IP Lookup: Some subtle concurrency issues

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

February 22, 2011
http://csg.csail.mit.edu/6.375

IP Lookup block in a router

- 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 $\Rightarrow$ 15Mpps for 10GE
Sparse tree representation

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

"C" version of LPM

```c
int lpm (IPA ipa)
{ 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 1/15 µs or 67 ns
Must sustain 3 memory dependent lookups in 67 ns
Longest Prefix Match for IP lookup:
3 possible implementation architectures

Rigid pipeline  Linear pipeline  Circular pipeline

Inefficient memory usage but simple design
Efficient memory usage through memory port replicator
Efficient memory with most complex control

Designer’s Ranking:

Which is “best”?

Arvind, Nikhil, Rosenband & Dave [ICCAD 2004]

Circular pipeline

The fifo holds the request while the memory access is in progress

The architecture has been simplified for the sake of the lecture. Otherwise, a “completion buffer” has to be added at the exit to make sure that packets leave in order.
**FIFO**

```
interface FIFO#(type t);
    method Action enq(t x); // enqueue an item
    method Action deq(); // remove oldest entry
    method t first(); // inspect oldest item
endinterface
```

$n = \# \text{ of bits needed to represent a value of type } t$

**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
Circular Pipeline Code

rule enter (True);
  IP ip = inQ.first();
  ram.req(ip[31:16]);
  fifo.enq(ip[15:0]);
  inQ.deq();
endrule

When can enter fire?

rule recirculate (True);
  TableEntry p = ram.peek(); ram.deq();
  IP rip = fifo.first();
  if (isLeaf(p)) outQ.enq(p);
  else begin
    fifo.enq(rip << 8);
    ram.req(p + rip[15:8]);
  end
  fifo.deq();
endrule

February 22, 2011 L06-9
http://csg.csail.mit.edu/6.375

Circular Pipeline Code:
discussion

rule enter (True);
  IP ip = inQ.first();
  ram.req(ip[31:16]);
  fifo.enq(ip[15:0]);
  inQ.deq();
endrule

When can recirculate fire?

rule recirculate (True);
  TableEntry p = ram.peek(); ram.deq();
  IP rip = fifo.first();
  if (isLeaf(p)) outQ.enq(p);
  else begin
    fifo.enq(rip << 8);
    ram.req(p + rip[15:8]);
  end
  fifo.deq();
endrule

February 22, 2011 L06-10
http://csg.csail.mit.edu/6.375
Ordinary FIFO won’t work but a pipeline FIFO would

Problem solved!

```vhdl
PipelineFIFO fifo <- mkPipelineFIFO;
  // use a Pipeline fifo
rule recirculate (True);
  TableEntry p = ram.peek();
  ram.deq();
  IP rip = fifo.first();
  if (isLeaf(p)) outQ.enq(p);
  else begin
    fifo.enq(rip << 8);
    ram.req(p + rip[15:8]);
  end
  fifo.deq();
endrule
```

◆ RWire has been safely encapsulated inside the Pipeline FIFO – users of the fifo need not be aware of RWires
Dead cycles

```
rule enter (True);
    IP ip = inQ.first();
    ram.req(ip[31:16]);
    fifo.enq(ip[15:0]); inQ.deq();
endrule

rule recirculate (True);
    TableEntry p = ram.peek(); ram.deq();
    IP rip = fifo.first();
    if (isLeaf(p)) outQ.enq(p);
    else begin
        fifo.enq(rip << 8);
        ram.req(p + rip[15:8]);
    end
    fifo.deq();
endrule
```

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

The Effect of Dead Cycles

- 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?
Scheduling conflicting rules

- When two rules conflict on a shared resource, they cannot both execute in the same clock
- The compiler produces logic that ensures that, when both rules are applicable, only one will fire
  - Which one?

source annotations

---

So is there a dead cycle?

