CSC 2229 – Software-Defined Networking

Handout # 7: Data Plane and Packet Forwarding

Professor Yashar Ganjali
Department of Computer Science
University of Toronto

yganjali@cs.toronto.edu
http://www.cs.toronto.edu/~yganjali
Announcements

- In class presentations
  - Starting next week
  - Papers will be sent by email, today
  - If you don’t hear from me, please contact me ASAP.

- Final project
  - Feedback on projects today
Data Plane Functionalities

- So far we have focused on control plane
  - It distinguishes SDN from traditional networks
  - Source of many (perceived) challenges ...
  - ... and opportunities

- What about the data plane?
  - Which features should be provided in the data plane?

- What defines the boundary between control and data planes?
OpenFlow and Data Plane

- Historically, OpenFlow defined data plane’s role as *forwarding*.
  - Other functions are left to middleboxes and edge nodes in this model.

**Question.** Why only forwarding?
- Other data path functionalities exist in today’s routers/switches too.
- What distinguishes forwarding from other functions?
Adding New Functionalities

- Consider a new functionality we want to implement in a software-defined network.
- Where is the best place to add that function?
  - Controller?
  - Switches?
  - Middleboxes?
  - Endhosts?
Adding New Functionalities

- What metrics/criteria do we have to decide where new functionalities should be implemented?

- **Example**: Elephant flow detection
  - **Data plane**: change forwarding elements (DevoFlow)
  - **Control plane**: push functionality close to the path (Kandoo)

- What does the development process look like?
  - Changing ASIC expensive and time-consuming
  - Changing software fast
Alternative View

- Decouple based on development cycle
  - Fast: mostly software
  - Slow: mostly hardware

- Development process
  - Start in software
  - As function matures, move towards hardware

- Not a new idea
  - Model has been used for a long-time in industry.
Changing the control plane

I can tailor my network to meet my needs ...

1. Quickly deploy new protocols.
2. Monitor precisely what my forwarding plane is doing.
3. Fold expensive middlebox functions into the network, for free.
4. Try out beautiful new ideas. Tailor my network to meet my needs.
5. Differentiate. Now I own my intellectual property.
But wait a minute...
New Switch OS Driver

OSPF  BGP  New  etc.
When you need a new feature...

1. Equipment vendor can’t just send you a software upgrade
2. New forwarding features take years to develop
3. By then, you’ve figured out a kludge to work around it
4. Your network gets more complicated, more brittle
5. Eventually, when the upgrade is available, it either
   - No longer solves your problem, or
   - You need a fork-lift upgrade, at huge expense.
Network systems are built “bottoms-up”

“This is how I process packets ...”

Switch OS

Driver

Fixed-function switch
“Programmable switches are 10-100x slower than fixed-function switches”

Conventional wisdom in networking
Network systems are starting to be programmed “top-down”

“This is precisely how you must process packets”

```
# Switch OS
Driver
Programmable Switch
```

```
table int_table { 
reads { 
 ipv4.proto; 
} 
actions { 
export_queue_latency; 
} }

action export_queue_latency (ev_id) { 
add_header(int_header); 
modify_field(int_header.kind, TCP_OPTION_INT); 
modify_field(int_header.len, TCP_OPTION_INT_LEN); 
modify_field(int_header.fragment, 0x0); 
modify_field(int_header.ttl, 0x0); 
modify_field(int_header withhold, 0x0); 
add_to_field(tcp.dataOffset, 2); 
add_to_field(tcp.total_len, 0); 
subtract_from_field(tcp.urgent, 12); 
}
```
Why are programmable forwarding planes happening now?
Domain Specific Processors

Computers
- Java Compiler
- OpenCL Compiler
- Matlab Compiler
- TensorFlow Compiler

Graphics Processing
- GPU

Signal Processing
- DSP

Machine Learning
- TPU

Networking
- Language Compiler

?
Domain Specific Processors

- Computers: Java Compiler
- Graphics: OpenCL Compiler
- Signal Processing: Matlab Compiler
- Machine Learning: TensorFlow Compiler
- Networking: P4 Compiler

CPU
GPU
DSP
TPU
PISA
PISA: Protocol Independent Switch Architecture
PISA: Protocol Independent Switch Architecture
Example P4 Program

