2IP2_OUT_AF3_FID 13

#define C1_OUT_AF4_FID    14
#define C2IP1_OUT_AF4_FID 0
#define C2IP2_OUT_AF4_FID 0
```

Example Classification Code (part 5)

```
/* ---------------------------------------------------------------------
 * Destination IDs
 */
#define C1_IN_EF_DID      6
#define C2IP1_IN_EF_DID   11
#define C2IP2_IN_EF_DID   11

#define C1_IN_AF1_DID     7
#define C2IP1_IN_AF1_DID  12
#define C2IP2_IN_AF1_DID  12

#define C1_IN_AF2_DID     8
#define C2IP1_IN_AF2_DID  13
#define C2IP2_IN_AF2_DID  13

#define C1_IN_AF3_DID     9
#define C2IP1_IN_AF3_DID  14
#define C2IP2_IN_AF3_DID  14

#define C1_IN_AF4_DID     10
#define C2IP1_IN_AF4_DID  15
#define C2IP2_IN_AF4_DID  15

#define C1_OUT_EF_DID     1
#define C2IP1_OUT_EF_DID  1
#define C2IP2_OUT_EF_DID  1

#define C1_OUT_AF1_DID    2
#define C2IP1_OUT_AF1_DID 2
#define C2IP2_OUT_AF1_DID 2
```

Example Classification Code (part 6)

```
#define C1_OUT_AF2_DID    3
#define C2IP1_OUT_AF2_DID 3
#define C2IP2_OUT_AF2_DID 3

#define C1_OUT_AF3_DID    4
#define C2IP1_OUT_AF3_DID 4
#define C2IP2_OUT_AF3_DID 4

#define C1_OUT_AF4_DID    5
#define C2IP1_OUT_AF4_DID 5
#define C2IP2_OUT_AF4_DID 5

/* ----------------------------------------------------------------------
 * Root functions for first and second pass
 */
SETUP   ROOT            (Pass1);
SETUP   REPLAYROOT      (Pass2);

SETUP PROTO (asiPoliceEOF0,24,16,24);
SETUP PROTO (asiPoliceEOF0,24,16,8,16);
SETUP PROTO (fQueueEOF,2,19,6,1,2,1,11,13);
SETUP PROTO (fTransmit, 1, 1, 20, 16, 5, 8, 8, 18);
SETUP PROTO (fTransmit, 1, 1, 20, 16, 5, 8, 2, 8, 16);
```

Example Classification Code (part 7)

```
/*
 * **********************************************************************
 * *****                           PASS 1                           *****
 * **********************************************************************
 */
Pass1: fbrnz ($framerErr:1, FirstPassException)
       fbrnz ($framerEOF:1, ProcessLastBlock)
       fQueue (0:2, $portNumber:19, $offset:6, 0:1, 0:2);

ProcessLastBlock:
 fQueueEOF(0:2,$portNumber:19,$offset:6,0:1,0:2,0:1,$portNumber:11,0:13);

/*
 * **********************************************************************
 * *****                           PASS 2                           *****
 * **********************************************************************
 * Process Ethernet header, only supporting IP
 */
Pass2:          fSkip(96)
                0x0800:16
                DemuxPort($tag);

/* ----------------------------------------------------------------------
 * Demultiplex based on port (extracted from $tag)
 */
DemuxPort:         C1_PORT:3 BITS:21      ProcessClTraffic();
DemuxPort:         C2_PORT:3 BITS:21      ProcessClTraffic();
DemuxPort:         IN_PORT:3 BITS:21      ProcessIncomingTraffic();
```

Example Classification Code (part 8)

```
/* ----------------------------------------------------------------------
 * Process traffic from clients
 * Classification based on: Port, Protocol and Source IP
 */
ProcessClTraffic:
  fStartIpv4HdrCksum($currOffset:6) /* start calculating header chksum  */
  0x4:4
  hlen = fExtract(4)                /* get header length                */
  DSCP_CU = fExtract(8)             /* get DSCP_CU (TOS)                */
  fSkip(48)                         /* skip to TTL                      */
  VerifyTTL()                       /* verify TTL                       */
  proto = fExtract(8)               /* get transport layer protocol     */
  fSkip(16)                         /* skip IP header checksum          */
  port = getPortNum($tag)           /* get port number                  */
  code = getFlowCode($port:3)       /* get code for a flow              */
  fskip(32)                         /* skip destination IP              */
  opt_words = fSub(hlen:16, 5:16)   /* IPv4 options size in words       */
  opt_bits  = fShift(opt_words:24,LEFT_SHIFT:1,5:5) /* IP options bits  */
  fSkip(opt_bits)                                   /* skip IPv4 options*/
  data = getDataType($proto:8)                      /* get payload type */
  FID = getFID($code:2, OUT:1, $data:2,$DSCP_CU:8)  /* get Flow ID      */
  /* assign color based on Flow ID and policing algorithm               */
  color = asiPoliceEOF0($FID:24,$currLength:16,0:24)
  DID = getDID($FID:8)            /* get Destination ID                 */
  fSkipToEnd()                    /* skip to the end of packet          */
  checksum = fGetIpHdrCksum()     /* get header checksum                */
  checkChecksum($checksum:2)      /* verify checksum                    */
  TMflags_DSCP = get_TMflags_DSCP(@FID:3,$color:16) /* get TM flags     */
  /* finish second pass and transmit */
  fTransmit (0:1, 0:1, $DID:20, 0:16, 0:5, 0:8, $TMflags_DSCP:8, 0:18);
```

Example Classification Code (part 9)

```
/* ----------------------------------------------------------------------
 * Process traffic for clients
 * Classification based on: Protocol, Destination IP and DSCP (TOS) field
 */
ProcessIncomingTraffic:
  fStartIpv4HdrCksum($currOffset:6) /* start calculating header chksum  */
  0x4:4
  hlen = fExtract(4)                /* get header length                */
  DSCP_CU = fExtract(8)             /* get DSCP_CU (TOS)                */
  fSkip(48)                         /* skip to TTL                      */
  VerifyTTL()                       /* verify TTL                       */
  proto = fExtract(8)               /* get transport layer protocol     */
  fSkip(48)                         /* skip to destination IP           */
  code = getFlowCode(IN_PORT:3)     /* get code for a traffic flow      */
  opt_words = fSub(hlen:16, 5:16)   /* IPv4 options length in words     */
  opt_bits  = fShift(opt_words:24,LEFT_SHIFT:1,5:5) /* IP options bits  */
  fSkip(opt_bits)                                   /* skip IPv4 options*/
  data = getDataType($proto:8)                      /* get payload type */
  FID = getFID($code:2,IN:1,$data:2,$DSCP_CU:8)     /* get Flow ID      */
  /* assign color based on Flow ID and policing algorithm               */
  color = asiPoliceEOF0($FID:24,$currLength:16,$DSCP_CU:8,0:16)
  DID = getDID($FID:8)            /* get Destination ID                 */
  fSkipToEnd()                    /* skip to the end of packet          */
  checksum = fGetIpHdrCksum()     /* get header checksum                */
  checkChecksum($checksum:2)      /* verify checksum                    */
  TMflags_DSCP = get_TMflags_DSCP(@FID:3,$color:16) /* get TM flags     */
  /* finish second pass and transmit */
  fTransmit (0:1, 0:1, $DID:20, 0:16, 0:5, 0:8, $TMflags_DSCP:8, 0:18);
```

Example Classification Code (part 10)

```
/* ---------------------------------------------------------------------
 * Set of functions to get payload type (Voice, Video or Unknown)
 */
getDataType: IPT_UDP:8 RTP_PORT:16
             fSkip(57) /* skip to RTP PT field */
             getRTPDataType();
getDataType: IPT_UDP:8 BITS:16
             checkUDPDstPort();
getDataType: BITS:24 fReturn(UNK);

checkUDPDstPort: RTP_PORT:16
                 fSkip(41) /* skip to RTP PT field */
                 getRTPDataType();
checkUDPDstPort: BITS:16 fReturn(UNK);

getRTPDataType: RANGE(0, 19):7 fReturn(AUD);
getRTPDataType: RANGE(31,34):7 fReturn(VID);
getRTPDataType: RANGE(25,26):7 fReturn(VID);
getRTPDataType: 28:7           fReturn(VID);
getRTPDataType: BITS:7         fReturn(UNK);

/* ---------------------------------------------------------------------
 * Get Flow ID
 */
getFID: CL1:2 OUT:1 AUD:2 0b101110 BITS:2 fReturn(EF,C1_OUT_EF_FID);
getFID: CL1:2 OUT:1 VID:2 0b101110 BITS:2 fReturn(EF,C1_OUT_EF_FID);
getFID: CL1:2 OUT:1 UNK:2 0b101110 BITS:2 fReturn(EF,C1_OUT_EF_FID);
getFID: CL1:2 OUT:1 AUD:2 BITS:8 fReturn(AF3,C1_OUT_AF3_FID);
getFID: CL1:2 OUT:1 VID:2 BITS:8 fReturn(AF4,C1_OUT_AF4_FID);
getFID: CL1:2 OUT:1 BITS:10   fReturn(AF1,C1_OUT_AF1_FID);
```

Example Classification Code (part 11)

```
getFID: CL2IP1:2 OUT:1 AUD:2 0b101110 BITS:2 fReturn(EF,C2IP1_OUT_EF_FID);
getFID: CL2IP1:2 OUT:1 VID:2 0b101110 BITS:2 fReturn(EF,C2IP1_OUT_EF_FID);
getFID: CL2IP1:2 OUT:1 UNK:2 0b101110 BITS:2 fReturn(EF,C2IP1_OUT_EF_FID);
getFID: CL2IP1:2 OUT:1 AUD:2 BITS:8 fReturn(AF3,C2IP1_OUT_AF3_FID);
getFID: CL2IP1:2 OUT:1 BITS:10   fReturn(AF1,C2IP1_OUT_AF1_FID);

getFID: CL2IP2:2 OUT:1 AUD:2 0b101110 BITS:2 fReturn(EF,C2IP2_OUT_EF_FID);
getFID: CL2IP2:2 OUT:1 VID:2 0b101110 BITS:2 fReturn(EF,C2IP2_OUT_EF_FID);
getFID: CL2IP2:2 OUT:1 UNK:2 0b101110 BITS:2 fReturn(EF,C2IP2_OUT_EF_FID);
getFID: CL2IP2:2 OUT:1 AUD:2 BITS:8 fReturn(AF3,C2IP2_OUT_AF3_FID);
getFID: CL2IP2:2 OUT:1 BITS:10   fReturn(AF1,C2IP2_OUT_AF1_FID);

getFID: CL1:2 IN:1 AUD:2 0b101110 BITS:2 fReturn(EF,C1_IN_EF_FID);
getFID: CL1:2 IN:1 VID:2 0b101110 BITS:2 fReturn(EF,C1_IN_EF_FID);
getFID: CL1:2 IN:1 UNK:2 0b101110 BITS:2 fReturn(EF,C1_IN_EF_FID);
getFID: CL1:2 IN:1 AUD:2 0b011    BITS:5 fReturn(AF3,C1_IN_AF3_FID);
getFID: CL1:2 IN:1 VID:2 0b100    BITS:5 fReturn(AF4,C1_IN_AF4_FID);
getFID: CL1:2 IN:1 BITS:10   fReturn(AF1,C1_IN_AF1_FID);

getFID: CL2IP1:2 IN:1 AUD:2 0b101110 BITS:2 fReturn(EF,C2IP1_IN_EF_FID);
getFID: CL2IP1:2 IN:1 VID:2 0b101110 BITS:2 fReturn(EF,C2IP1_IN_EF_FID);
getFID: CL2IP1:2 IN:1 UNK:2 0b101110 BITS:2 fReturn(EF,C2IP1_IN_EF_FID);
getFID: CL2IP1:2 IN:1 AUD:2 0b011    BITS:5 fReturn(AF3,C2IP1_IN_AF3_FID);
getFID: CL2IP1:2 IN:1 BITS:10   fReturn(AF1,C2IP1_IN_AF1_FID);

getFID: CL2IP2:2 IN:1 AUD:2 0b101110 BITS:2 fReturn(EF,C2IP2_IN_EF_FID);
getFID: CL2IP2:2 IN:1 VID:2 0b101110 BITS:2 fReturn(EF,C2IP2_IN_EF_FID);
getFID: CL2IP2:2 IN:1 UNK:2 0b101110 BITS:2 fReturn(EF,C2IP2_IN_EF_FID);
getFID: CL2IP2:2 IN:1 AUD:2 0b011    BITS:5 fReturn(AF3,C2IP2_IN_AF3_FID);
getFID: CL2IP2:2 IN:1 BITS:10   fReturn(AF1,C2IP2_IN_AF1_FID);
```

Example Classification Code (part 12)

```
/* ----------------------------------------------------------------------
 * Get Destination ID
 */
getDID: C1_OUT_EF_FID:8   fReturn(C1_OUT_EF_DID);
getDID: C1_OUT_AF1_FID:8  fReturn(C1_OUT_AF1_DID);
getDID: C1_OUT_AF2_FID:8  fReturn(C1_OUT_AF2_DID);
getDID: C1_OUT_AF3_FID:8  fReturn(C1_OUT_AF3_DID);
getDID: C1_OUT_AF4_FID:8  fReturn(C1_OUT_AF4_DID);

getDID: C2IP1_OUT_EF_FID:8   fReturn(C2IP1_OUT_EF_DID);
getDID: C2IP1_OUT_AF1_FID:8  fReturn(C2IP1_OUT_AF1_DID);
getDID: C2IP1_OUT_AF2_FID:8  fReturn(C2IP1_OUT_AF2_DID);
getDID: C2IP1_OUT_AF3_FID:8  fReturn(C2IP1_OUT_AF3_DID);
getDID: C2IP1_OUT_AF4_FID:8  fReturn(C2IP1_OUT_AF4_DID);

getDID: C2IP2_OUT_EF_FID:8   fReturn(C2IP2_OUT_EF_DID);
getDID: C2IP2_OUT_AF1_FID:8  fReturn(C2IP2_OUT_AF1_DID);
getDID: C2IP2_OUT_AF2_FID:8  fReturn(C2IP2_OUT_AF2_DID);
getDID: C2IP2_OUT_AF3_FID:8  fReturn(C2IP2_OUT_AF3_DID);
getDID: C2IP2_OUT_AF4_FID:8  fReturn(C2IP2_OUT_AF4_DID);

getDID: C1_IN_EF_FID:8    fReturn(C1_IN_EF_DID);
getDID: C1_IN_AF1_FID:8   fReturn(C1_IN_AF1_DID);
getDID: C1_IN_AF2_FID:8   fReturn(C1_IN_AF2_DID);
getDID: C1_IN_AF3_FID:8   fReturn(C1_IN_AF3_DID);
getDID: C1_IN_AF4_FID:8   fReturn(C1_IN_AF4_DID);
```

Example Classification Code (part 13)

```
getDID: C2IP1_IN_EF_FID:8    fReturn(C2IP1_IN_EF_DID);
getDID: C2IP1_IN_AF1_FID:8   fReturn(C2IP1_IN_AF1_DID);
getDID: C2IP1_IN_AF2_FID:8   fReturn(C2IP1_IN_AF2_DID);
getDID: C2IP1_IN_AF3_FID:8   fReturn(C2IP1_IN_AF3_DID);
getDID: C2IP1_IN_AF4_FID:8   fReturn(C2IP1_IN_AF4_DID);

getDID: C2IP2_IN_EF_FID:8    fReturn(C2IP2_IN_EF_DID);
getDID: C2IP2_IN_AF1_FID:8   fReturn(C2IP2_IN_AF1_DID);
getDID: C2IP2_IN_AF2_FID:8   fReturn(C2IP2_IN_AF2_DID);
getDID: C2IP2_IN_AF3_FID:8   fReturn(C2IP2_IN_AF3_DID);
getDID: C2IP2_IN_AF4_FID:8   fReturn(C2IP2_IN_AF4_DID);

