IP Lookup: Application Requirements and concurrency issues

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology
A packet is routed based on the “Longest Prefix Match” (LPM) of its IP address with entries in a routing table.

Line rate and the order of arrival must be maintained.

Line rate $\Rightarrow 15\text{Mpps for 10GE}$
**Sparse tree representation**

<table>
<thead>
<tr>
<th>IP address</th>
<th>Result</th>
<th>M Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.13.7.3</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>10.18.201.5</td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>7.14.7.2</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>5.13.7.2</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>10.18.200.7</td>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

In this lecture:
- Level 1: 16 bits
- Level 2: 8 bits
- Level 3: 8 bits

⇒ 1 to 3 memory accesses
“C” version of LPM

```c
int lpm (IPA ipa)
/* 3 memory lookups */
{
    int p;
    /* Level 1: 16 bits */
    p = RAM [ipa[31:16]];
    if (isLeaf(p)) return value(p);
    /* Level 2: 8 bits */
    p = RAM [ptr(p) + ipa [15:8]];
    if (isLeaf(p)) return value(p);
    /* Level 3: 8 bits */
    p = RAM [ptr(p) + ipa [7:0]];
    return value(p);
    /* must be a leaf */
}
```

Not obvious from the C code how to deal with
- memory latency
- pipelining

Memory latency
\~30ns to 40ns

- Must process a packet every 1/15 ms or 67 ns
- Must sustain 3 memory dependent lookups in 67 ns
Longest Prefix Match for IP lookup: 3 possible implementation architectures

- **Rigid pipeline**: Inefficient memory usage but simple design
- **Linear pipeline**: Efficient memory usage through memory port replicator
- **Circular pipeline**: Efficient memory with most complex control

**Designer's Ranking**: 1. **Which is "best"?** 2. 3.

Arvind, Nikhil, Rosenband & Dave [ICCAD 2004]
Completion buffer ensures that departures take place in order even if lookups complete out-of-order

Since cbuf has finite capacity it gives out tokens to control the entry into the circular pipeline

The fifo must also hold the “token” while the memory access is in progress: Tuple2#(Token, Bit#(16))
interface Mem#(type addrT, type dataT);
    method Action req(addrT x);
    method Action deq;
    method dataT peek;
endinterface

Use a BSV wrapper to make a synchronous component latency-insensitive
Completion buffer

Completion buffer is used to restore the order in which the processing of inputs was started

- Tokens are given out in order, e.g., (1,2,3,...,16,1,2,...)
- Data with a token can be put in any order in cbuf
- Results are returned in the same order in which tokens were issued

interface CBuffer#(type t);
  method ActionValue#(Token) getToken;
  method Action put(Token tok, t d);
  method ActionValue#(t) getResult;
endinterface
module mkIPLookup(IPLookup);
    instantiate cbuf, RAM and fifo
    rule recirculate...  
    method Action enter (IP ip);
        Token tok <- cbuf.getTokken;
        ram.req(ip[31:16]);
        fifo.enq(tuple2(tok,ip[15:0]));
    endmethod
    method ActionValue #(Msg) getResult();
        let result <- cbuf.getResult;
        return result;
    endmethod
endmodule

When can enter fire?

cbuf, ram & fifo, each has space (is rdy)
Circular Pipeline Rules:

When can recirculate fire?

- ram & fifo each has an element and ram and fifo, or cbuf has space

```plaintext
rule recirculate;
  match{.tok,.rip} = fifo.first;
  fifo.deq; ram.deq;
  if (isLeaf(ram.peek))
    cbuf.put(tok, ram.peek);
  else begin
    fifo.enq(tuple2(tok,(rip << 8)));
    ram.req(ram.peek + rip[15:8]);
  end
endrule
```

Requires simultaneous enq and deq in the same rule!
Is this possible?
Performance

Can a new request enter the system when an old one is leaving?

No

Dead cycle

Is this worth worrying about?

rule recirculate;
  match{.tok,.rip} = fifo.first;
  fifo.deq; ram.deq;
  if (isLeaf(ram.peek))
    cbuf.put(tok, ram.peek);
  else begin
    fifo.enq(tuple2(tok,(rip << 8)));
    ram.req(ram.peek + rip[15:8]);
  end
endrule

method Action enter (IP ip);
  Token tok <- cbuf.getToken;
  ram.req(ip[31:16]);
  fifo.enq(tuple2(tok,ip[15:0]));
endmethod
The Effect of Dead Cycles

What is the performance loss if “exit” and “enter” can’t ever happen in the same cycle?