Parser Program

```p4
parser parse_ethernet {
  extract(ethernet);
  return switch(ethernet.ethertype) {
    0x8100 : parse_vlan_tag;
    0x0800 : parse_ipv4;
    0x8847 : parse_mpls;
    default: ingress;
  }
}
```

Header and Data Declarations

```p4
header_type ethernet_t { ... }
header_type l2_metadata_t { ... }
header ethernet_t ethernet;
header vlan_tag_t vlan_tag[2];
metadata l2_metadata_t l2_meta;
```

Tables and Control Flow

```p4
table port_table { ... }
control ingress {
  apply(port_table);
  if (l2_meta.vlan_tags == 0) {
    process_assign_vlan();
  }
}
```
The world’s fastest switch in production. Forwarding defined in software (P4). Programs always run at line-rate.

Same power. Same cost.
How programmability is being used

1. Reducing complexity
# Reducing complexity

## Switch OS

### switch.p4

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Unicast Routing</td>
<td>- IPv4 and IPv6 Routing &amp; Switching</td>
<td>- Storm Control, IP Source Guard</td>
<td>- Storm Control, IP Source Guard</td>
<td></td>
<td>- Route Table Entry Counters</td>
<td>- Forwarding, QoS</td>
<td></td>
</tr>
<tr>
<td>- Routed Ports &amp; SVI</td>
<td>- IPv in IP (Gin4, Gin6)</td>
<td></td>
<td></td>
<td></td>
<td>- VLAN/Bridge Domain Counters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- VRF</td>
<td>- VXLAN, NVGRE, GENEVE &amp; GRE</td>
<td></td>
<td></td>
<td></td>
<td>- Port/Interface Counters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Unicast RPF</td>
<td>- Segment Routing, ILM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Strict and Loose</td>
<td>- IPv4 and IPv6 Routing &amp; Switching</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Multicast</td>
<td>- IP in IP (Gin4, Gin6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- PIM-SM/DM &amp; PIM-Bidir</td>
<td>- VXLAN, NVGRE, GENEVE &amp; GRE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Ethernet switching

- VLAN Flooding
  - MAC Learning & Aging
  - STP state
  - VLAN Translation

### Load balancing

- LAG
  - ECMP & WCMP
  - Resilient Hashing
  - Flowlet Switching

### Fast Failover

- LAG & ECMP

### Load balancing

- LAG
  - ECMP & WCMP
  - Resilient Hashing
  - Flowlet Switching

### Fast Failover

- LAG & ECMP
Reducing complexity

Compiler

My switch.p4

Switch OS

Driver

Programmable Switch
How programmability is being used

2 Adding new features
Protocol complexity 20 years ago

- Ethernet
- IPv4
- IPX

Diagram:
- Ethernet
  - ethtype
  - IPv4
  - ethtype
  - IPX
Adding features: Some examples so far

1. New encapsulations and tunnels

2. New ways to tag packets for special treatment

3. New approaches to routing: *e.g.* source routing in MSDCs

4. New approaches to congestion control

5. New ways to process packets: *e.g.* processing ticker-symbols
New applications: Some examples so far

1. Layer-4 Load Balancer\(^1\)
   - Replace 100 servers or 10 dedicated boxes with one programmable switch
   - Track and maintain mapping for 5-10 million http flows

2. Fast stateless firewall
   - Add/delete and track 100s of thousands of new connections per second

3. Cache for Key-value store\(^2\)
   - Memcache in-network cache for 100 servers
   - 1-2 billion operations per second

---

How programmability is being used

3 Network telemetry
“Which path did my packet take?”

“I visited Switch 1 @780ns, Switch 9 @1.3µs, Switch 12 @2.4µs”

“Which rules did my packet follow?”

“In Switch 1, I followed rules 75 and 250. In Switch 9, I followed rules 3 and 80.”

<table>
<thead>
<tr>
<th>#</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>192.168.0/24</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
“How long did my packet queue at each switch?”

“Delay: 100ns, 200ns, 19740ns”

“Who did my packet share the queue with?”
3. "How long did my packet queue at each switch?"

"Delay: 100ns, 200ns, 19740ns"

4. "Who did my packet share the queue with?"
We’d like the network to answer these questions

1. “Which path did my packet take?”
2. “Which rules did my packet follow?”
3. “How long did it queue at each switch?”
4. “Who did it share the queues with?”

