Bluespec-4: Rule Scheduling and Synthesis

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology
Exploring microarchitectures

IP Lookup Module
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.
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>

Real-world lookup algorithms are more complex but all make a sequence of dependent memory references.
Table representation issues

Table size
- Depends on the number of entries: 10K to 100K
- Too big to fit on chip memory ⇒ SRAM ⇒ DRAM ⇒ latency, cost, power issues

Number of memory accesses for an LPM?
- Too many ⇒ difficult to do table lookup at line rate (say at 10Gbps)

Control-plane issues:
- Incremental table update
- Size, speed of table maintenance software

In this lecture (to fit the code on slides!):
- Level 1: 16 bits, Level 2: 8 bits, Level 3: 8 bits
  ⇒ from 1 to 3 memory accesses for an LPM
"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 p;
  /* Level 2: 8 bits */
  p = RAM [p + ipa [15:8]];
  if (isLeaf(p)) return p;
  /* Level 3: 8 bits */
  p = RAM [p + ipa [7:0]];
  return p; /* must be a leaf */
}
```

How to implement LPM in HW?
Not obvious from the C code!

Must process a packet every 1/15 μs or 67 ns
Must sustain 3 memory dependent lookups in 67 ns
Static Pipeline

Assume the memory has a latency of n cycles and can accept a request every cycle.

Inefficient memory usage – unused memory slots represent wasted bandwidth.

Difficult to schedule table updates.
Circular pipeline

Completion buffer
- gives out tokens to control the entry into the circular pipeline
- ensures that departures take place in order even if lookups complete out-of-order
RAMs: Synchronous vs Asynchronous view

- Basic memory components are "synchronous":
  - Present a read-address $A_j$ on clock $J$
  - Data $D_j$ arrives on clock $J+N$
  - If you don't "catch" $D_j$ on clock $J+N$, it may be lost, i.e., data $D_{j+1}$ may arrive on clock $J+1+N$

- This kind of synchronicity can pervade the design and cause complications
Asynchronous RAMs

Asynchronous RAMs are easier to work with compared to synchronous RAMs. They have a lower latency, and the control logic is simpler. Here is a brief explanation:

interface AsyncRAM#(type addr_T, type data_T);
  method Action req(addr_T a);
  method ActionValue#(data_T) resp();
endinterface

It's easier to work with an "asynchronous" block.
rule static (True);
    if (c5 == 3) begin
        IP ip = in.first();
        ram.req(ip[31:16]); r1 <= ip[15:0];
        in.deq(); c1 <= 1;
    end
    else begin
        r1 <= r5; c1 <= c5+1;
        ram.req(r5);
    end
    r2 <= r1; c2 <= c1;
    r3 <= r2; c3 <= c2;
    r4 <= r3; c4 <= c3;
    TableEntry p <- ram.resp();
    r5 <= nextReq(p, r4); c5 <= c4;
    if (c5 == 3) out.enq(r5);
endrule
Circular Pipeline Code

```
rule enter (True);
    Token t <- cbuf.getToken();
    IP ip = in.first();
    ram.req(ip[31:16]);
    active.enq(tuple2(ip[15:0], t)); in.deq();
endrule