/* -----------------------------------------------------------------------
 * Get TM flags (color) and DSCP, depending on class
 * and color of out-bound packet: GR=Green YE=Yellow RED=RE
 */
get_TMflags_DSCP: EF:3  0b00:2 BITS:14 fReturn(0b00101110); /* GR EF   */
get_TMflags_DSCP: EF:3  0b01:2 BITS:14 fReturn(0b01101110); /* YE EF   */
get_TMflags_DSCP: EF:3  0b1:1  BITS:15 fReturn(0b10101110); /* RE EF   */
get_TMflags_DSCP: AF1:3 0b00:2 BITS:14 fReturn(0b00001010); /* GR AF11 */
get_TMflags_DSCP: AF1:3 0b01:2 BITS:14 fReturn(0b01001100); /* YE AF12 */
get_TMflags_DSCP: AF1:3 0b1:1  BITS:15 fReturn(0b10001110); /* RE AF13 */
get_TMflags_DSCP: AF2:3 0b00:2 BITS:14 fReturn(0b00010010); /* GR AF21 */
get_TMflags_DSCP: AF2:3 0b01:2 BITS:14 fReturn(0b01010100); /* YE AF22 */
get_TMflags_DSCP: AF2:3 0b1:1  BITS:15 fReturn(0b10010110); /* RE AF23 */
get_TMflags_DSCP: AF3:3 0b00:2 BITS:14 fReturn(0b00011010); /* GR AF31 */
get_TMflags_DSCP: AF3:3 0b01:2 BITS:14 fReturn(0b01011100); /* YE AF32 */
get_TMflags_DSCP: AF3:3 0b1:1  BITS:15 fReturn(0b10011110); /* RE AF33 */
get_TMflags_DSCP: AF4:3 0b00:2 BITS:14 fReturn(0b00100010); /* GR AF41 */
```

Example Classification Code (part 14)

```
get_TMflags_DSCP: AF4:3 0b01:2 BITS:14 fReturn(0b01100100); /* YE AF42 */
get_TMflags_DSCP: AF4:3 0b1:1  BITS:15 fReturn(0b10100110); /* RE AF43 */

/* ----------------------------------------------------------------------
 * Extract port number from $tag
 */
getPortNum:         C1_PORT:3 BITS:21       fReturn(C1_PORT);
getPortNum:         C2_PORT:3 BITS:21       fReturn(C2_PORT);
getPortNum:         IN_PORT:3 BITS:21       fReturn(IN_PORT);

/* ----------------------------------------------------------------------
 * Get flow code based on port and source or destination IP
 */
getFlowCode:    C1_PORT:3 IP_C1      fReturn(CL1);
getFlowCode:    C2_PORT:3 IP1_C2     fReturn(CL2IP1);
getFlowCode:    C2_PORT:3 IP2_C2     fReturn(CL2IP2);
getFlowCode:    IN_PORT:3 IP_C1      fReturn(CL1);
getFlowCode:    IN_PORT:3 IP1_C2     fReturn(CL2IP1);
getFlowCode:    IN_PORT:3 IP2_C2     fReturn(CL2IP2);

/* ----------------------------------------------------------------------
 * Verify the Time To Live field in the IP Header
 */
VerifyTTL: 0:8    SecondPassException();
VerifyTTL: BITS:8 fReturn();
```

Example Classification Code (part 15)

```
/* ----------------------------------------------------------------------
 * Verify checksum
 */
checkChecksum: 0b1 0b1 fReturn(); /* check passed  */
/* otherwise error handler is called automatically */

/* ----------------------------------------------------------------------
 * Main error handler
 */
MyError: 0b0 BITS:7 FirstPassException();  /* first pass  */
MyError: 0b1 BITS:7 SecondPassException(); /* second pass */

/* ----------------------------------------------------------------------
 * First pass error handler
 * Discard
 */
FirstPassException:
                fQueueEOF(0:2, 0:19, $offset:6, 0:1, 0:2, 1:1, 0:24);

/*----------------------------------------------------------------------
 * Second pass error handler
 * Send PDU to RSP for discard
 */
SecondPassException:
                fSkipToEnd()
                fTransmit (0:1, 0:1, 0:20, 0:16, 0:5, 0:10, 0:24);
```

Error Handling

- FPL provides error handling in case none of the patterns in a tree function matches
- 1-bit argument specifies whether error occurred in pass 1 or 2
- To initialize the error handler:

```
SETUP ERROR(MyError);
```

- Argument used to choose a pass

```
MyError: 0b0 BITS:7 FirstPassException();  /* first pass   */
MyError: 0b1 BITS:7 SecondPassException(); /* second pass */
```

Example Policing Script (part 1)

```
/* ds_police_eof_0.asl - policing functions for DiffServ example        */

/*
 * Policing script implementing dual token bucket algorithm
 */

#include "policeNp5.h"

#define MAX_RTC_TIME 0xffffffff


/*
 * Dual token bucket parameters
 * (these are initialized during configuration)
 * Bit rates are measured in RTC ticks per byte.
 * Burst sizes are measured in RTC ticks, i.e.
 * BurstSize_in_ticks = BurstSize_in_bytes x ticks_per_byte
 */
unsigned PBR param_block[0:1] input;   /* peak bit rate        */
unsigned SBR param_block[2:3] input;   /* sustained bit rate   */
unsigned PBS param_block[4:11] input;  /* peak burst size      */
unsigned SBS param_block[12:19] input; /* sustained burst size */

/*
 * Previous pdu arrival time
 * (initialized to zero during configuration)
 */
unsigned last_pdu_arrival param_block[20:23] inout;
```

Example Policing Script (part 2)

```
/*
 * token counter for peak bucket
 * (initialized to PBS during configuration)
 */
unsigned p_tokens param_block[24:31]  inout;

/* token counter for sustained bucket
 * (initialized to SBS during configuration)
 */
unsigned s_tokens param_block[32:39] inout;

/* PDU length in bytes (passed by FPP) */
unsigned pdu_length fpp_args[0:1] input;

/* DSCP_CU field (passed by FPP) */
unsigned DSCP_CU    fpp_args[2];

/* RTC ticks since last packet arrived, temporary variable */
unsigned(4) delta_t;

/* PDU length in RTC ticks for sustained rate, temporary variable */
unsigned(4) pdu_peak_len_t;

/* PDU length in RTC ticks for peak rate, temporary variable */
unsigned(4) pdu_sust_len_t;

/* resulting predicate bits */
boolean PeakBucketFailed $[15] output;
boolean SustBucketFailed $[14] output;
```

Example Policing Script (part 3)

```
script dual_tbucket {

  delta_t = (MAX_RTC_TIME - last_pdu_arrival) + current_time;
  if (current_time >= last_pdu_arrival) {
    delta_t = current_time - last_pdu_arrival;
  }

  pdu_peak_len_t = pdu_length*PBR;
  pdu_sust_len_t = pdu_length*SBR;


  /* update first bucket */
  p_tokens = p_tokens + delta_t;
  if (p_tokens > PBS) {
    p_tokens = PBS;
  }

  /* update second bucket */
  s_tokens = s_tokens + delta_t;
  if (s_tokens > SBS) {
    s_tokens = SBS;
  }

  /* assign color bits (account for pre-coloring) */
  SustBucketFailed = (s_tokens < pdu_sust_len_t)||((DSCP_CU&0x10)==0x10);
  PeakBucketFailed = (p_tokens < pdu_peak_len_t)||((DSCP_CU&0x18)==0x18);
```

Example Policing Script (part 4)

```
  if (!PeakBucketFailed) {
    /* update peak bucket */
    p_tokens = p_tokens - pdu_peak_len_t;
  }

  if (!SustBucketFailed) {
    /* update sustained bucket */
    s_tokens = s_tokens - pdu_sust_len_t;
  }

  /* timestamp */
  last_pdu_arrival = current_time;
}
```

Buffer Management And Packet Discard

- Implementation of *Weighted RED* (*WRED*)
- Results of policing passed to the Traffic Manager in two individual bits
  - PeakBucketFailed
  - SustBucketFailed
- Implementation uses intervals rather than floating point

Buffer Management And Discard Code (part 1)

```
/* ds_tm_wred.asl - buffer management for DiffServ example              */

/*
 *  Discrete Weighted RED for tri-color scheme
 */
#include "tmNp5.h"

#define CLR_GREEN  0b00
#define CLR_YELLOW 0b01
#define CLR_RED    0b10

/* Queue size limits for different colors */

unsigned qthresh_red_min        param_block_in_extended[ 0:1 ];
unsigned qthresh_red_max        param_block_in_extended[ 2:3 ];
unsigned qthresh_yellow_min     param_block_inout_extended[ 0:1 ];
unsigned qthresh_yellow_max     param_block_inout_extended[ 2:3 ];
unsigned qthresh_green_min      param_block_inout_extended[ 4:5 ];
unsigned qthresh_green_max      param_block_inout_extended[ 6:7 ];

/* average queue size */
unsigned Qaverage               param_block_inout_extended[ 8:9 ];

/* Queue steps (4 steps per interval) */
unsigned qstep_red     param_block_inout_extended[ 10:11 ];
unsigned qstep_yellow  param_block_inout_extended[ 12:13 ];
unsigned qstep_green   param_block_inout_extended[ 14:15 ];
/* ds_tm_wred.asl - buffer management for DiffServ example              */
```

Buffer Management And Discard Code (part 2)

```
/* color of the PDU (passed by FPL) */
unsigned color parameters_tm[1];

/* drop probability for current PDU */
unsigned(1) drop_pr;

/*
 * weighted running average for the queue size
 * as if current PDU is not dropped
 */
unsigned(2) q_average;

/* temporaries for min and max queue sizes */
unsigned(2) qthresh_min;
unsigned(2) qthresh_max;

/* step to increment queue size threshold */
unsigned(2) q_step;

script tm_wred {

  /*
   * compute average queue size as if the PDU is not dropped
   * Qaverage = Qaverage + 1/8*(Qcurrent-Qaverage)
   */
   q_average = Qaverage + ((blocks_in_Q + pdu_blocks)>>3) - (Qaverage>>3);
```

Buffer Management And Discard Code (part 3)

```
/* Set thresholds according to color of current PDU */

if (color == CLR_GREEN) {
      qthresh_min = qthresh_green_min;
      qthresh_max = qthresh_green_max;
      q_step = qstep_green;
 } else {
      if (color ==  CLR_YELLOW) {
            qthresh_min = qthresh_yellow_min;
            qthresh_max = qthresh_yellow_max;
            q_step = qstep_yellow;
      } else {
            qthresh_min = qthresh_red_min;
            qthresh_max = qthresh_red_max;
            q_step = qstep_red;
      }
 }

 /*
  * Calculate drop probability, depending on the
  * threshold interval for the current packet
  */

 if ((q_average > qthresh_max) ||
     (sch_mem > sch_thresh) ||
     (port_mem > port_thresh) ||
     (used_mem > glob_thresh1)) {
        drop_pr = 0xff;  /* (100% drop probability) */
```

Scheduler Ports, Queues, and Weights

Port Number	Port Manager	Logical Port	QoS Queue	DiffServ Class	Weight for SDWRR
0	0	0	1	EF	16
0	0	0	2	AF1	1
0	0	0	3	AF2	2
0	0	0	4	AF3	4
0	0	0	5	AF4	8
1	1	1	6	EF	16
1	1	1	7	AF1	1
1	1	1	8	AF2	2
1	1	1	9	AF3	4
1	1	1	10	AF4	8
2	2	2	11	EF	16
2	2	2	12	AF1	1
2	2	2	13	AF2	2
2	2	2	14	AF3	4
2	2	2	15	AF4	8

Scheduling Parameters

- Each queue stores three *scheduling parameters*
- Example: for SDWRR, parameters corresponds to the three limits used in the algorithm
- To achive weighted bandwidth sharing, assigned values are function of queue weight

Scheduling Parameters
(continued)

- The three limits are assigned for the i^{th} queue as follows:

$$limit\,1_i = \frac{W_i L_{\max}}{3}$$

$$limit\,2_i = 2 limit\,1_i = \frac{2 W_i L_{\max}}{3}$$

$$limit\,3_i = 3 limit\,1_i = W_i L_{\max}$$

- For Ethernet, L_{max} is 1514

Dynamic Scheduling

- Rate of given queue depends on other queues
  - no a priori limit
  - If no other queues have traffic, given queue can consume all available bandwidth
- Handled by *Shared Dynamic Rescheduler* (known as *Rate Limiting Dynamic Rescheduler*)

Example SDWRR Scheduler (part 1)

```
/* ds_ts_sdwrr.asl -- scheduling script for DiffServ example            */
/*
 * Smoothed Weighted Deficit Round Robin scheduler
 */
#include "tsNp5.h"

/*
 * Typical limits assignment:
 * Quantum = max_PDU_size/3
 * limit1 = queue_weight x Quantum
 * limit2 = queue_weight x Quantum x 2
 * limit3 = queue_weight x Quantum x 3
 */
unsigned limit1       param_block_inout_extended [0:3];
unsigned limit2       param_block_inout_extended [4:7];
unsigned limit3       param_block_inout_extended [8:11];
unsigned expense      param_block_inout_extended [12:15];

unsigned(4) updated_expense;
unsigned(1) FIFO_advance;

/*
 * Maximal rate is used to penalize oversubscribed queues with
 * shared dynamic scheduler. Rates are measured in block times.
 */
unsigned max_rate          param_block_inout_extended [16:17];
unsigned average_rate      param_block_inout_extended [18:19];
unsigned last_sched_time   param_block_inout_extended [20:23];
```

Example SDWRR Scheduler (part 2)

```
script sdwrr {
   if (is_first) {

     /* average_rate= 1/8 (current_rate-average_rate) + average rate   */
     average_rate = average_rate + ((current_time - last_sched_time)>>3)
                      - (average_rate>>3);

     /* update scheduling timestamp */
     last_sched_time = current_time + pdu_ttt;

     /*
      * see if the queue had just entered busy period,
      * in which case initialize expense to 0
      */
     if (is_Q_new_to_FIFO)
        expense = 0;

     /* calculate new expense */
     updated_expense = expense + pdu_length;

     /* calculate how many FIFO lists this queue should advance */
     FIFO_advance = 0;
     expense = updated_expense;
     if (updated_expense > limit1) {
         FIFO_advance = 1;
         expense = updated_expense - limit1;
     }
```

Example SDWRR Scheduler (part 3)

```
if (updated_expense > limit2) {
    FIFO_advance = 2;
    expense = updated_expense - limit2;
}
if ( updated_expense > limit3) {
    FIFO_advance = 3;
    expense = updated_expense - limit3;
}

/*
 * calculate next FIFO list for current queue:
 *   current = (current + advance) mod 4
 */
queue_currentlist = (queue_currentlist + FIFO_advance) & 0x3;

/*
 * if maximum rate exceeded, send queue to
 * shared dynamic rescheduler
 */
if (average_rate < max_rate ) {
   send_Q_to_dynamic_rescheduler=true;
   pdu_interval = max_rate + pdu_ttt;
   upd_interval = true;
}
```

Example SDWRR Scheduler (part 4)