A PISA device programmed using P4 can answer all four questions at line rate, for the first time. Without generating additional packets.
INT: Inband Network Telemetry

Add: SwitchID, Arrival Time, Queue Delay, Matched Rules, ...
P4 code snippet: Insert switch ID, ingress and egress timestamp

```p4
/* INT: add switch id */
action int_set_header_0() {
    add_header(int_switch_id_header);
    modify_field(int_switch_id_header.switch_id,
                 global_config_Metadata.switch_id);
}

/* INT: add ingress timestamp */
action int_set_header_1() {
    add_header(int_ingress_tstamp_header);
    modify_field(int_ingress_tstamp_header.ingress_tstamp,
                 i2e_metadata.ingress_tstamp);
}

/* INT: add egress timestamp */
action int_set_header_2() {
    add_header(int_egress_tstamp_header);
    modify_field(int_egress_tstamp_header.egress_tstamp,
                 eg_intr_md_from_parser_aux.egress_global_tstamp);
}
```
How programmability is being used

1. Reducing complexity
2. Adding new features
3. Network telemetry
In summary...

1. SDN is about who is in charge!
   1. Network owners and operators took charge of how their networks are controlled.
   2. They also decide how packets are processed.

2. Chip technology: Programmable forwarding now has the same power, performance and cost as fixed function.

3. New ideas: Beautiful new ideas now owned by the programmer, not the chip designer.
Back to Packet Forwarding in Data Plane ...
Most important functionality of the data plane: *packet forwarding*

- Packet forwarding
  - Receive packet on ingress lines
  - Transfer to the appropriate egress line
  - Transmit the packet
Output-Queued Switching

1. **Header Processing**
   - Lookup IP Address
   - Update Header
   - Address Table

2. **Queue Packet**
   - Buffer Memory

3. **Data**
   - Hdr

4. **Queue Packet**
   - Buffer Memory

5. **N times line rate**
Properties of OQ Switches

• Require speedup of N
  • Crossbar fabric N times faster than the line rate

• Work conserving
  • If there is a packet for a specific output port, link will not stay idle

• Ideal in theory
  • Achieves 100% throughput
  • Very hard to build in practice
Input-Queued Switching

Header Processing:
- Lookup IP Address
- Update Header
- Address Table

Queue Packet
- Data
- Hdr

Scheduler

Data
- Hdr

Packet Buffer
- Memory

N

1

2

N

1
Properties of Input-Queued Switches

- Need for a scheduler
  - Choose which input port to connect to which output port during each time slot

- No speedup necessary
  - But sometimes helpful to ensure 100% throughput

- Throughput might be limited due to poor scheduling
Head-of-Line Blocking

The switch is NOT work-conserving!
VOQs: How Packets Move

VOQs
HoL Blocking vs. OQ Switch

IQ switch with HoL blocking

OQ switch

\[2 - \sqrt{2} \approx 58\%\]
Scheduling in Input-Queued Switches

- **Input:** the scheduler can use ...
  - Average rate
  - Queue sizes
  - Packet delays

- **Output:** which ingress port to connect to which egress port
  - A matching between input and output ports

- **Goal:** achieve 100% throughput
Common Definitions of 100% throughput

Work-conserving

For all \( n,i,j \), i.e.,
\[
Q_{ij}(n) < C,
\]
\[
\sup_{n,i,j} Q_{ij}(n) < C
\]

For all \( n,i,j \), i.e.,
\[
E[Q_{ij}(n)] < C
\]
\[
\sup_{n,i,j} E[Q_{ij}(n)] < C
\]

Departure rate = arrival rate, i.e.,
\[
\lim_{n \to \infty} \frac{D_{ij}(n)}{n} = \lim_{n \to \infty} \frac{A_{ij}(n)}{n} = \lambda_{ij}
\]

Usually we focus on this definition.
Scheduling in Input-Queued Switches

Uniform traffic
- Uniform cyclic
- Random permutation
- Wait-until-full
- Non-uniform traffic, known traffic matrix
  - Birkhoff-von-Neumann
- Unknown traffic matrix
  - Maximum Size Matching
  - Maximum Weight Matching
100% Throughput for Uniform Traffic

- Nearly all algorithms in literature can give 100% throughput when traffic is uniform.
- For example:
  - Uniform cyclic.
  - Random permutation.
  - Wait-until-full [simulations].
  - Maximum size matching (MSM) [simulations].
  - Maximal size matching (e.g. WFA, PIM, iSLIP) [simulations].
Uniform Cyclic Scheduling

Each \((i, j)\) pair is served every \(N\) time slots: Geom/D/1

\[
\lambda = \rho / N < 1 / N
\]

Stable for \(\rho < 1\)
Wait Until Full

- We don’t have to do much at all to achieve 100% throughput when arrivals are Bernoulli IID uniform.
- Simulation suggests that the following algorithm leads to 100% throughput.

