IP Lookup: Some subtle concurrency issues

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

A packet is routed based on the “Longest Prefix Match” (LPM) of it’s IP address with entries in a routing table.

Line rate and the order of arrival must be maintained.

Line rate $\Rightarrow$ 15Mpps for 10GE
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 */
}
```

Must process a packet every $\frac{1}{15} \mu s$ 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:**

*Which is “best”?*

Arvind, Nikhil, Rosenband & Dave [ICCAD 2004]

---

IP-Lookup module: Circular pipeline

- 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))

remainingIP

March 4, 2013  http://csg.csail.mit.edu/6.375  L08-6
**Completion buffer:**

**Interface**

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

typedef Bit#{TLog#{(n)}} TokenN#{numeric type n};
typedef TokenN#{(16)} Token;

cbuf getResult getToken
put (result & token)

---

**Request-Response Interface for Synchronous Memory**

```
interface Mem#{type addrT, type dataT);
method Action req(addrT x);
method Action deq;
method dataT peek;
endinterface
```

Making a synchronous component latency-insensitive
**IP-Lookup module: Interface methods**

```verilog
done?
RAM
cbuf

module mkIPLookup(IPLookup);
  rule recirculate;
   method Action enter (IP ip);
    Token tok <- cbuf.getToken;
    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
```

**Circular Pipeline Rules:**

```verilog
done?
RAM
fifo

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
```

Dead Cycles

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

Is this worth worrying about?

The Effect of Dead Cycles

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” don’t ever happen in the same cycle?
So is there a dead cycle?

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

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

Rule Spliting

rule foo (True);
  if (p) r1 <= 5;
  else r2 <= 7;
endrule
Splitting the recirculate rule

```plaintext
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
```

Concurrent FIFO methods

pipelined FIFO

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

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

Can foo be enabled?


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

March 4, 2013  http://csg.csail.mit.edu/6.375  L08-16
Concurrent FIFO methods

CF FIFO

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

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

Can foo be enabled?

make implicit conditions explicit

Completion buffer: Interface

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

getToken  cbuf  getResult

put (result & token)
Completion buffer: Completion buffer: Completion buffer:
Concurrency requirements Concurrency requirements Concurrency requirements

- For no dead cycles `cbuf.getToken` and `cbuf.put` and `cbuf.getResult` must be able to execute concurrently.
- If we make these methods CF then everything will work concurrently, i.e. (enter CF exit), (enter CF `getResult`) and (exit CF `getResult`).

Completion buffer: Completion buffer: Completion buffer:
Implementation Implementation Implementation

A circular buffer with two pointers `iiidx` and `ridx`, and a counter `cnt`.

Elements are of Maybe type.

```
module mkCompletionBuffer(CompletionBuffer#(size));
  Vector#(size, EHR#(Maybe#(t))) cb <- replicateM(mkEHR(Invalid));
  Reg#(Bit#(TAdd#(TLog#(size),1))) iiidx <- mkReg(0);
  Reg#(Bit#(TAdd#(TLog#(size),1))) ridx <- mkReg(0);
  EHR#(Bit#(TAdd#(TLog#(size),1))) cnt <- mkEHR(0);
  Integer vsize = valueOf(size);
  Bit#(TAdd#(TLog#(size),1)) sz = fromInteger(vsize);
  rules and methods...
endmodule
```

March 4, 2013 http://csg.csail.mit.edu/6.375 L08-20
Completion Buffer cont

```verbatim
method ActionValue#(t) getToken() if (cnt[0] !== sz);
    cb[iidx][0] <= Invalid;
    iidx <= iidx==sz-1 ? 0 : iidx + 1;
    cnt[0] <= cnt[0] + 1;
    return iidx;
endmethod

method Action put(Token idx, t data);
    cb[idx][1] <= Valid data;
endmethod

method ActionValue#(t) getResult() if (cnt[1] !== 0
    && (cb[ridx][2] matches tagged (Valid .x)));
    cb[ridx][2] <= Invalid;
    ridx <= ridx==sz-1 ? 0 : ridx + 1;
    return x;
endmethod
```

Concurrency properties?

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"?

Arvind, Nishil, Rosenband & Dave [ICCAD 2004]
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>

Replicated:
- IP addr
- MUX / De-MUX
- FSM
- FSM
- FSM
- Counter
- MUX / De-MUX
- RAM

BEST:
- IP addr
- result
- MUX
- Shared FSM
- FSM
- RAM

March 4, 2013
http://csg.csail.mit.edu/6.375
L08-23

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>

- 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

March 4, 2013
http://csg.csail.mit.edu/6.375
L08-24