```
     /*
      * Go to the next FIFO list and update enqueuing list.
      * Parameter FIFO_sched_next_currentlist is set only by the hardware
      * as follows: Bits 0:1 are set to the next FIFO list, which is the
      * same as current one if there is still a non-empty queue on the
      * current one, or is set to the next non-empty FIFO list otherwise.
      * Bits 4:5 are set to (bits 0:1 + 1 ) mod 4
      * Here we leave bits 0:1 as is, and increment bits 4:5 by 2 mod 4,
      * so that enqueuing list is (current list + 3) mod 4
      */
     FIFO_sched_currentlist = (FIFO_sched_next_currentlist+0x20)&0x3f;
   }
}
```

Packet Marking (Modification)

- DiffServ uses the term *marking* to refer to insertion of a codepoint value
- Example code used SED engine to perform modification
- SED programmed with script

Example Of Packet Marking

- Copy Ethernet source and destination addresses from parameter block to the packet
- Decrement time-to-live
- Assign DiffServ codepoint
- Recompute checksum

Example Packet Marking Code (part 1)

```
/* ds_sed_ip_ethernet.asl - SED script for DiffServ example             */

#include "sedNp5.h"

/*
 *------------------------------------------------------------------------
 * DiffServ for IP over Ethernet
 */
script DS_ip_ethernet {

  unsigned    prm_dst_mac     param_block[0:5];
  unsigned    prm_src_mac     param_block[6:11];

  /* DSCP value, passed by FPP */
  unsigned    prm_dscp        flags[0];
```

Example Packet Marking Code (part 2)

```
/*
 * Only the first block of the PDU is modified.  All other blocks are
 * passed through the SED CE without modification.
 */
if (is_first) {

   /*
    * The format of the first output data block.
    * It consists of an Ethernet MAC header followed by an IP header.
    * Within the IP header, the TOS, TTL and the checksum fields are
    * defined.
    */
   unsigned        dst_mac data_block[0:5];
   unsigned        src_mac data_block[6:11];
   block           ip_header data_block[14:33];
   unsigned        ip_tos ip_header[1];
   unsigned        ip_ttl ip_header[8];
   unsigned        ip_checksum ip_header[10:11];
   unsigned(4)     checksum;
   unsigned(2)     temp;

   /* Fill in source & destination MAC addresses from parameter block */
   dst_mac = prm_dst_mac;
   src_mac = prm_src_mac;

   /*
    * Update ttl field.  Assuming that FPL code has already
    * verified that it is nonzero
    */
   ip_ttl = ip_ttl - 1;
```

Example Packet Marking Code (part 3)

```
     /* Set TOS (DSCP) and adjust the checksum */
     temp = prm_dscp; /* overcoming conversion problems */
     checksum = ip_checksum + ip_tos + 0x0100 + (~temp);
     ip_tos = prm_dscp;

     /* Wrap checksum carry around */
     checksum = checksum[0:1]+checksum[2:3];
     checksum = checksum[0:1]+checksum[2:3];
     ip_checksum = checksum[2:3];
  }
}
```

Host Interface

- External host
  - Initializes APP550
  - Loads code and configuration information onto chip
- After chip initialized, host uses *object API* to interact with chip

Example Host Interface Code (part 1)

```
/* ds_host_iface.c - external host interface for DiffServ example       */

/*
 * Load configuration file and provide command interface for host
 */

#include <agere_np5.h>
#include <stdio.h>

#define MAXQUEUE 15        /* maximal queue ID number     */
#define MAX_FRAMESIZE 1514 /* maximal frame size          */
#define MAX_LINE_LEN  16   /* maximal command line length */

/* locations of limits and expense in parameters block */
#define LIMIT1_NUM  0
#define LIMIT2_NUM  1
#define LIMIT3_NUM  2
#define EXPENSE_NUM 3

/* read config function prototype */
int cfg_read_func(void *fp, char *buf, int len);

/* chipset handle definitions */
ag_chipset_t chipsetHdl;
ag_np5_t app550Hdl;
ag_rsp_t rspHdl;
```

Example Host Interface Code (part 2)

```
#ifdef VXWORKS
static int DiffServStart(int argc, char ** argv)
#else
int main(int argc, char ** argv)
#endif
{
   int weight, queue, rc;
   char cmd_buf[MAX_LINE_LEN];
   ag_uint32_t parameters[4];
   ag_uint8_t param_nums[]={LIMIT1_NUM,LIMIT2_NUM,LIMIT3_NUM,EXPENSE_NUM};
   char * filename;
   FILE * cfg_fp;
   ag_chipset_chip_error_t chipError;

   /* get and check arguments */
   if (argc != 2) {
     fprintf(stderr,"Usage: DiffServStart <cfg file name>\n");
     exit(-1);
   }
   filename = argv[1];
   if ((cfg_fp = fopen(filename, "r")) == NULL) {
          fprintf(stderr,"Error: can not open config file %s\n",filename);
          exit(-1);
   }
```

Example Host Interface Code (part 3)

```
/* initialize chipset */
rc = ag_chipset_init(0, &chipsetHdl);
if (rc != AG_CHIPSET_ST_SUCCESS) {
       fprintf(stderr,"Error: ag_chipset_init failed\n");
       fclose(cfg_fp);
       exit(-1);
}

/* configure chipset */
rc = ag_chipset_config(chipsetHdl, (ag_read_fn_t) cfg_read_func,
                        (void *) cfg_fp, NULL, NULL, &chipError);
if (rc != AG_CHIPSET_ST_SUCCESS) {
       fprintf(stderr,"Error: ag_chipset_config failed\n");
       fclose(cfg_fp);
       exit(-1);
}
/* close config file */
fclose(cfg_fp);

/* get object-level APP550 handle */
rc = ag_chipset_hdl_get(chipsetHdl, (char*) "APP550", 0, &app550Hdl);
if (rc != AG_CHIPSET_ST_SUCCESS) {
    fprintf(stderr,"Error: ag_chipset_hdl_get failed\n");
    exit(-1);
}
```

Example Host Interface Code (part 4)

```
/* get object-level RSP handle */
rc = ag_rsp_hdl_get(app550Hdl, &rspHdl);
if (rc != AG_ST_SUCCESS) {
    fprintf(stderr, "Error: ag_rsp_hdl_get failed\n");
    exit(-1);
}

/* start our own "shell" */
while (1) { /* do forever */
   /* read queue ID and weight from standard input */
   fprintf(stdout,"Enter Queue ID and Weight (separated by space): ");
   if ( fgets(cmd_buf,MAX_LINE_LEN,stdin) == NULL )
         exit(0); /* EOF encountered */
   queue  = atoi(strtok(cmd_buf," \n"));
   weight = atoi(strtok(NULL," \n"));
   if (queue < 1 || queue > MAXQUEUE) {
     fprintf(stderr,"Error: invalid queue number: %i\n", queue);
     continue;
   }
   if (weight < 1) {
      fprintf(stderr,"Error: invalid weight value: %i\n", weight);
      continue;
   }

   /* compute new limits */
   parameters[0] = MAX_FRAMESIZE*weight/3; /* limit1      */
   parameters[1] = parameters[0]*2;        /* limit2      */
   parameters[2] = parameters[0]*3;        /* limit3      */
   parameters[3] = 0;                      /* expense = 0 */
```

Example Host Interface Code (part 5)

```
      /* update parameters */
      rc=ag_rsp_qid_unatomic_replace_ts_parms_words(rspHdl, queue,
                                          parameters, param_nums,4);
      switch (rc) {
        case AG_ST_SUCCESS:
          /* success */
          fprintf(stdout,"Weight for queue %i set to %i\n",queue,weight);
          break;
        case AG_ST_RSP_QID_INVALID:
          fprintf(stderr, "Error: QID %i is invalid\n",queue);
          continue;
        case AG_ST_RSP_QID_NOT_EXIST:
          fprintf(stderr, "Error: QID %i does not exist\n",queue);
          continue;
        case AG_ST_RSP_INVALID_HANDLE:
          fprintf(stderr, "Error: invalid RSP handle\n");
          exit(-1);
        default:
          fprintf(stderr, "Error: invalid return code %i\n",rc);
          exit(-1);
      }
   }
   exit(0);
}
```

Example Host Interface Code (part 6)