- Wait-until-full:
  - If any VOQ is empty, do nothing (i.e. serve no queues).
  - If no VOQ is empty, pick a random permutation.
Simple Algorithms with 100% Throughput

Maximal Matching Algorithm (iSLIP)

Uniform Cyclic

MSM

Wait until full
Outline

• Uniform traffic
  • Uniform cyclic
  • Random permutation
  • Wait-until-full

Non-uniform traffic, known traffic matrix
  • Birkhoff-von-Neumann

• Unknown traffic matrix
  • Maximum Size Matching
  • Maximum Weight Matching
Non-Uniform Traffic

- Assume the traffic matrix is:

\[ \Lambda = \begin{pmatrix}
0.26 & 0.57 & 0.08 & 0 & 0.91 \\
0 & 0 & 0.4 & 0.45 & 0.85 \\
0.08 & 0.24 & 0.35 & 0.15 & 0.82 \\
0.56 & 0.09 & 0 & 0.28 & 0.93 \\
0.9 & 0.9 & 0.83 & 0.88 &
\end{pmatrix} \]

- \( \Lambda \) is admissible (rates < 1) and non-uniform
Uniform Schedule?

- Uniform schedule cannot work

- Each VOQ serviced at rate $\mu = \frac{1}{N} = \frac{1}{4}$

- Arrival rate > departure rate $\Rightarrow$ switch unstable!

Need to adapt schedule to traffic matrix.
Example 1 – Scheduling (Trivial)

- Assume we know the traffic matrix, it is admissible, and it follows a permutation:

\[ \Lambda = 0.99 \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \]

- Then we can simply choose:

\[ S(n) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \forall n \]
Example 2 - BvN

Consider the following traffic matrix:

\[
\Lambda = \begin{bmatrix}
0.26 & 0.57 & 0.08 & 0 & 0.91 \\
0 & 0 & 0.4 & 0.45 & 0.85 \\
0.08 & 0.24 & 0.35 & 0.15 & 0.82 \\
0.56 & 0.09 & 0 & 0.28 & 0.93 \\
0.9 & 0.9 & 0.83 & 0.88 & \\
\end{bmatrix}
\]

Step 1) Increase entries so rows/cols each add up to 1.

\[
\begin{bmatrix}
0.26 & 0.57 & 0.08 & 0 \\
0 & 0 & 0.4 & 0.45 \\
0.08 & 0.24 & 0.35 & 0.15 \\
0.56 & 0.09 & 0 & 0.28 \\
\end{bmatrix} \triangleright \begin{bmatrix}
0.3 & 0.6 & 0.1 & 0 \\
0 & 0 & 0.5 & 0.5 \\
0.1 & 0.3 & 0.4 & 0.2 \\
0.6 & 0.1 & 0 & 0.3 \\
\end{bmatrix} = \Lambda'
\]
Step 2) Find permutation matrices and weights so that ...

\[
\begin{bmatrix}
0.3 & 0.6 & 0.1 & 0 \\
0 & 0 & 0.5 & 0.5 \\
0.1 & 0.3 & 0.4 & 0.2 \\
0.6 & 0.1 & 0 & 0.3
\end{bmatrix}
= 0.2 \begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0
\end{bmatrix}
+ 0.4 \begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0
\end{bmatrix}
+ 0.3 \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
+ 0.1 \begin{bmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{bmatrix}
\]

Schedule these permutation matrices based on the weights.

For details, please see the references.
Unknown Traffic Matrix

- We want to maximize throughput
- Traffic matrix unknown \(\Rightarrow\) cannot use BvN
- **Idea**: maximize instantaneous throughput
  - In other words: transfer as many packets as possible at each time-slot
- Maximum Size Matching (MSM) algorithm
Maximum Size Matching (MSM)

- MSM maximizes instantaneous throughput

- MSM algorithm: among all maximum size matches, pick a random one

\[ Q_{11}(n) > 0 \]
\[ Q_{N2}(n) > 0 \]

Request Graph  \[ \rightarrow \]  Bipartite Match

\[ \text{Maximum Size Match} \]
Question

• Is the intuition right?
  • Answer: NO!