```verilog
rule enter (True);
    IP ip = inQ.first();
    ram.req(ip[31:16]);
    fifo.enq(ip[15:0]); inQ.deq();
endrule

rule recirculate (True);
    TableEntry p = ram.peek(); ram.deq();
    IP rip = fifo.first();
    if (isLeaf(p)) outQ.enq(p);
    else begin
        fifo.enq(rip << 8);
        ram.req(p + rip[15:8]);
    end
    fifo.deq();
endrule
```

In general these two rules conflict but when `isLeaf(p)` is true there is no apparent conflict!
Rule Splitting

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

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

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

rule fooT and fooF can be scheduled independently with some other rule

Spliting the recirculate rule

rule recirculate (!isLeaf(ram.peek()));
  IP rip = fifo.first(); fifo.enq(rip << 8);
  ram.req(ram.peek() + rip[15:8]);
  fifo.deq(); ram.deq();
endrule

rule exit (isLeaf(ram.peek()));
  outQ.enq(ram.peek()); fifo.deq(); ram.deq();
endrule

rule enter (True);
  IP ip = inQ.first(); ram.req(ip[31:16]);
  fifo.enq(ip[15:0]); inQ.deq();
endrule

Now rules enter and exit can be scheduled simultaneously, assuming fifo.enq and fifo.deq can be done simultaneously.
Packaging a module:
Turning a rule into a method

```verilog
rule enter (True);
  IP ip = inQ.first();
  ram.req(ip[31:16]);
  fifo.enq(p[15:0]);
  inQ.deq();
endrule
```

IP-Lookup module with the completion buffer

- 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 now must also hold the "token" while the memory access is in progress: Tuple2#(Token, Bit#(16))
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;

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

for enter and
getResult to execute
simultaneously,
cbuf.getToken
and
cbuf.getResult
must execute simultaneously
IP Lookup rules with completion buffer

```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()));
    cbuf.put(ram.peek()); fifo.deq(); ram.deq();
endrule
```

For rule `exit` and method `enter` to execute simultaneously, `cbuf.put` and `cbuf.getToken` must execute simultaneously.

→ For no dead cycles, `cbuf.getToken` and `cbuf.put` and `cbuf.getResult` must be able to execute simultaneously.

Completion buffer: Interface Requirements

```
getToken

<table>
<thead>
<tr>
<th>cbuf</th>
<th>getResult</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

put (result & token)
```

Rules and methods concurrency requirement to avoid dead-cycles:

- `exit < getResult < enter`

⇒ `cbuf` methods' concurrency:

`cbuf.getResult < cbuf.put < cbuf.getToken`
Completion buffer: Implementation

A circular buffer with two pointers i and o, and a counter cnt

Elements are of Maybe type

module mkCBuffer (CBuffer#(t))
provisos (Bits#(t,sz))
RegFile#(Token, Maybe#(t)) buf <- mkRegFileFull();
Reg#(Token) i <- mkReg(0); //input index
Reg#(Token) o <- mkReg(0); //output index
Reg#(Int#(32)) cnt <- mkReg(0); //number of filled slots

Elements must be representable as bits


Completion buffer: Implementation  Problem 1

A circular buffer with two pointers i and o, and a counter cnt

Elements are of Maybe type

◆ buf must allow two simultaneous updates and one read
  ■ Needs a register file with one read and two write ports
◆ Since the updates are always to different addresses there is no data hazard and concurrent operations should be permitted

Completion buffer: Implementation Problem 2

// state elements
// buf, i, o, cnt ...

method ActionValue#(t) getToken()
    if (cnt < maxToken);
        cnt <= cnt + 1;
        i <= (i==maxToken) ? 0 : i+1; buf.upd(i, Invalid);
    return i;
endmethod

method Action put(Token tok, t data);
    buf.upd(tok, Valid data);
endmethod

method ActionValue#(t) getResult()
    if (cnt > 0) &&
        (buf.sub(o) matches tagged (Valid .x));
    o <= (o==maxToken) ? 0 : o + 1; cnt <= cnt - 1;
    return x;
endmethod

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, Nikhil, 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 ↓ | result
MUX / De-MUX
FSM  FSM  FSM  FSM
Counter MUX / De-MUX
RAM

BEST:                               
IP addr
MUX
FSM
RAM

Each packet is processed by one 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>

- 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

V = Verilog; BSV = Bluespec System Verilog