```
/* function to read configuration file */
int cfg_read_func(void *fp, char *buf, int len) {
    int nread;
    if ((nread=fread(buf, 1, len, (FILE *)fp)) < len) {
      if (ferror((FILE *)fp)) {
           fprintf(stderr,"Error: cfg_read_func failed reading\n");
           return -1;
      }
    }
    return nread;
}
```

Summary

- We have reviewed complete code for a DiffServ system
  - Classification
  - Policing and marking
  - Buffer management and discard
  - Dynamic rescheduling

Questions?

X

Switching Fabrics

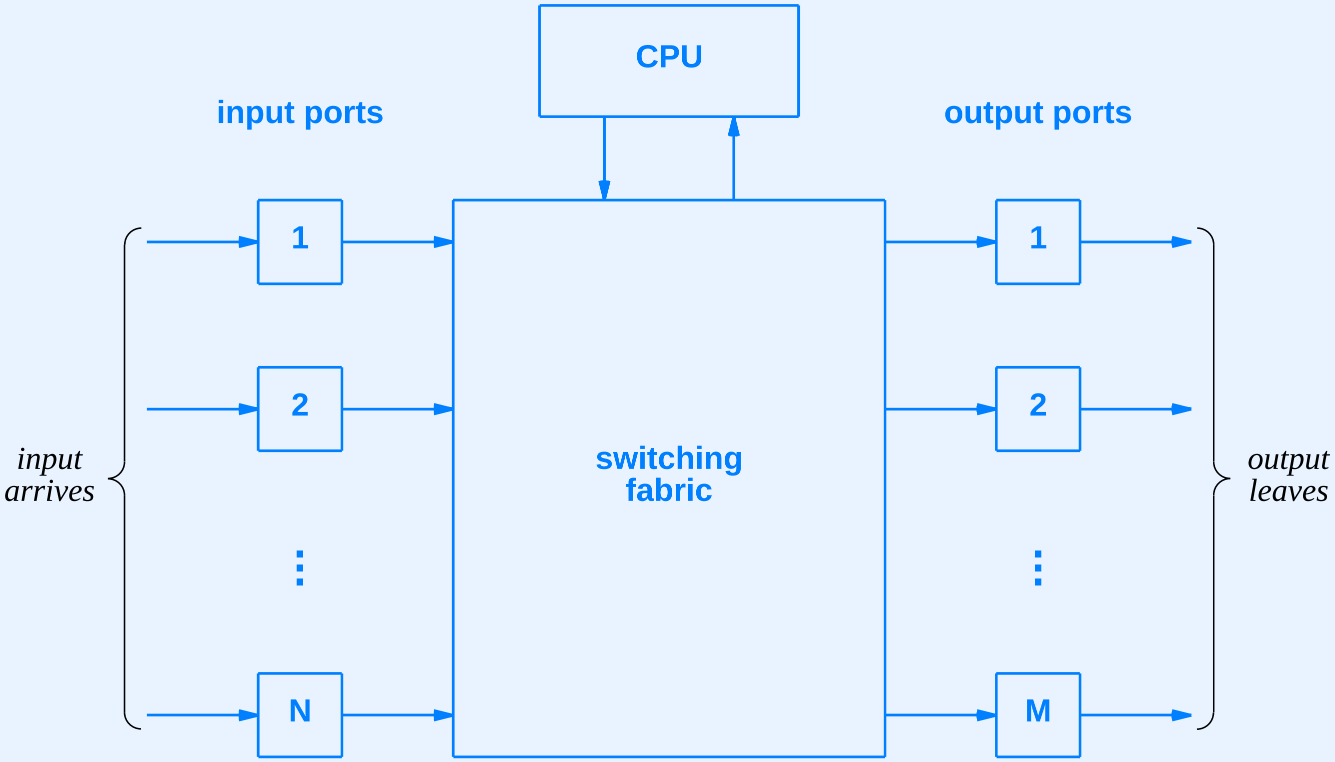
Physical Interconnection

- Physical box with backplane
- Individual *blades* plug into backplane slots
- Each blade contains one or more network connections

Logical Interconnection

- Known as *switching fabric*
- Handles transport from one blade to another
- Becomes bottleneck as number of interfaces scales

Illustration Of Switching Fabric



- Any input port can send to any output port

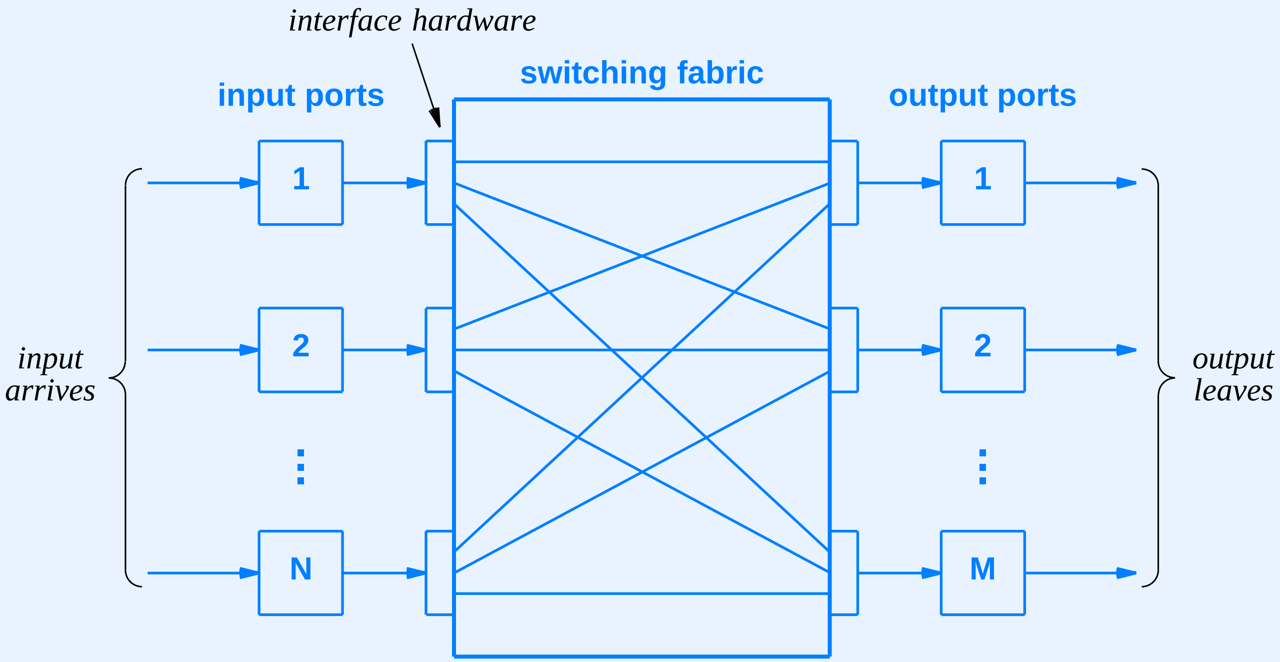
Switching Fabric Properties

- Used inside a single network system
- Interconnection among I/O ports (and possibly CPU)
- Can transfer unicast, multicast, and broadcast packets
- Scales to arbitrary data rate on any port
- Scales to arbitrary packet rate on any port
- Scales to arbitrary number of ports
- Has low overhead
- Has low cost

Types Of Switching Fabrics

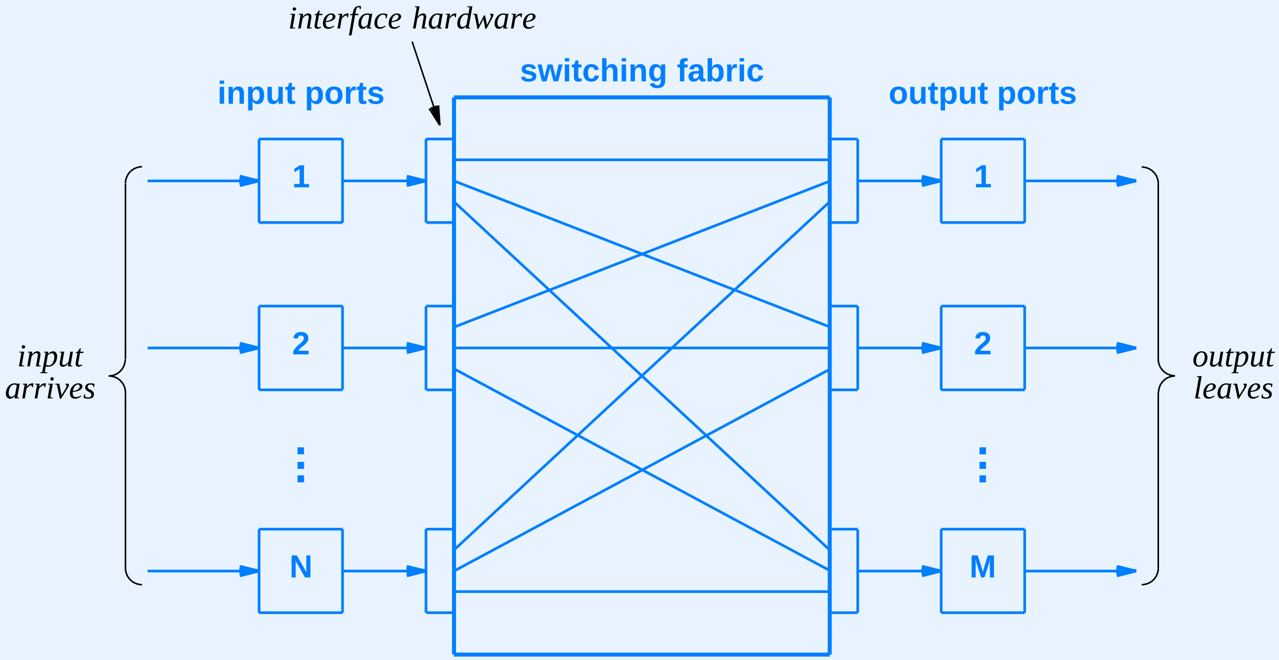
- Space-division (separate paths)
- Time-division (shared medium)

Space-Division Fabric (separate paths)



- Can use multiple paths simultaneously

Space-Division Fabric (separate paths)



- Can use multiple paths simultaneously
- Still have *port contention*

Desires

Desires

- High speed

Desires

- High speed
- Low cost

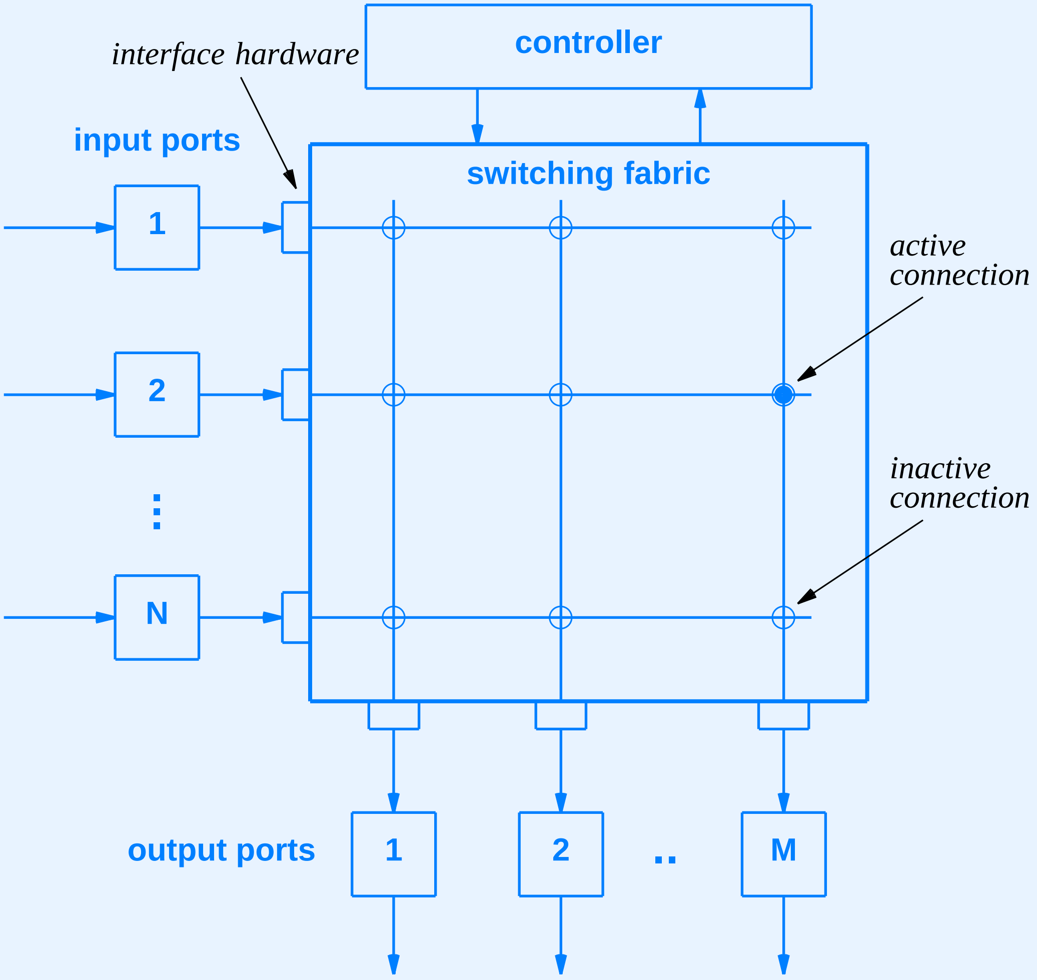
Desires

- High speed *and* low cost!

Possible Compromise

- Separation of physical paths
- Less parallel hardware
- Crossbar design

Space-Division (Crossbar Fabric)



Crossbar

- Allows simultaneous transfer on disjoint pairs of ports
- Can still have *port contention*

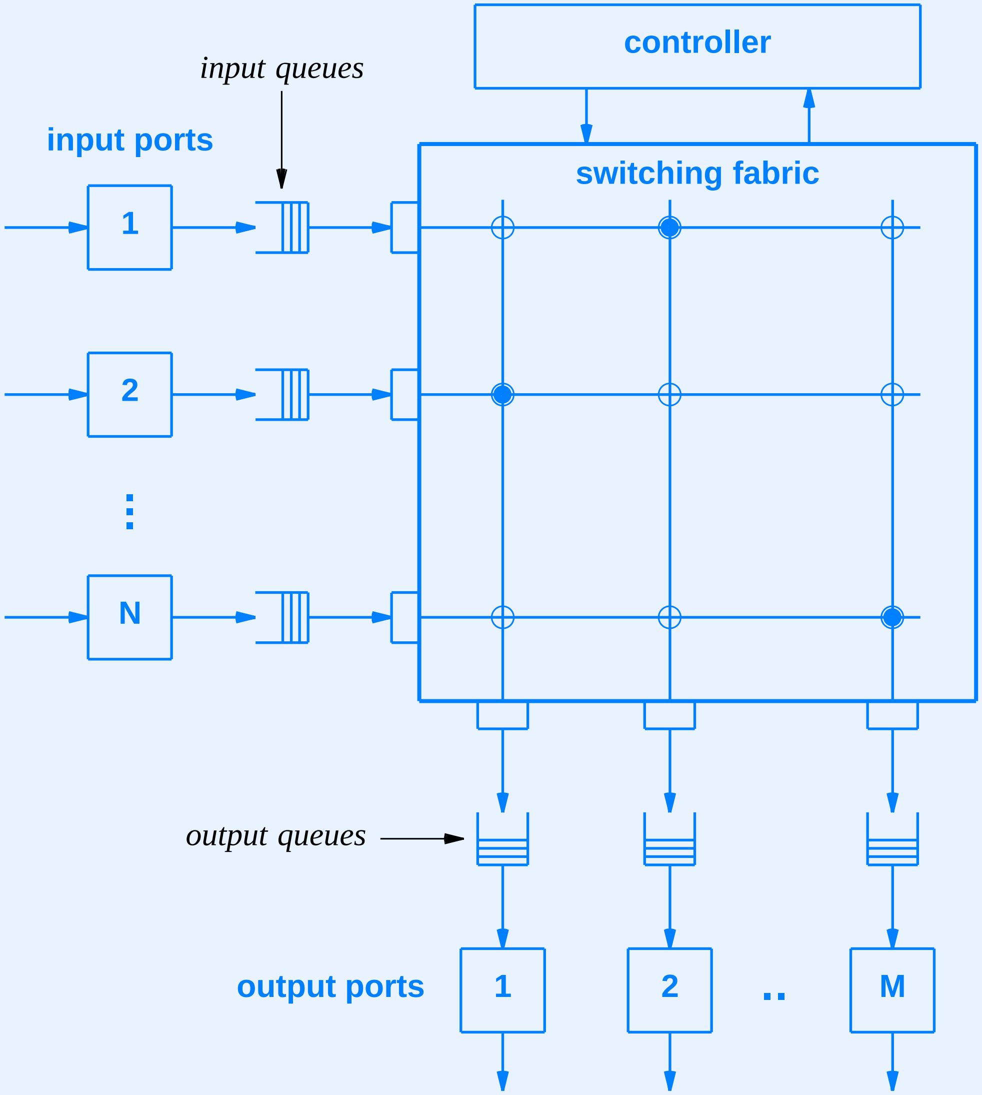
Crossbar

- Allows simultaneous transfer on disjoint pairs of ports
- Can still have *port contention*

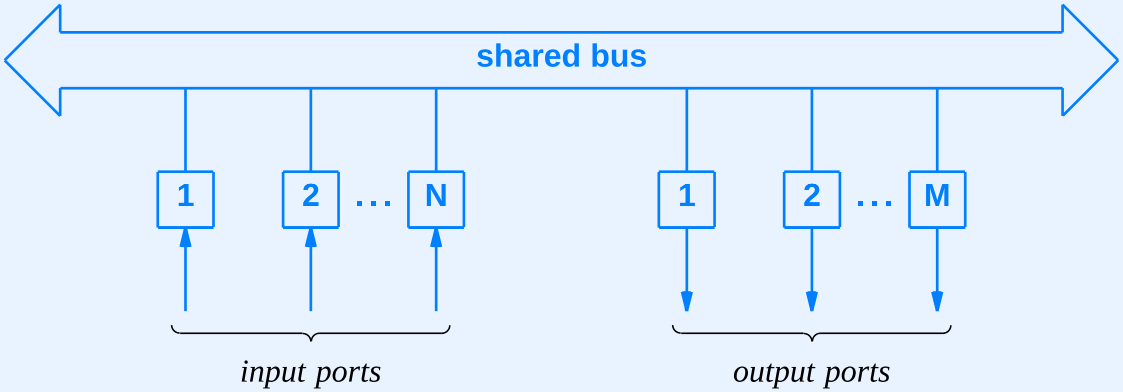
Solving Contention

- Queues (FIFOs)
  - Attached to input
  - Attached to output
  - At intermediate points

Crossbar Fabric With Queuing

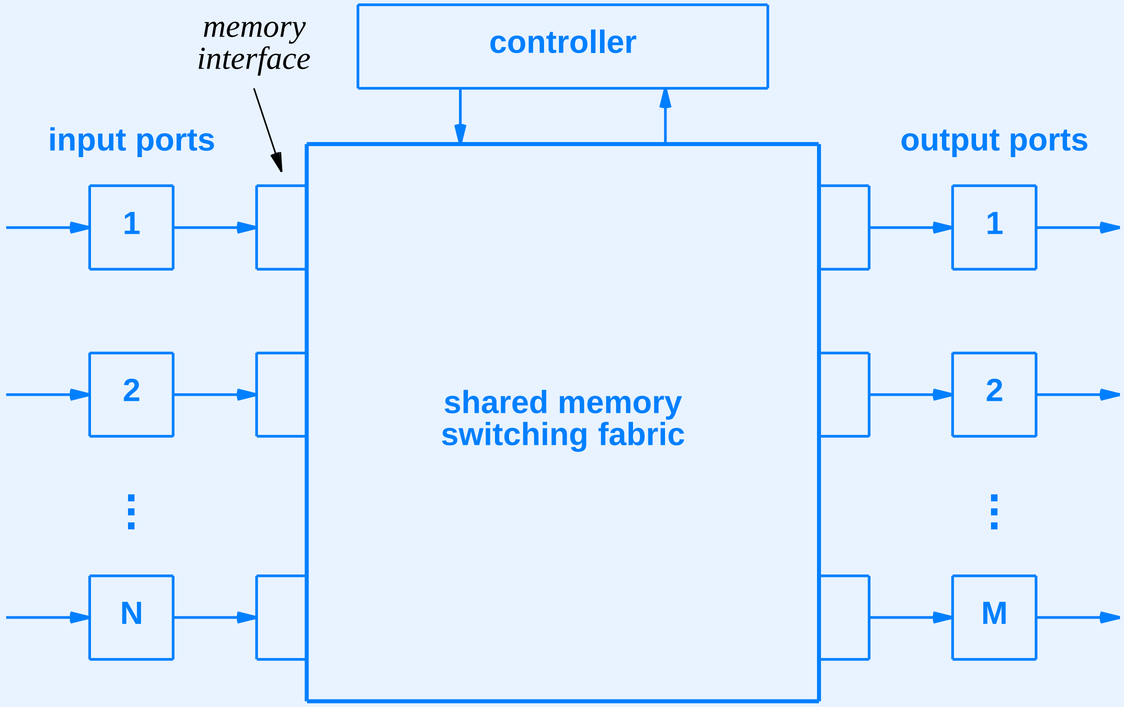


Time-Division Fabric (shared bus)



- Chief advantage: low cost
- Chief disadvantage: low speed

Time-Division Fabric (shared memory)



- *May* be better than shared bus
- Usually more expensive

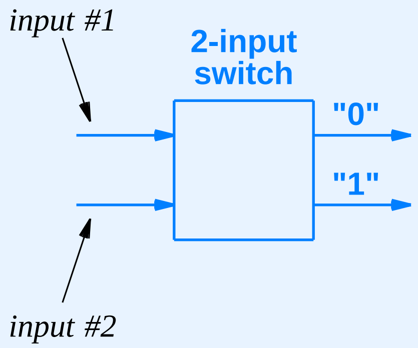
Multi-Stage Fabrics

- Compromise between pure time-division and pure space-division
- Attempt to combine advantages of each
  - Lower cost from time-division
  - Higher performance from space-division
- Technique: limited sharing

Banyan Fabric

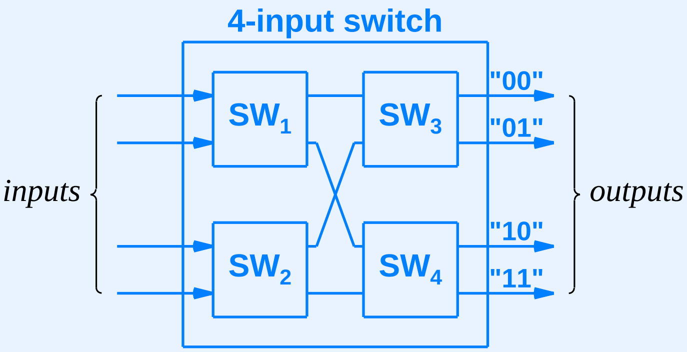
- Example of multi-stage fabric
- Features
  - Scalable
  - Self-routing
  - Packet queues allowed, but not required

Basic Banyan Building Block

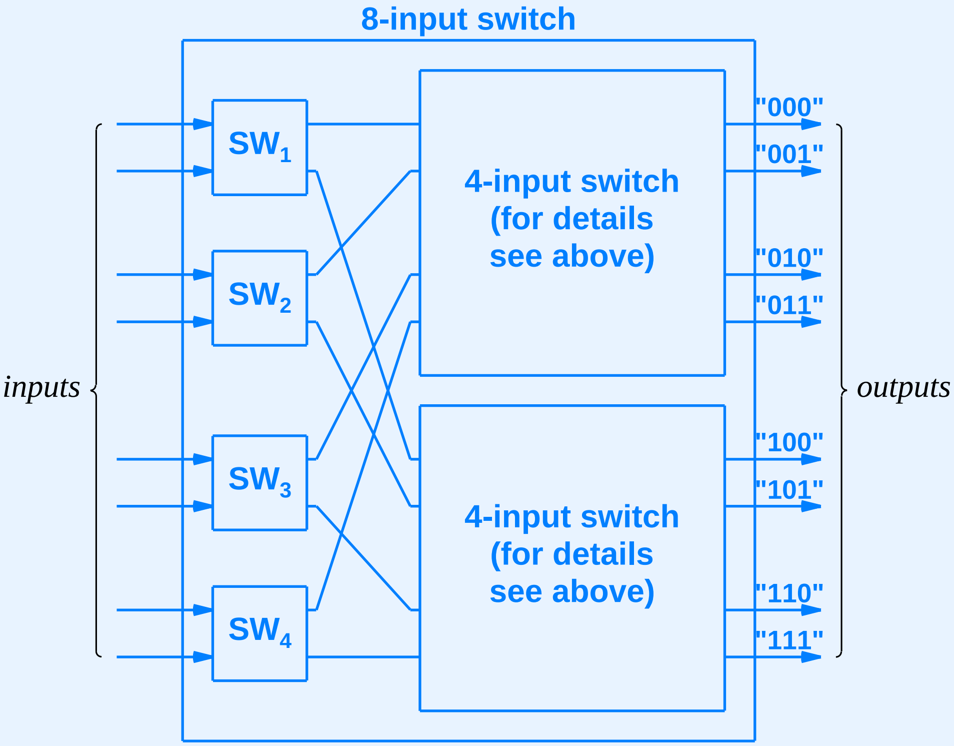


- Address added to front of each packet