• There is a counter-example for which, in a given VOQ (i,j), $\lambda_{ij} < \mu_{ij}$ but MSM does not provide 100% throughput.
Counter-example

- Consider the following non-uniform traffic pattern, with Bernoulli IID arrivals:

\[
\lambda_{11} = \lambda_{12} = 1/2 - \delta \\
\lambda_{21} = 1/2 - \delta \\
\lambda_{32} = 1/2 - \delta 
\]

- Consider the case when \( Q_{21}, Q_{32} \) both have arrivals, w. p. \((1/2 - \delta)^2\).
- In this case, input 1 is served w. p. at most \(2/3\).

- Overall, the service rate for input 1, \( \mu_1 \) is at most

\[
2/3.[(1/2-\delta)^2] + 1.[1-(1/2- \delta)^2]
\]

- \( i.e. \mu_1 \leq 1 - 1/3.(1/2- \delta)^2 \).

- Switch unstable for \( \delta \leq 0.0358 \)
Simulation of Simple 3x3 Example
Outline

• Uniform traffic
  • Uniform cyclic
  • Random permutation
  • Wait-until-full

• Non-uniform traffic, known traffic matrix
  • Birkhoff-von-Neumann

• Unknown traffic matrix
  • Maximum Size Matching
  • Maximum Weight Matching
**Problem.** Maximum size matching

- Maximizes instantaneous throughput.
- Does not take into account VOQ backlogs.

**Solution.** Give higher priority to VOQs which have more packets.
Maximum Weight Matching (MWM)

- Assign weights to the edges of the request graph.

Request Graph

\[ Q_{11}(n) > 0 \]
\[ Q_{N1}(n) > 0 \]

Weighted Request Graph

\[ W_{11} \]
\[ W_{N1} \]

- Find the matching with maximum weight.
MWM Scheduling

- Create the request graph.
- Find the associated link weights.
- Find the matching with maximum weight.
  - How?
- Transfer packets from the ingress lines to egress lines based on the matching.

- **Question.** How often do we need to calculate MWM?
Weights

- **Longest Queue First (LQF)**
  - Weight associated with each link is the length of the corresponding VOQ.
  - MWM here, tends to give priority to long queues.
  - Does not necessarily serve the longest queue.

- **Oldest Cell First (OCF)**
  - Weight of each link is the waiting time of the HoL packet in the corresponding queue.
Recap

• Uniform traffic
  • Uniform cyclic
  • Random permutation
  • Wait-until-full
• Non-uniform traffic, known traffic matrix
  • Birkhoff-von-Neumann
• Unknown traffic matrix
  • Maximum Size Matching
  • Maximum Weight Matching
Summary of MWM Scheduling

- MWM – LQF scheduling provides 100% throughput.
  - It can starve some of the packets.
- MWM – OCF scheduling gives 100% throughput.
  - No starvation.

**Question.** Are these fast enough to implement in real switches?
Complexity of Maximum Matchings

- **Maximum Size Matchings:**
  - Typical complexity $O(N^{2.5})$

- **Maximum Weight Matchings:**
  - Typical complexity $O(N^3)$

- In general:
  - Hard to implement in hardware
  - Sloooooow

- Can we find a faster algorithm?
Maximal Matching

- A *maximal* matching is a matching in which each edge is added one at a time, and is not later removed from the matching.

- No augmenting paths allowed (they remove edges added earlier)

- Consequence: no input and output are left unnecessarily idle.
Example of Maximal Matching

Maximal Size Matching

Maximum Size Matching
Properties of Maximal Matchings

- In general, maximal matching is much simpler to implement, and has a much faster running time.

- A maximal size matching is at least half the size of a maximum size matching. (Why?)