rule done (True);
    TableEntry p <- ram.resp();
    match {.rip, .t} = active.first();
    if (isLeaf(p)) cbuf.done(t, p);
    else begin
        active.enq(rip << 8, t);
        ram.req(p + signExtend(rip[15:7]));
        end
    active.deq();
endrule
```

March 6, 2006
Completion buffer

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

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

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

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

method Action done(Token t, any_T data);
    return buf.upd(t, Valid data);
endmethod

method ActionValue#(any_T) get() if (cnt > 0) &&
    (buf.sub(o) matches tagged (Valid .x));
    o <= o + 1;
    cnt <= cnt - 1;
    return x;
endmethod
Synthesis from rules ...

we will revisit IP LPM block synthesis results after a better understanding of the synthesis procedure
Synthesis: From State & Rules into Synchronous FSMs

Interface

Module

Transition Logic

Collection of State Elements

Transition Logic

S "Next"

I

S

O

http://csg.csail.mit.edu/6.375/
Hardware Elements

- **Combinational circuits**
  - Mux, Demux, ALU, ...

- **Synchronous state elements**
  - Flipflop, Register, Register file, SRAM, DRAM

---

**Mux**

<table>
<thead>
<tr>
<th>Sel</th>
<th>I₀</th>
<th>I₁</th>
<th>...</th>
<th>Iₙ</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Demux**

<table>
<thead>
<tr>
<th>Sel</th>
<th>I</th>
<th>O₀</th>
<th>O₁</th>
<th>...</th>
<th>Oₙ</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ALU**

<table>
<thead>
<tr>
<th>OpSelect</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Add, Sub, AddU, ...</td>
</tr>
<tr>
<td>- And, Or, Not, ...</td>
</tr>
<tr>
<td>- GT, LT, EQ, ...</td>
</tr>
<tr>
<td>- SL, SR, SRA, ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result</th>
<th>NCVZ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register**

<table>
<thead>
<tr>
<th>D</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>En</td>
<td>Clock</td>
<td>ff</td>
<td>ff</td>
<td>ff</td>
<td>ff</td>
<td>ff</td>
<td>ff</td>
<td>ff</td>
<td>ff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q</th>
<th>Q</th>
<th>Q</th>
<th>Q</th>
<th>Q</th>
<th>Q</th>
<th>Q</th>
<th>Q</th>
<th>Q</th>
<th>Q</th>
<th>Q</th>
</tr>
</thead>
</table>

March 6, 2006
Flip-flops with Write Enables

Edge-triggered: Data is sampled at the rising edge

dangerous!
Semantics and synthesis

Rules
Semantics: “Untimed” (one rule at a time)

Scheduling and Synthesis by the BSV compiler

Verilog RTL
Semantics: clocked synchronous HW (multiple rules per clock)

Verification activities
Using Rule Semantics, establish functional correctness

Using Schedules, establish performance correctness
Rule semantics

Given a set of rules and an initial state

while ( some rule is applicable in the current state )
  ■ choose one applicable rule
  ■ apply that rule to the current state to produce the next state of the system*

(* ) These rule semantics are “untimed” – the action to change the state can take as long as necessary provided the state change is seen as atomic, i.e., not divisible.

Bluespec synthesis is all about executing many rules concurrently while preserving the above semantics
Rule: As a State Transformer

A rule may be decomposed into two parts $\pi(s)$ and $\delta(s)$ such that

$$s_{next} = \text{if } \pi(s) \text{ then } \delta(s) \text{ else } s$$

$\pi(s)$ is the condition (predicate) of the rule, a.k.a. the “CAN_FIRE” signal of the rule. (conjunction of explicit and implicit conditions)

$\delta(s)$ is the “state transformation” function, i.e., computes the next-state value in terms of the current state values.
Compiling a Rule

rule r (f.first() > 0) ;
    x <= x + 1 ;   f.deq ();
endrule

\( \pi \) = enabling condition
\( \delta \) = action signals & values

\[ \pi = \text{enabling condition} \]
\[ \delta = \text{action signals & values} \]
Combining State Updates: strawman

What if more than one rule is enabled?
Combining State Updates

Scheduler: Priority Encoder

\[ \pi_1 \rightarrow \phi_1 \]
\[ \vdots \]
\[ \pi_n \rightarrow \phi_n \]

\[ \delta_1, R \rightarrow \text{latch enable} \]
\[ \vdots \]
\[ \delta_n, R \rightarrow \text{next state value} \]

\( \delta's \text{ from the rules that update } R \)
\( \pi's \text{ from all the rules} \)

Scheduler ensures that at most one \( \phi_i \) is true
One-rule-at-a-time Scheduler

1. $\phi_i \Rightarrow \pi_i$

2. $\pi_1 \lor \pi_2 \lor \ldots \lor \pi_n \Rightarrow \phi_1 \lor \phi_2 \lor \ldots \lor \phi_n$

3. One rewrite at a time
   i.e. at most one $\phi_i$ is true
Executing Multiple Rules Per Cycle

Can these rules be executed simultaneously?

These rules are “conflict free” because they manipulate different parts of the state.

```
rule ra (z > 10);
  x <= x + 1;
endrule

rule rb (z > 20);
  y <= y + 2;
endrule
```

Rule\textsubscript{a} and Rule\textsubscript{b} are conflict-free if

\[ \forall s. \pi_a(s) \land \pi_b(s) \Rightarrow \]

1. \[ \pi_a(\delta_b(s)) \land \pi_b(\delta_a(s)) \]
2. \[ \delta_a(\delta_b(s)) = \delta_b(\delta_a(s)) \]
Executing Multiple Rules Per Cycle

Can these rules be executed simultaneously?

These rules are “sequentially composable”, parallel execution behaves like \( ra < rb \)

\[
\text{Rule}_a \quad \text{and Rule}_b \quad \text{are sequentially composable if}
\]
\[
\forall s . \pi_a(s) \land \pi_b(s) \implies \pi_b(\delta_a(s))
\]
Multiple-Rules-per-Cycle Scheduler

1. $\phi_i \Rightarrow \pi_i$

2. $\pi_1 \lor \pi_2 \lor \ldots \lor \pi_n \Rightarrow \phi_1 \lor \phi_2 \lor \ldots \lor \phi_n$

3. Multiple operations such that $\phi_i \land \phi_j \Rightarrow R_i$ and $R_j$ are conflict-free or sequentially composable.

Divide the rules into smallest conflicting groups; provide a scheduler for each group.
Muxing structure

Muxing logic requires determining for each register (action method) the rules that update it and under what conditions.

Conflict Free

\[
\begin{align*}
\delta_1 \quad & \quad \pi_1 \\
\delta_2 \quad & \quad \pi_2 \\
\text{and} \quad & \quad \text{or} \\
\end{align*}
\]

\[\pi_1 \Rightarrow \sim \pi_2\]

Sequentially composable

\[
\begin{align*}
\pi_1 \quad & \quad \delta_1 \delta_2 \\
\sim \pi_2 \quad & \quad \pi_1 \pi_2 \\
\text{and} \quad & \quad \text{or} \\
\end{align*}
\]
Scheduling and control logic

Modules (Current state)

\[ \begin{align*}
\pi_1 \\
\delta_1 \\
\vdots \\
\pi_n \\
\delta_n 
\end{align*} \]

Rules

\[ \begin{align*}
\text{\textit{CAN\_FIRE}} \\
\pi_1 \\
\vdots \\
\pi_n 
\end{align*} \]

Scheduler

\[ \begin{align*}
\phi_1 \\
\vdots \\
\phi_n 
\end{align*} \]

“WILL\_FIRE”

Modules (Next state)

cond

action

Muxing
Synthesis Summary

Bluespec generates a *combinational hardware scheduler* allowing multiple enabled rules to execute in the same clock cycle

- The hardware makes a rule-execution decision on every clock (i.e., it is not a static schedule)
- Among those rules that CAN_FIRE, only a subset WILL_FIRE that is consistent with a Rule order

Since multiple rules can write to a common piece of state, the compiler introduces suitable muxing and mux control logic

- This is very simple logic: the compiler will not introduce long paths on its own (details later)
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? 
    - more on this later