- One bit of address used to select output

4-Input And 8-Input Banyan Switches



(a)



(b)

Summary

- Switching fabric provides connections inside single network system
- Two basic approaches
  - Time-division has lowest cost
  - Space-division has highest performance
- Multistage designs compromise between two
- Banyan fabric is example of multistage

Questions?

XIV

Issues In Scaling A Network Processor

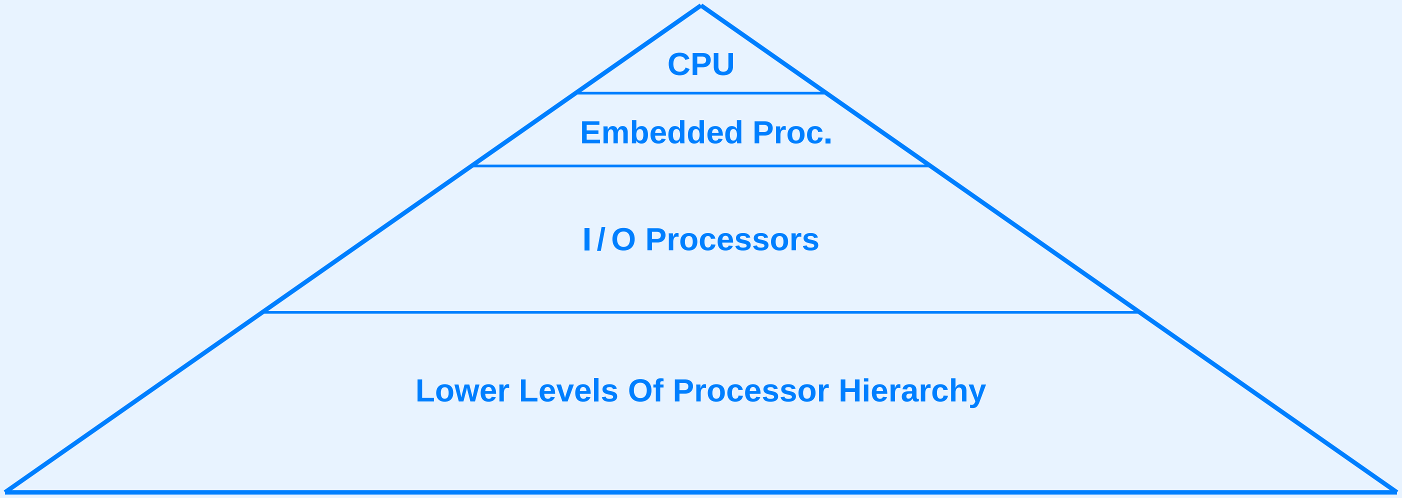
Design Questions

- Can we make network processors
  - Faster?
  - Easier to use?
  - More powerful?
  - More general?
  - Cheaper?
  - All of the above?
- Scale is fundamental

Scaling The Processor Hierarchy

- Make processors faster
- Use more concurrent threads
- Increase processor types
- Increase numbers of processors

The Pyramid Of Processor Scale



- Lower levels need the most increase

Scaling The Memory Hierarchy

- Size
- Speed
- Throughput
- Cost

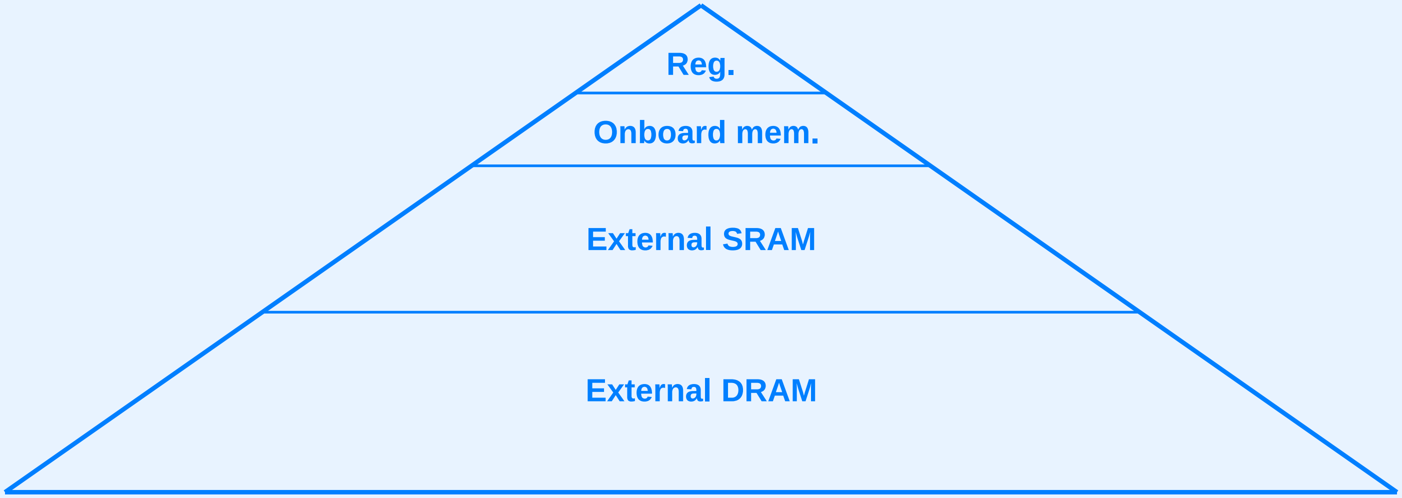
Memory Speed

- Access latency
  - Raw read/write access speed
  - SRAM 2 - 10 ns
  - DRAM 50 - 70 ns
  - External memory takes order of magnitude longer than onboard

Memory Speed
(continued)

- Memory cycle time
  - Measure of successive read/write operations
  - Important for networking because packets are large
  - Read Cycle time (tRC) is time for successive fetch operations
  - Write Cycle time (tWC) is time for successive store operations

The Pyramid Of Memory Scale



- Largest memory is least expensive

Memory Bandwidth

- General measure of throughput
- More parallelism in access path yields more throughput
- Cannot scale arbitrarily
  - Pinout limits
  - Processor must have addresses as wide as bus

Types Of Memory

Memory Technology	Abbreviation	Purpose
Synchronized DRAM	SDRAM	Synchronized with CPU for lower latency
Quad Data Rate SRAM	QDR-SRAM	Optimized for low latency and multiple access
Zero Bus Turnaround SRAM	ZBT-SRAM	Optimized for random access
Fast Cycle RAM	FCRAM	Low cycle time optimized for block transfer
Double Data Rate DRAM	DDR-DRAM	Optimized for low latency
Reduced Latency DRAM	RLDRAM	Low cycle time and low power requirements

Memory Cache

- General-purpose technique
- May not work well in network systems

Memory Cache

- General-purpose technique
- May not work well in network systems
  - Low temporal locality

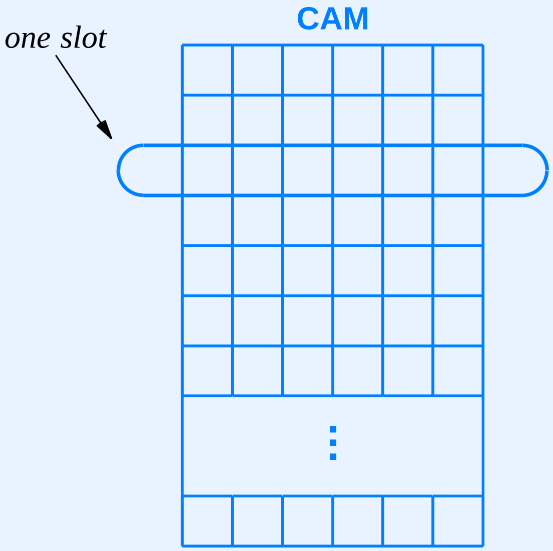
Memory Cache

- General-purpose technique
- May not work well in network systems
  - Low temporal locality
  - Large cache size (either more entries or larger granularity of access)

Content Addressable Memory (CAM)

- Combination of mechanisms
  - Random access storage
  - Exact-match pattern search
- Rapid search enabled with parallel hardware

Arrangement Of CAM



- Organized as array of slots

Lookup In Conventional CAM

- Given
  - Pattern for which to search
  - Known as *key*
- CAM returns
  - First slot that matches key, or
  - All slots that match key

Ternary CAM (T-CAM)

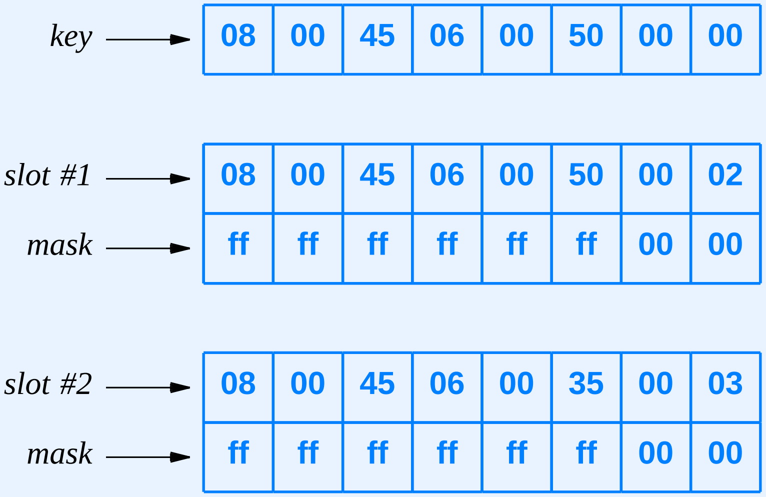
- Allows masking of entries
- Good for network processor

T-CAM Lookup

- Each slot has bit mask
- Hardware uses mask to decide which bits to test
- Algorithm

