Bluespec-4: Rule Scheduling and Synthesis

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Exploring microarchitectures

IP Lookup Module

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 \( \Rightarrow 15 \text{Mpps for 10GE} \)

Sparse tree representation

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 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 Aj on clock J
  - Data Dj arrives on clock J+N
  - If you don't "catch" Dj on clock J+N, it may be lost, i.e., data Dj+1 may arrive on clock J+1+N

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

---

Asynchronous RAMs

It's easier to work with an "asynchronous" block

---

Static code

```
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
```

---

Circular Pipeline Code

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

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
```

---
### Completion buffer

```verilog
text = 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

  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.
Hardware Elements

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

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

Flip-flops with Write Enables

- **OpSelect**
  - Add, Sub, AddU, ...
  - And, Or, Not, ...
  - GT, LT, EQ, ...
  - SL, SR, SRA, ...

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

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

$\pi(s)$ 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**

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

$\pi$ = enabling condition

$\delta$ = action signals & values

**Combining State Updates:**

- $\pi$’s from the rules that update $R$
- $\delta$’s from the rules that update $R$

**Combining State Updates**

- $\pi$’s from all the rules
- $\delta$’s from the rules that update $R$

Scheduler: Priority Encoder

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**

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

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

**Rule** $a$ and Rule $b$ 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 $ra < rb$

**Rule**

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

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

**Rule** $a$ and Rule $b$ are sequentially composable if

\[
\forall s . \pi_a(s) \land \pi_b(s) \Rightarrow \\
1. \pi_a(\delta_b(s)) \land \pi_b(\delta_a(s)) \\
2. \delta_a(\delta_b(s)) == \delta_b(\delta_a(s))
\]

---

### Multiple-Rules-per-Cycle Scheduler

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

**Rule**

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

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

**Rule** $a$ and Rule $b$ are sequentially composable if

\[
\forall s . \pi_a(s) \land \pi_b(s) \Rightarrow \\
1. \pi_a(\delta_b(s)) \land \pi_b(\delta_a(s)) \\
2. \delta_a(\delta_b(s)) == \delta_b(\delta_a(s))
\]

---

**Multi-rule operations such that**

\[
\phi_i \land \phi_j \Rightarrow R_i \text{ and } R_i \text{ are conflict-free or sequentially composable}
\]
Muxing structure

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

  - Conflict Free: \( \delta_1 \land \pi_1 \lor \neg \pi_2 \)
  - Sequentially composable: \( \pi_1 \land \neg \pi_2 \land \delta_2 \land \pi_2 \land \delta_2 \land \pi_2 \)

Scheduling and control logic

- Modules (Current state) and Modules (Next state)
- Rules: \( \pi_1 \rightarrow \delta_1 \rightarrow \cdots \rightarrow \phi_1 \)
- "CAN_FIRE" and "WILL_FIRE" states
- Scheduler
- Muxing
- Cond
- Action

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