- We’ll study the following algorithms:
  - Greedy LQF
  - WFA
  - PIM
  - iSLIP
Greedy LQF

- **Greedy LQF** (Greedy Longest Queue First) is defined as follows:
  - Pick the VOQ with the most number of packets (if there are ties, pick at random among the VOQs that are tied). Say it is VOQ($i_1, j_1$).
  - Then, among all free VOQs, pick again the VOQ with the most number of packets (say VOQ($i_2, j_2$), with $i_2 \neq i_1$, $j_2 \neq j_1$).
  - Continue likewise until the algorithm converges.
- Greedy LQF is also called iLQF (iterative LQF) and Greedy Maximal Weight Matching.
Properties of Greedy LQF

- The algorithm converges in at most $N$ iterations. (Why?)
- Greedy LQF results in a maximal size matching. (Why?)
- Greedy LQF produces a matching that has at least half the size and half the weight of a maximum weight matching. (Why?)
Wave Front Arbiter (WFA) [Tamir and Chi, 1993]

Requests

Match
Wave Front Arbiter

Requests

Match
Wave Front Arbiter – Implementation

Simple combinational logic blocks
Wave Front Arbiter – \textit{Wrapped WFA (WWFA)}

$N$ steps instead of $2N-1$
Properties of Wave Front Arbiters

- Feed-forward (i.e. non-iterative) design lends itself to pipelining.
- Always finds maximal match.
- Usually requires mechanism to prevent $Q_{11}$ from getting preferential service.
- In principle, can be distributed over multiple chips.
Parallel Iterative Matching [Anderson et al., 1993]

Iteration:

#1

Φ1: Requests

1 → 1
2 → 2
3 → 3
4 → 4

Φ2: Grant

1 → 1
2 → 2
3 → 3
4 → 4

Φ3: Accept/Match

1 → 1
2 → 2
3 → 3
4 → 4
PIM Properties

- Guaranteed to find a maximal match in at most $N$ iterations. (Why?)
- In each phase, each input and output arbiter can make decisions independently.
- In general, will converge to a maximal match in $< N$ iterations.
- How many iterations should we run?
Parallel Iterative Matching – Convergence Time

Number of iterations to converge:

\[
E[U_i] \leq \frac{N^2}{4^i}
\]
\[
E[C] \approx \log N
\]

C = # of iterations required to resolve connections
N = # of ports
U_i = # of unresolved connections after iteration i

Anderson et al., “High-Speed Switch Scheduling for Local Area Networks,” 1993.
Parallel Iterative Matching
Parallel Iterative Matching

PIM with a single iteration
Parallel Iterative Matching

PIM with 4 iterations
**iSLIP [McKeown et al., 1993]**

1. **Φ1: Requests**
   - #1
   - #2

2. **Φ2: Grant**
   - #1
   - #2

3. **Φ3: Accept/Match**
   - #1
   - #2
iSLIP Operation

- **Grant phase**: Each output selects the requesting input at the pointer, or the next input in round-robin order. It only updates its pointer if the grant is accepted.
- **Accept phase**: Each input selects the granting output at the pointer, or the next output in round-robin order.
- **Consequence**: Under high load, grant pointers tend to move to unique values.
iSLIP Properties

- Random under low load
- TDM under high load
- Lowest priority to MRU (most recently used)
- 1 iteration: fair to outputs
- Converges in at most N iterations. (On average, simulations suggest < log2N)
- Implementation: N priority encoders
- 100% throughput for uniform i.i.d. traffic.
- But...some pathological patterns can lead to low throughput.
iSLIP

![Graph showing performance metrics over load](image)

- **X-axis:** Offered Load (%)
- **Y-axis:** Avg Cell Latency (Cells)

Legend:
- FIFO
- OUTPUT
- Iterations

- Iterations 1
- Iterations 2
- Iterations 4

Observations:
- As load increases, cell latency also increases.
- FIFO performs better than OUTPUT at lower loads.
- Iterations have a minimal impact on latency at high loads.
iSLIP Implementation

Programmable Priority Encoder

State

Grant

Accept

Decision

N

N

N

N

1

2

1

2

log_2 N

log_2 N

log_2 N

log_2 N

Programmable Priority Encoder

Grant

Accept

N

N

N

N

1

2

1

2

log_2 N

log_2 N

log_2 N

log_2 N


Maximal Matches

- Maximal matching algorithms are widely used in industry (especially algorithms based on WFA and iSLIP).

- PIM and iSLIP are rarely run to completion (i.e. they are sub-maximal).

- A maximal match with a speedup of 2 is stable for non-uniform traffic.
References


References