```
for each slot do  {
        if ( ( key & mask ) == ( slot & mask ) ) {
                declare key matches slot;
        }  else  {
                declare key does not match slot;
        }
}
```

Partial Matching With A T-CAM



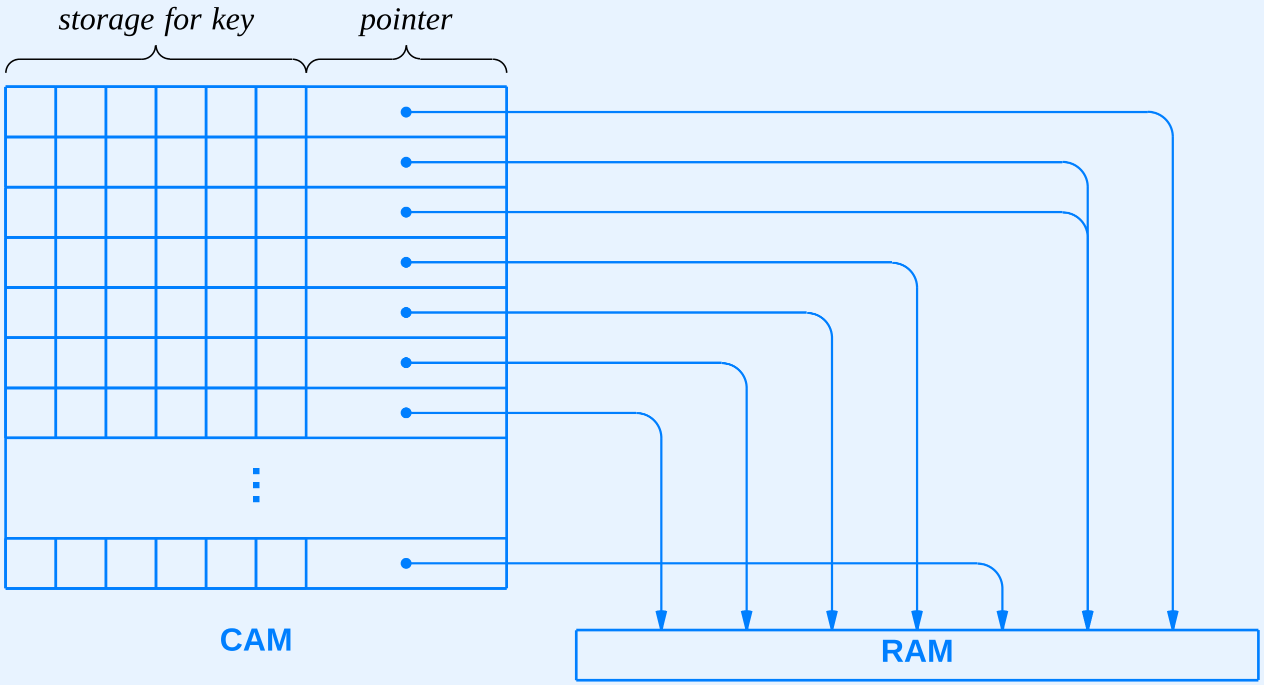
key	08	00	45	06	00	50	00	00
slot #1	08	00	45	06	00	50	00	02
mask	ff	ff	ff	ff	ff	ff	00	00
slot #2	08	00	45	06	00	35	00	03
mask	ff	ff	ff	ff	ff	ff	00	00

- Key matches slot #1

Using A T-CAM For Classification

- Extract values from fields in headers
- Form values in contiguous string
- Use a key for T-CAM lookup
- Store classification in slot

Classification Using A T-CAM



Software Scalability

- Not always easy
- Many resource constraints
- Difficulty arises from
  - Explicit parallelism
  - Code optimized by hand
  - Pipelines on heterogeneous hardware

Summary

- Scalability key issue
- Primary subsystems affecting scale
  - Processor hierarchy
  - Memory hierarchy
- Many memory types available
  - SRAM
  - SDRAM
  - CAM
- T-CAM useful for classification

Questions?

XV

Examples Of Commercial Network Processors

Commercial Products

- Emerge in late 1990s
- Become popular in early 2000s
- Exceed thirty vendors by 2003
- Fewer than thirty vendors by 2004

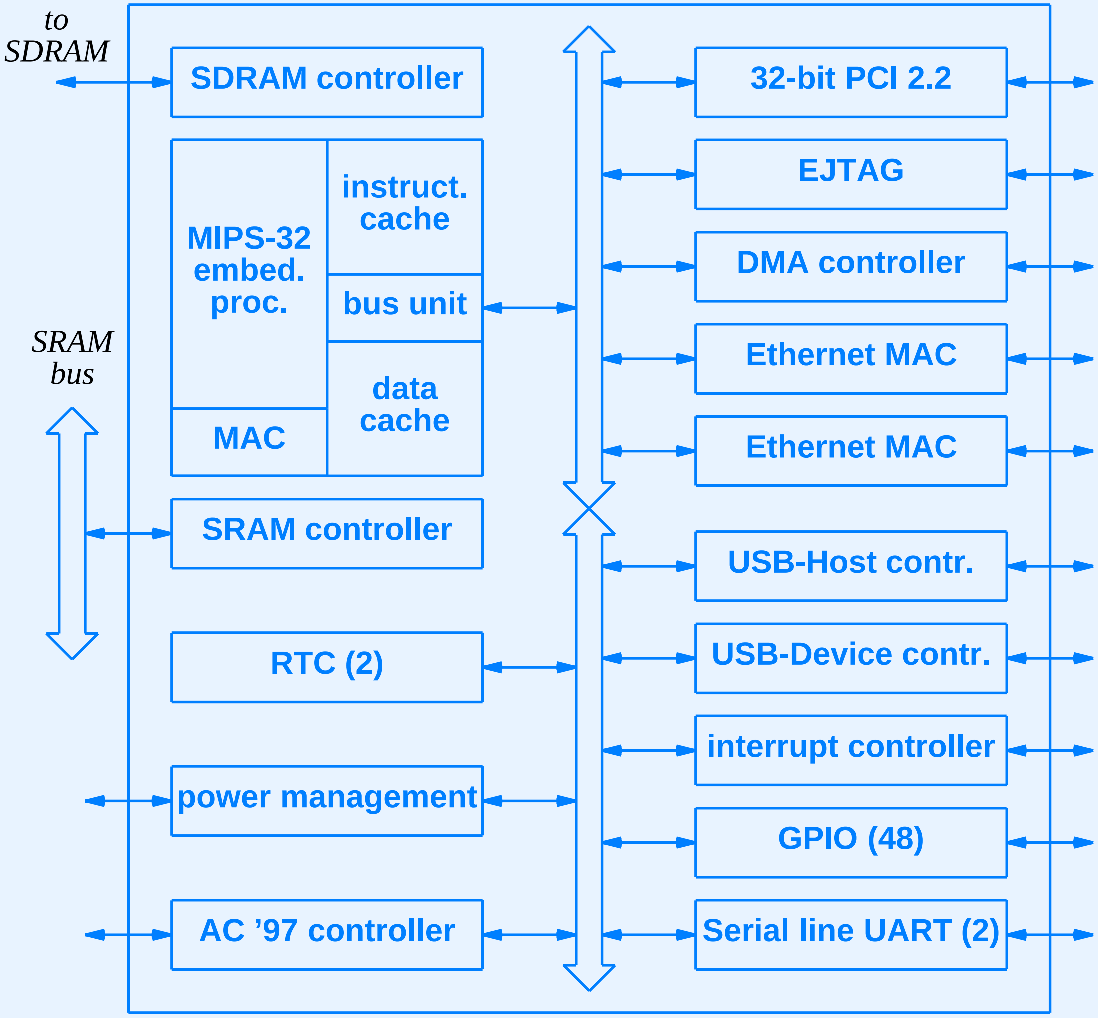
Examples

- Chosen to
  - Illustrate concepts
  - Show broad categories
  - Expose the variety
- Not necessarily ‘‘best’’
- Not meant as an endorsement of specific vendors
- Show a snapshot as of 2004

Augmented RISC (Alchemy)

- Based on MIPS-32 CPU
  - Five-stage pipeline
- Augmented for packet processing
  - Instructions (e.g. multiply-and-accumulate)
  - Memory cache
  - I/O interfaces

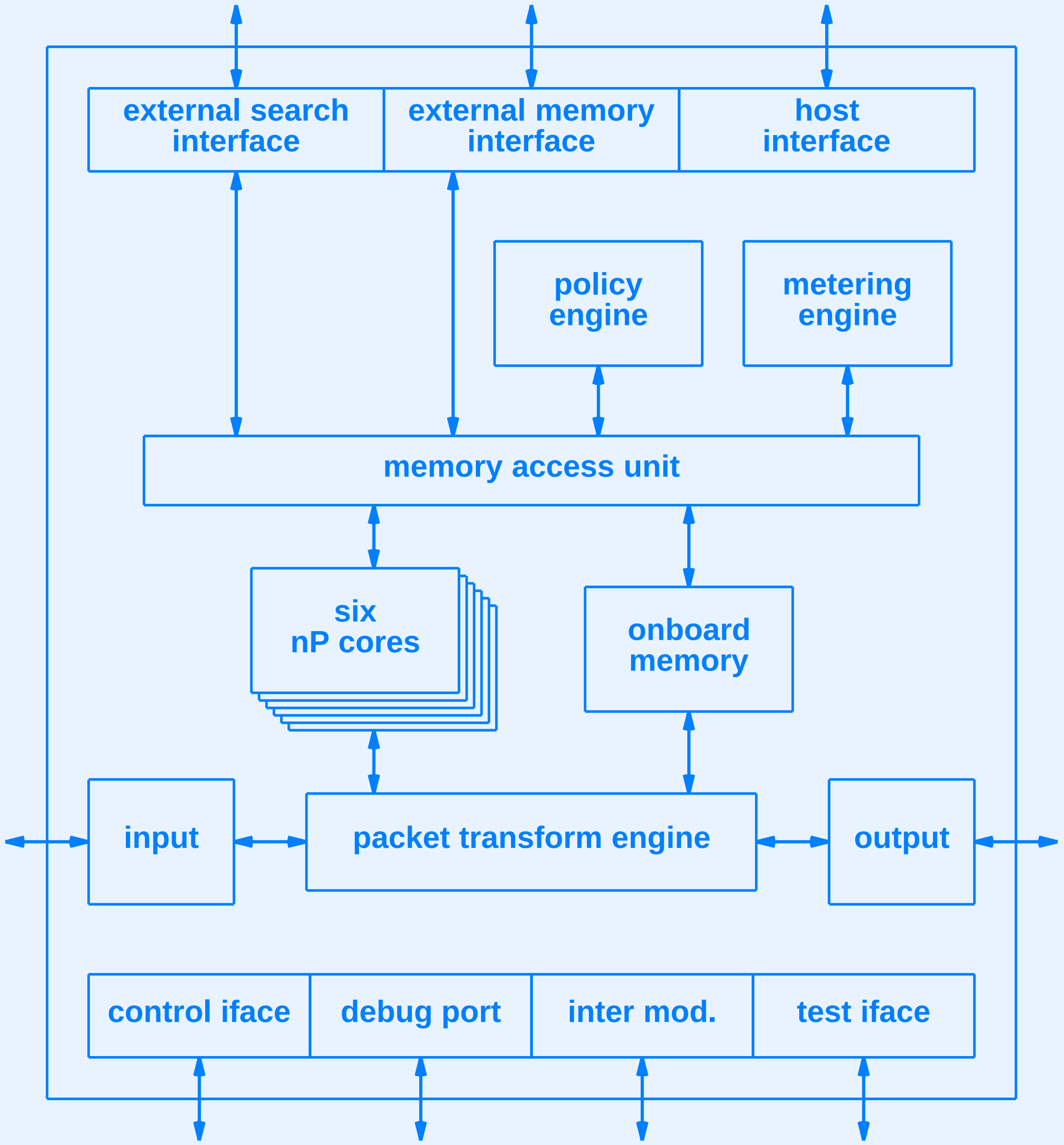
Alchemy Architecture



Parallel Embedded Processors Plus Coprocessors (AMCC)

- One to six nP core processors
- Various engines
  - Packet metering
  - Packet transform