>33% slowdown! Unacceptable

Circular Pipeline

- RAM takes several cycles to respond to a request
- Each IP request generates 1-3 RAM requests
- FIFO entries hold base pointer for next lookup and unprocessed part of the IP address

What is the performance loss if “exit” and “enter” can’t ever happen in the same cycle?
Dead Cycles

In general enter and recirculate conflict but when isLeaf(p) is true there is no apparent conflict!

```plaintext
rule recirculate;
  match{.tok,.rip} = fifo.first;
  fifo.deq; ram.deq;
  if (isLeaf(ram.peek))
    cbuf.put(tok, ram.peek);
endrule

method Action enter (IP ip);
  Token tok <- cbuf.getToken;
  ram.req(ip[31:16]);
  fifo.enq(tuple2(tok, ip[15:0]));
endmethod
```

September 25, 2019
http://csg.csail.mit.edu/6.375
L09-13
Rule Spliting

rule foo;
  if (p) r1 <= 5;
  else r2 <= 7;
endrule

rule baz;
  r1 <= 9;
endrule

- rules foo and baz conflict
- rules fooF and baz do not and can be scheduled together

\[ \equiv \]

rule fooT if (p);
  r1 <= 5;
endrule

rule fooF if (!p);
  r2 <= 7;
endrule

rule baz;
  r1 <= 9;
endrule
Splitting the recirculate rule

```
rule recirculate(!isLeaf(ram.peek));
    match{.tok,.rip} = fifo.first;
    fifo.enq(tuple2(tok,(rip << 8)));
    ram.req(ram.peek + rip[15:8]);
    fifo.deq; ram.deq;
endrule
```

```
rule exit (isLeaf(ram.peek));
    match{.tok,.rip} = fifo.first;
    cbuf.put(tok, ram.peek);
    fifo.deq; ram.deq;
endrule
```

This rule is valid only if enq and deq can be executed concurrently

```
method Action enter (IP ip);
    Token tok <= cbuf.getToken;
    ram.req(ip[31:16]);
    fifo.enq(
        tuple2(tok,ip[15:0]));
endmethod
```

Rule exit and method enter can execute concurrently, if cbuf.put and cbuf.getToken can execute concurrently.
Concurrent FIFO methods
pipelined FIFO

```plaintext
rule foo;
  f.enq (5) ; f.deq;
endrule
```

make implicit conditions explicit

```plaintext
rule foo (f.notFull && f.notEmpty);
  f.enq (5) ; f.deq;
endrule
```

Can foo be enabled?

- f.notFull can be calculated only after knowing if f.deq fires or not, i.e. there is a combinational path from enable of f.deq to f.notFull
- Firing condition for rule foo has to be independent of the body
Concurrent FIFO methods

CF FIFO

The firing condition for rule foo is independent of the body
The FIFO in the IP lookup must therefore be CF

```
rule foo;
    f.enq (5) ; f.deq;
endrule
```

```
rule foo (f.notFull && f.notEmpty);
    f.enq (5) ; f.deq;
endrule
```

Can foo be enabled?
Two-Element FIFO

module mkCFFifo (Fifo#(2, Bit#(n)));
    Ehr#(2, Bit#(n)) da <- mkEhr(?);
    Ehr#(2, Bool) va <- mkEhr(False);
    Ehr#(2, Bit#(n)) db <- mkEhr(?);
    Ehr#(2, Bool) vb <- mkEhr(False);
    rule canonicalize (vb[1] && !va[1]);
        vb[1] <= False;
    endrule
    method Action enq(Bit#(n) x) if (!vb[0]);
        db[0] <= x; vb[0] <= True;
    endmethod
    method Action deq if (va[0]);
        va[0] <= False;
    endmethod
    method Bit#(n) first if (va[0]);
        return da[0];
    endmodule

In any given cycle simultaneous enq and deq are permitted provided the FIFO is neither full nor empty.
Completion buffer: Implementation