  - Packet policy

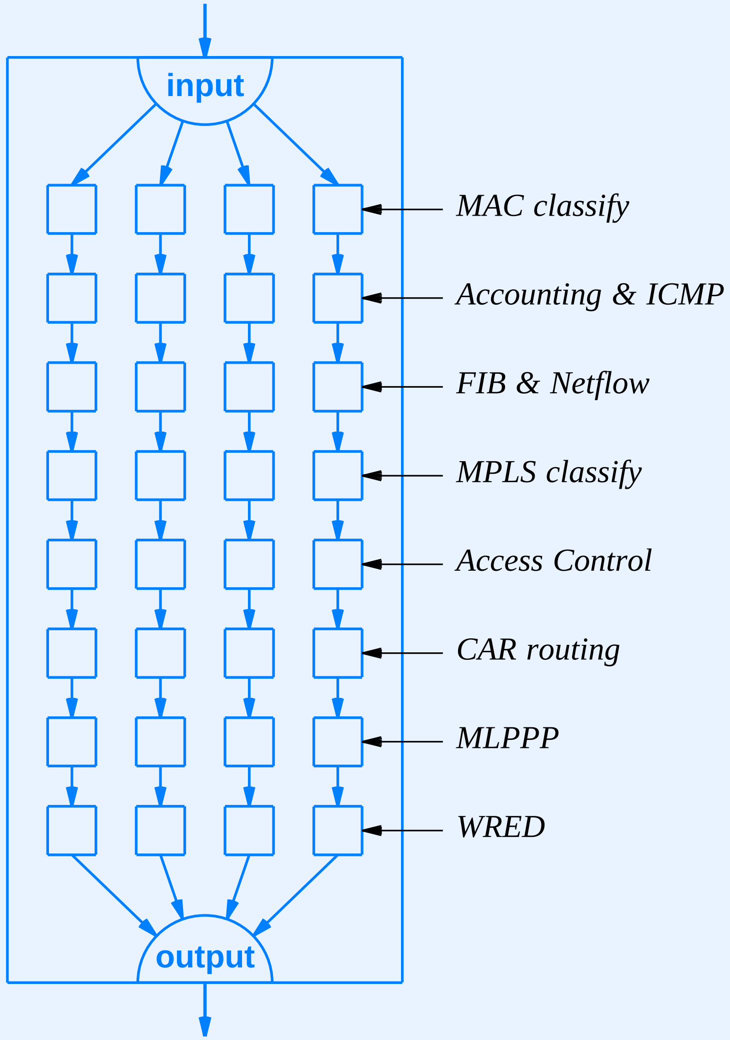
AMCC Architecture



Parallel Pipelines Of Homogeneous Processors (Cisco)

- Parallel eXpress Forwarding (PXF)
- Arranged in parallel pipelines
- Packet flows through one pipeline
- Each processor in pipeline dedicated to one task

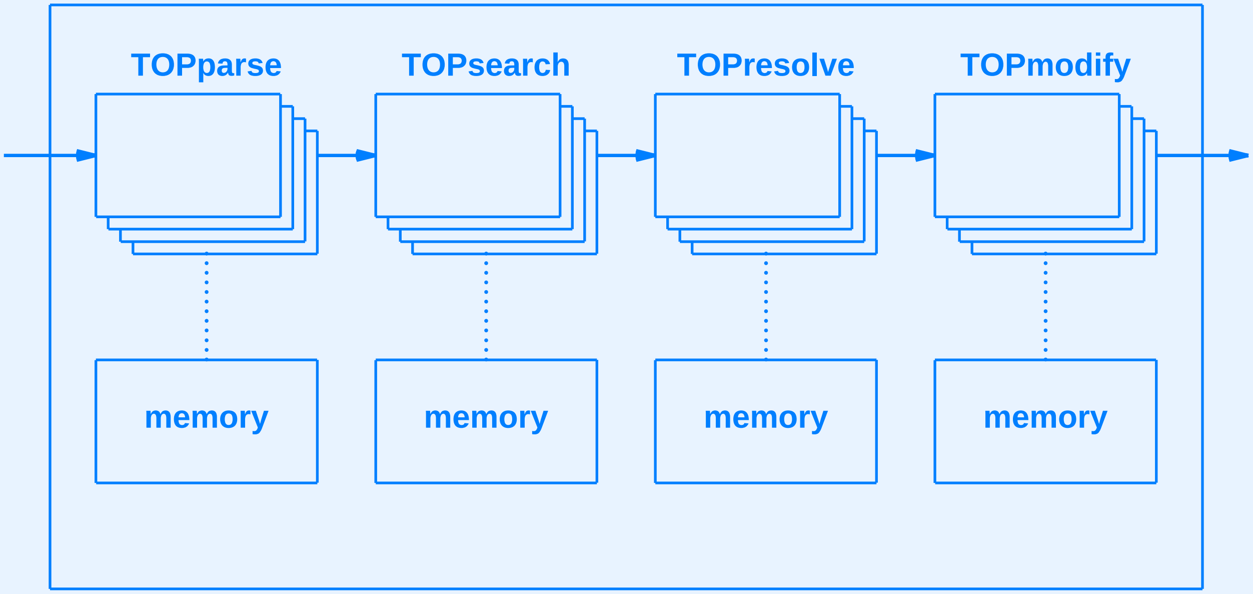
Cisco Architecture



Pipeline Of Parallel Heterogeneous Processors (EZchip)

- Four processor types
- Each type optimized for specific task

EZchip NP-1c Architecture



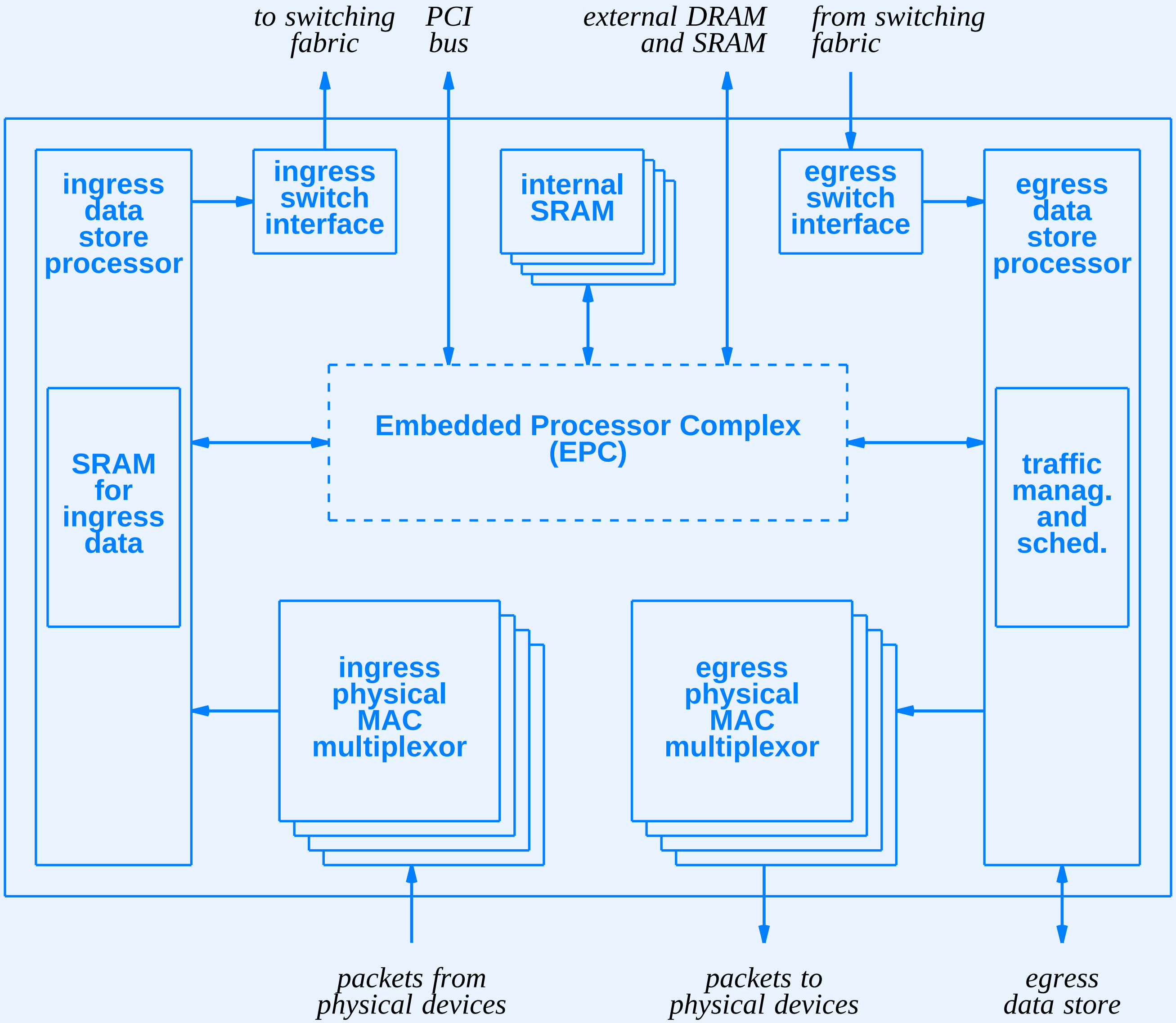
EZchip Processor Types

Processor Type	Optimized For
TOPparse	Header field extraction and classification
TOPsearch	Table lookup
TOPresolve	Queue management and forwarding
TOPmodify	Packet header and content modification

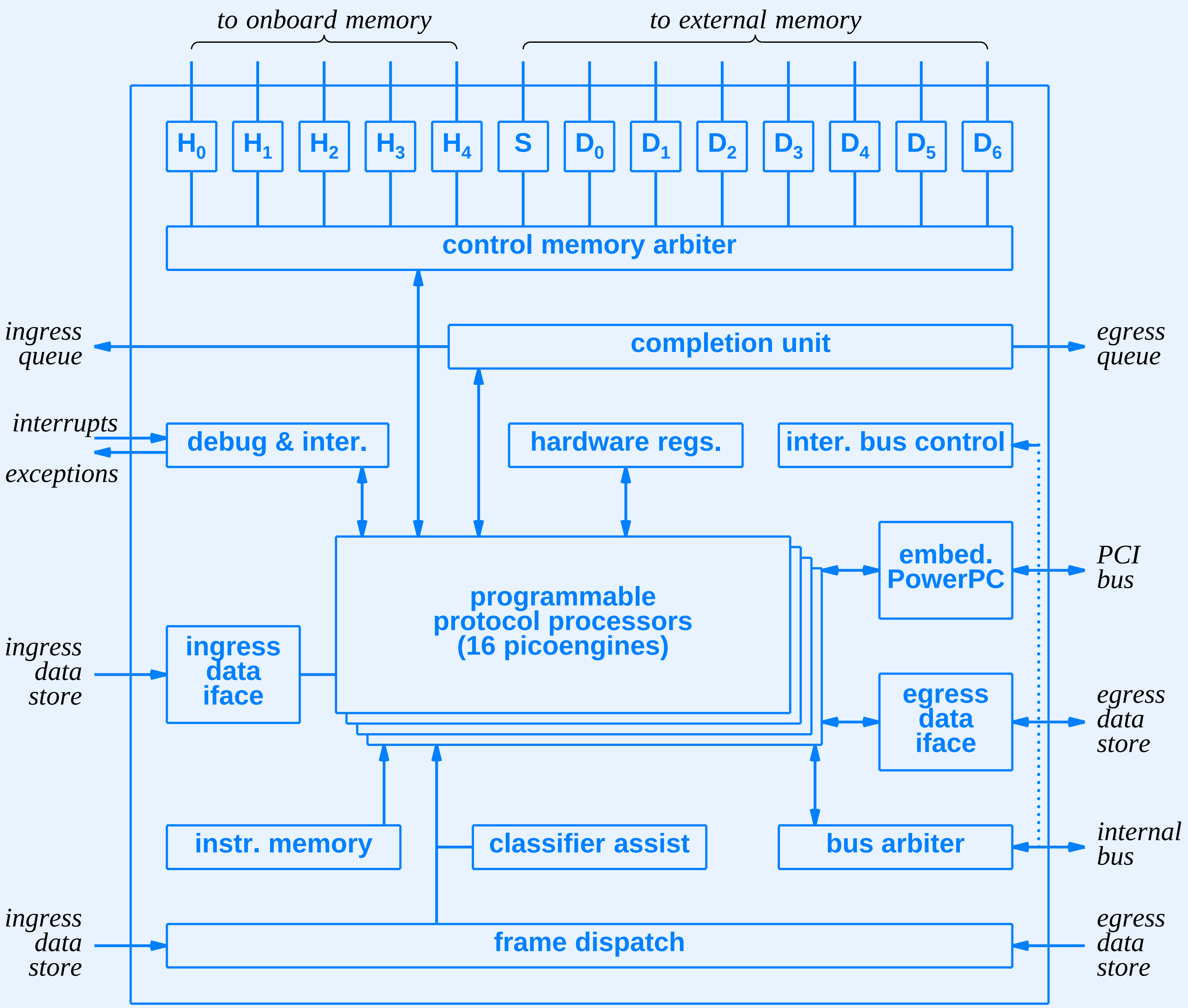
Extensive And Diverse Processors (Hifn, formerly IBM)

- Multiple processor types
- Extensive use of parallelism
- Separate ingress and egress processing paths
- Multiple onboard data stores
- Model is *NP4GS3*

Hifn NP4GS3 Architecture


to switching fabric
PCI bus
external DRAM and SRAM
from switching fabric
ingress data store processor
ingress switch interface
internal SRAM
egress switch interface
egress data store processor
SRAM for ingress data
Embedded Processor Complex (EPC)
traffic manag. and sched.
ingress physical MAC multiplexor
egress physical MAC multiplexor
packets from physical devices
packets to physical devices
egress data store

Hifn's Embedded Processor Complex



Packet Engines

- Found in Embedded Processor Complex
- Programmable
- Handle many packet processing tasks
- Operate in parallel (sixteen)
- Known as *picoengines*

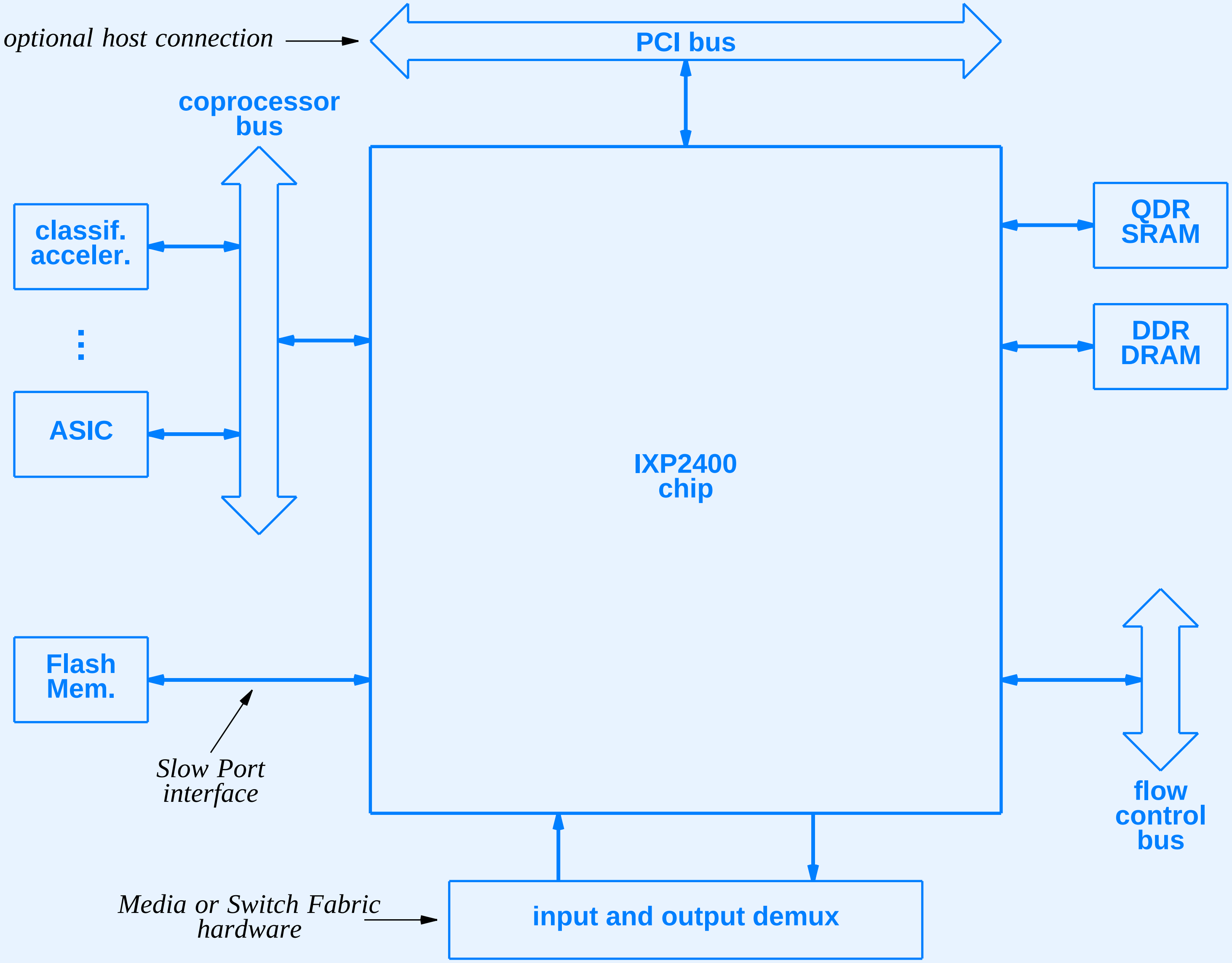
Other Processors On The IBM Chip

Coprocessor	Purpose
Data Store	Provides frame buffer DMA
Checksum	Calculates or verifies header checksums
Enqueue	Passes outgoing frames to switch or target queues
Interface	Provides access to internal registers and memory
String Copy	Transfers internal bulk data at high speed
Counter	Updates counters used in protocol processing
Policy	Manages traffic
Semaphore	Coordinates and synchronizes threads

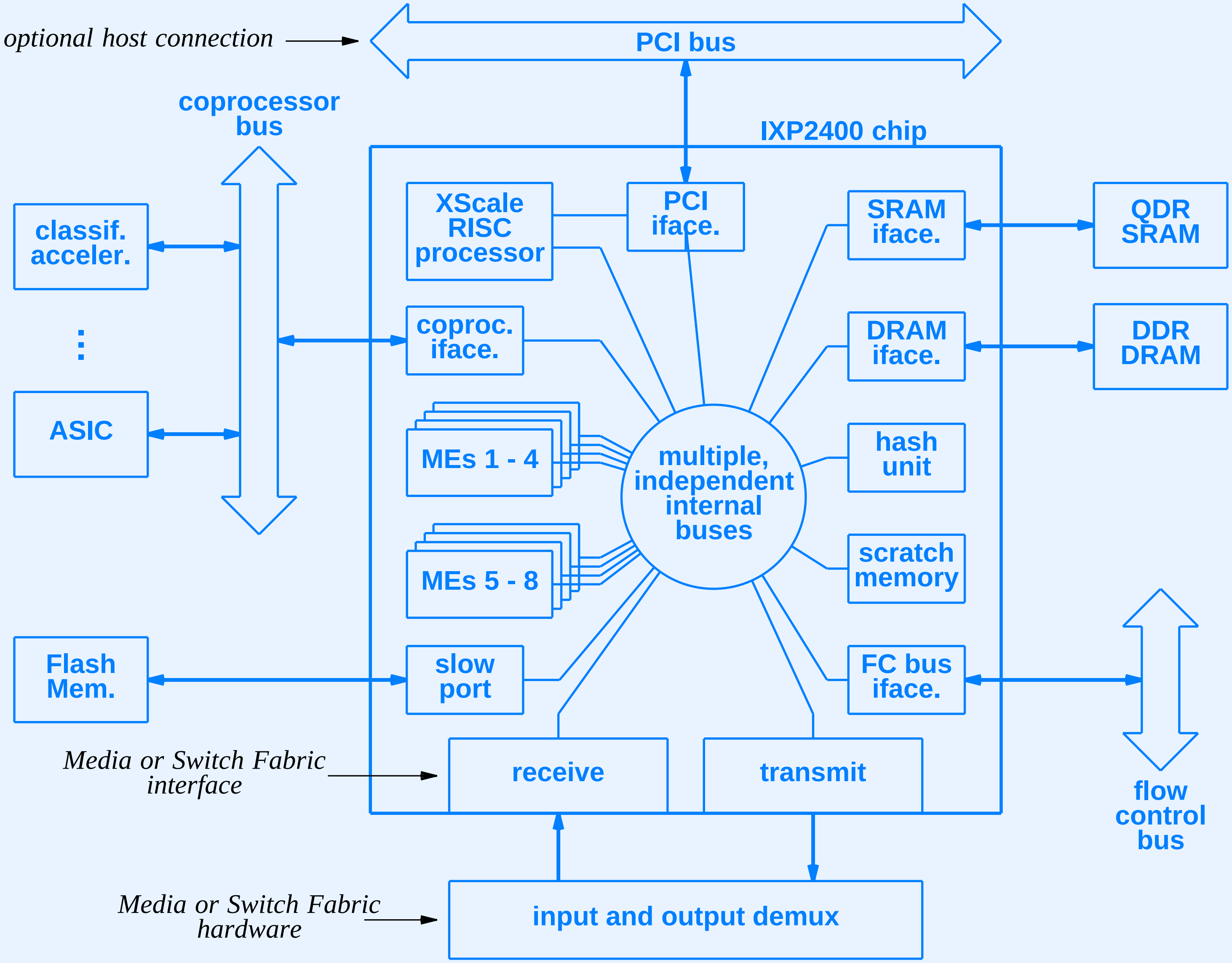
Homogeneous Parallel Processors Plus Controller (Intel IXP2xxx)

- Two basic models
  - IXP2400
  - IXP2800
- Eight or sixteen parallel programmable packet processors known as *microengines*
- One XScale embedded RISC processor
- High-speed Media and Switch Fabric interface
- Connections to external buses (e.g., for memory)

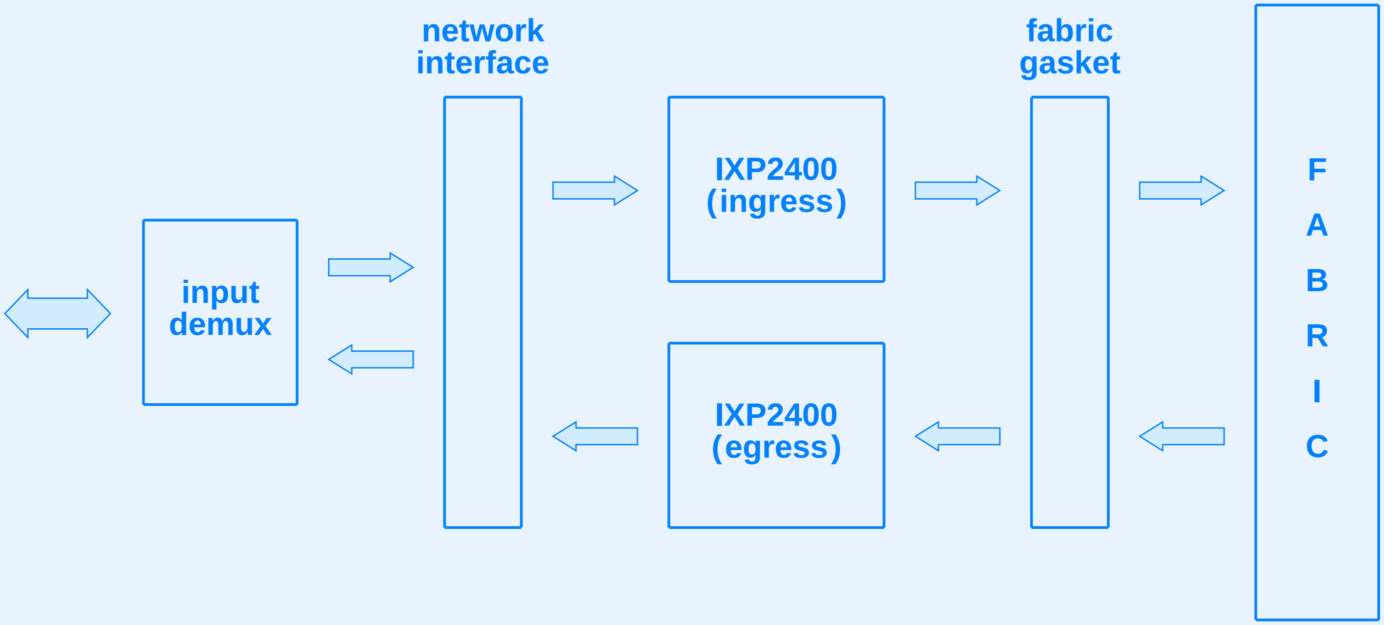
Intel Chip External Connections



Intel Chip Internal Architecture



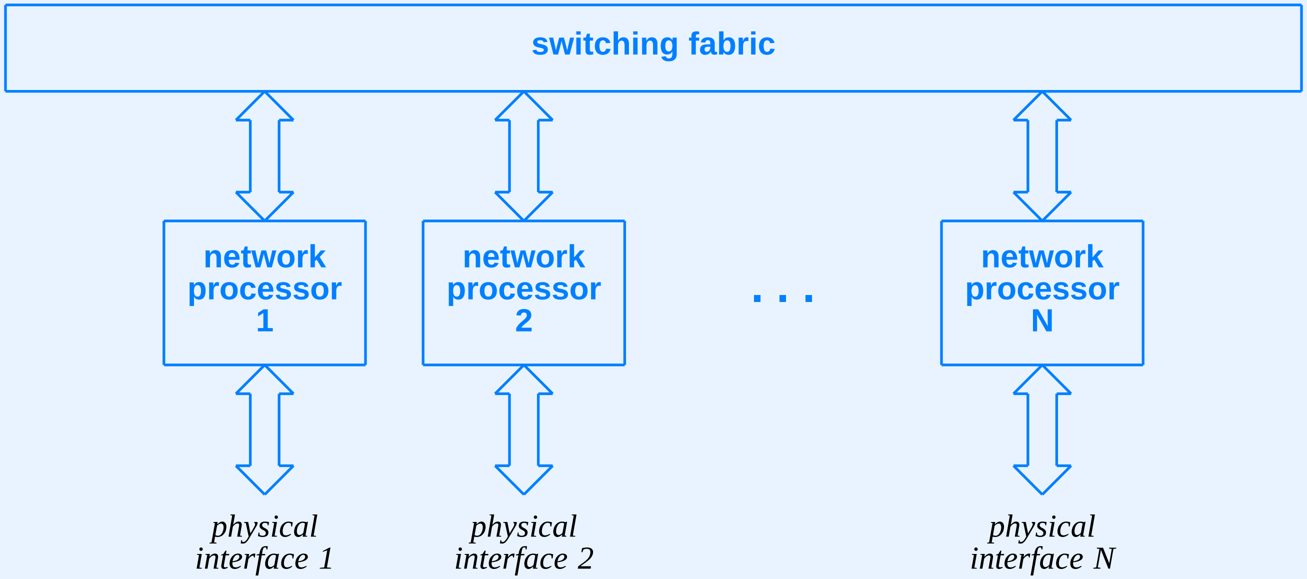
Two Intel Chips Used For High Speed



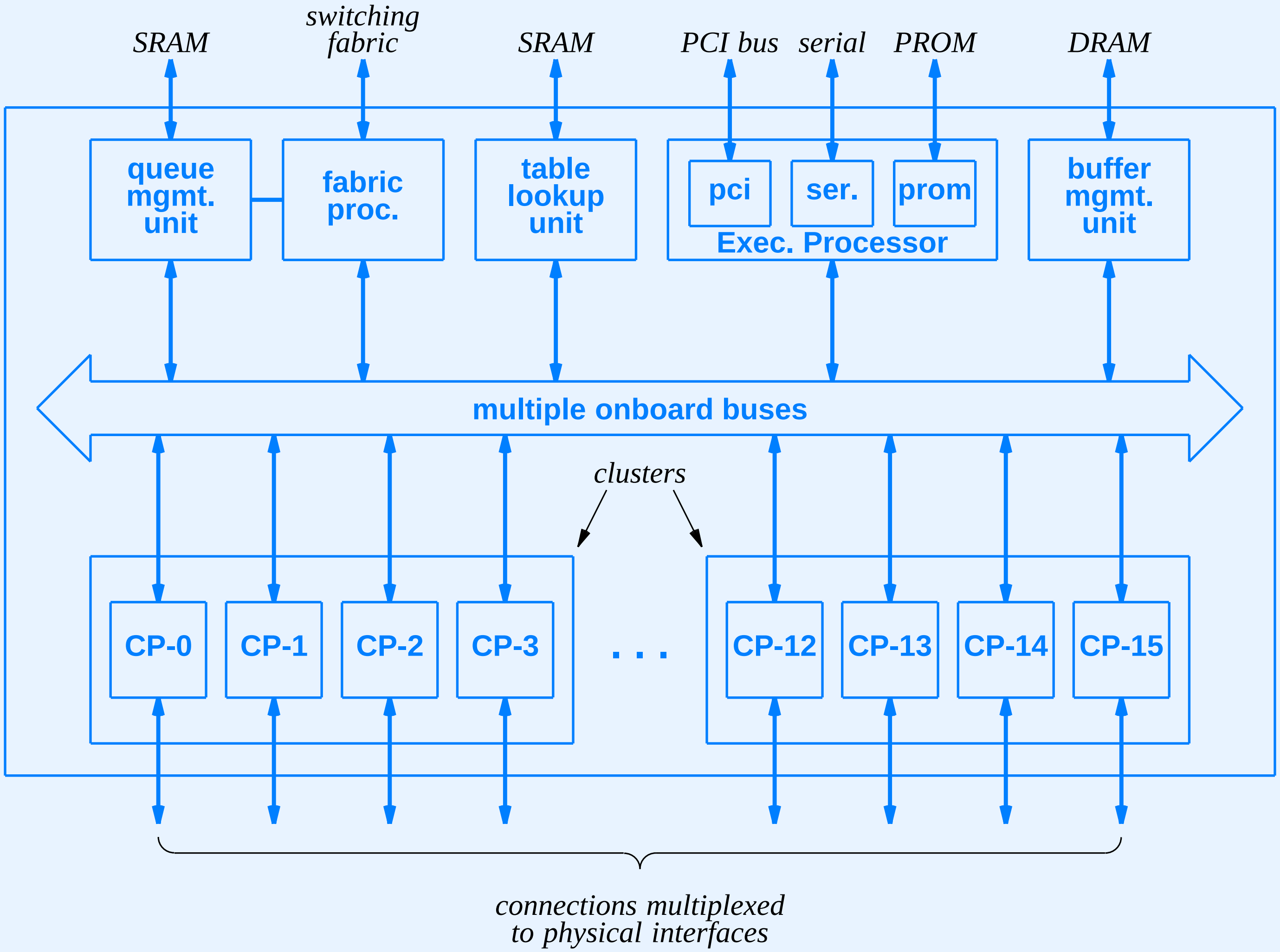
Flexible RISC Plus Coprocessors (Motorola C-PORT)

- Onboard processors can be
  - Dedicated
  - Parallel clusters
  - Pipeline

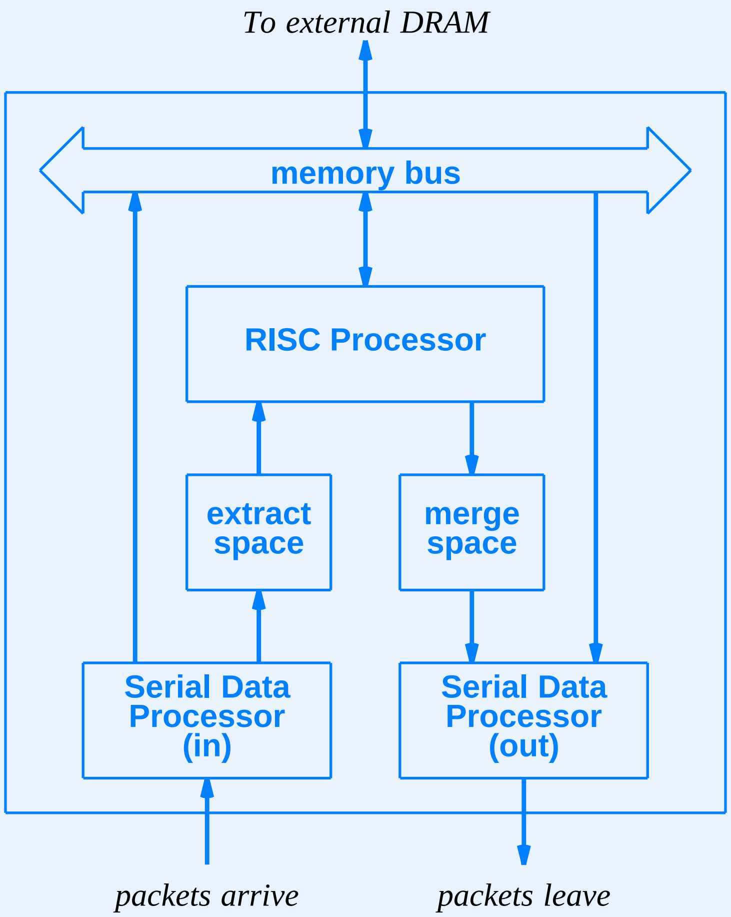
C-Port Architecture



Internal Structure Of A C-Port Channel Processor



Channel Processor Architectuer

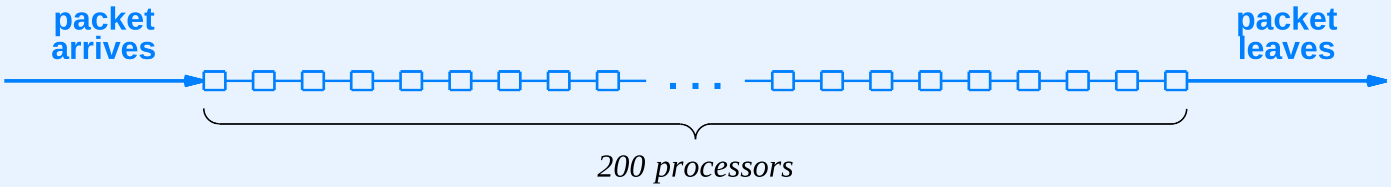


- Actually a processor complex

Extremely Long Pipeline (Xelerated)

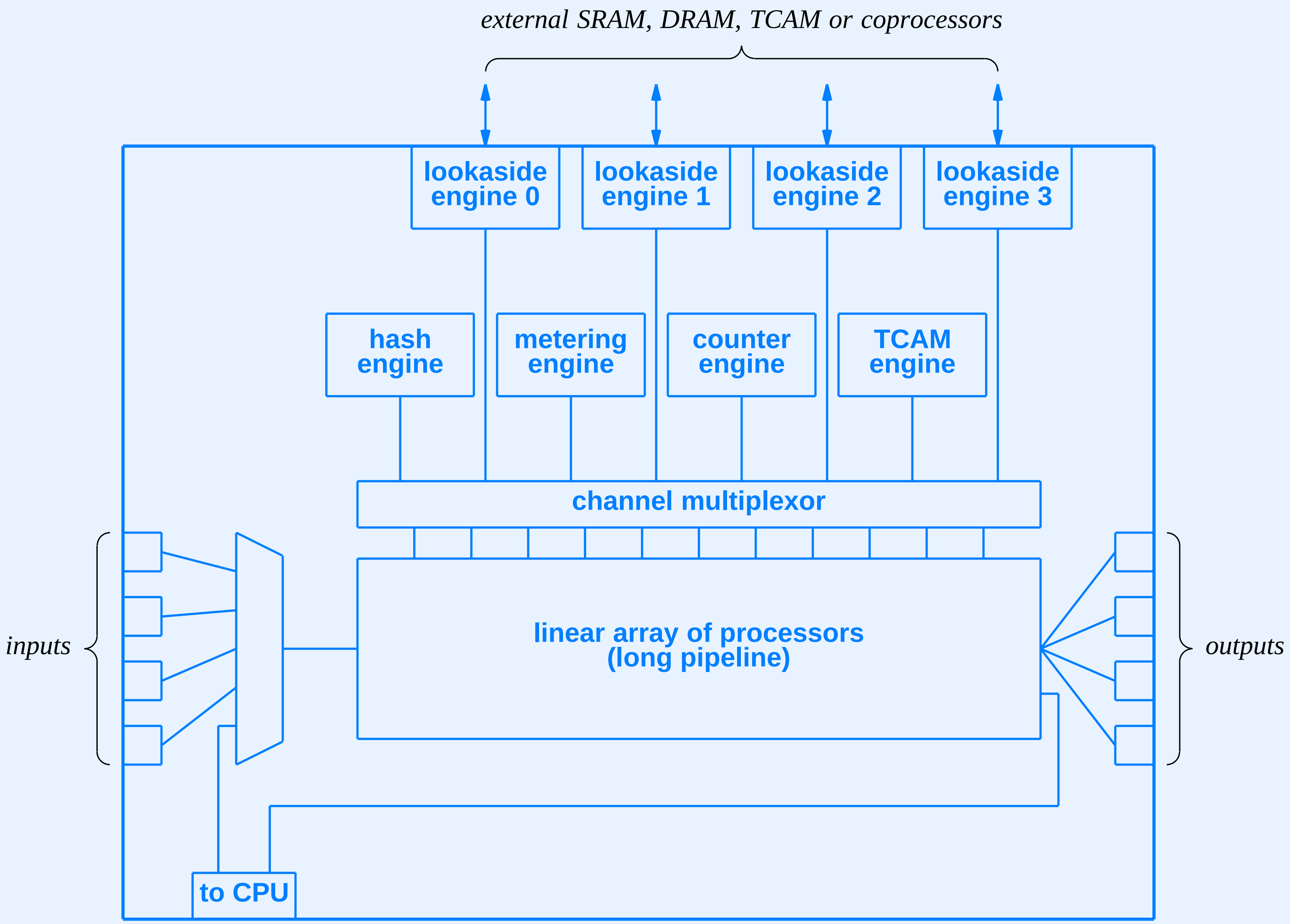
- Pipeline contains 200 processors
- Each processor can execute four instructions per packet
- External coprocessor calls used to pass state

Xelerated Architecture



- Pipeline has 200 stages
- Four instructions per packet per stage

Xelerated Internal Architecture



Summary

- Many network processor architecture variations
- Examples include
  - Augmented RISC processor
  - Embedded parallel processors plus coprocessors
  - Parallel pipelines of homogeneous processors
  - Pipeline of parallel heterogeneous processors
  - Extensive and diverse processors
  - Flexible RISC plus coprocessors
  - Extremely long pipeline

Questions?

STOP