- A circular buffer with two pointers iidx and ridx, and a counter cnt
- Each data element has a valid bit associated with it

```verilog
module mkCompletionBuffer(CompletionBuffer#(t));
  Vector#(32, Reg#(Bool)) cbv <- replicateM(mkReg(False));
  Vector#(32, Reg#(t)) cbData <- replicateM(mkRegU());
  Reg#(Bit#(5)) iidx <- mkReg(0);
  Reg#(Bit#(5)) ridx <- mkReg(0);
  Reg#(Bit#(6)) cnt <- mkReg(0);
  rules and methods...
endmodule
```
Completion Buffer cont

```vhdl
method ActionValue#(Bit#(5)) getToken() if (cnt < 32);
    cbv[iidx] <= False;
    iidx <= (iidx==31) ? 0 : iidx + 1;
    cnt <= cnt + 1;
    return iidx;
endmethod

method Action put(Token idx, t data);
    cbData[idx] <= data;
    cbv[idx] <= True;
endmethod

method ActionValue#(t) getResult() if ((cnt > 0)&&(cbv[ridx]));
    cbv[ridx] <= False;
    ridx <= (ridx==31) ? 0 : ridx + 1;
    cnt <= cnt - 1;
    return cbData[ridx];
endmethod
```

Concurrency properties?
Completion buffer
Concurrency requirements

```haskell
interface CBuffer#(type t);
  method ActionValue#(Token) getToken;
  method Action put(Token tok, t d);
  method ActionValue#(t) getResult;
endinterface
```

- For no dead cycles `getToken`, `put` and `getResult` must be able to execute concurrently.
- If we make these methods CF then every thing will work concurrently, i.e. (enter CF exit), (enter CF `getResult`) and (exit CF `getResult`)
- However CF methods are hard to design. Suppose (getToken < put), (getToken < `getResult`) and (put < `getResult`) then (enter < exit), (enter < `getResult`) and (exit < `getResult`)
- In fact, any ordering will work
Longest Prefix Match for IP lookup: 3 possible implementation architectures

Rigid pipeline
Inefficient memory usage but simple design

Linear pipeline
Efficient memory usage through memory port replicator

Circular pipeline
Efficient memory with most complex control

Which is "best"?
Implementations of Static pipelines
Two designers, two results

<table>
<thead>
<tr>
<th>LPM versions</th>
<th>Best Area (gates)</th>
<th>Best Speed (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static V (Replicated FSMs)</td>
<td>8898</td>
<td>3.60</td>
</tr>
<tr>
<td>Static V (Single FSM)</td>
<td>2271</td>
<td>3.56</td>
</tr>
</tbody>
</table>

Each packet is processed by one FSM

BEST:

- IP addr
- mux/ de-mux
- RAM

Replicated

- IP addr
- mux/ de-mux
- RAM

Shared FSM
## Synthesis results

<table>
<thead>
<tr>
<th>LPM versions</th>
<th>Code size (lines)</th>
<th>Best Area (gates)</th>
<th>Best Speed (ns)</th>
<th>Mem. util. (random workload)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static V</td>
<td>220</td>
<td>2271</td>
<td>3.56</td>
<td>63.5%</td>
</tr>
<tr>
<td>Static BSV</td>
<td>179</td>
<td>2391 (5% larger)</td>
<td>3.32 (7% faster)</td>
<td>63.5%</td>
</tr>
<tr>
<td>Linear V</td>
<td>410</td>
<td>14759</td>
<td>4.7</td>
<td>99.9%</td>
</tr>
<tr>
<td>Linear BSV</td>
<td>168</td>
<td>15910 (8% larger)</td>
<td>4.7 (same)</td>
<td>99.9%</td>
</tr>
<tr>
<td>Circular V</td>
<td>364</td>
<td>8103</td>
<td>3.62</td>
<td>99.9%</td>
</tr>
<tr>
<td>Circular BSV</td>
<td>257</td>
<td>8170 (1% larger)</td>
<td>3.67 (2% slower)</td>
<td>99.9%</td>
</tr>
</tbody>
</table>

V=Verilog

- Bluespec results can match carefully coded Verilog
- Micro-architecture has a dramatic impact on performance
- Architecture differences are much more important than language differences in determining QoR

Synthesis: TSMC 0.18 µm lib