

Bluespec Tutorial: Rule Scheduling and Synthesis

Michael Pellauer
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Based on material prepared by Bluespec Inc,
January 2005

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-1

Improving performance via scheduling

- ◆ Latency and bandwidth can be improved by performing more operations in each clock cycle
 - That is, by firing more rules per cycle
- ◆ Bluespec schedules all applicable rules in a cycle to execute, except when there are resource conflicts
- ◆ Therefore: Improving performance is often about resolving conflicts found by the scheduler

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-2

Viewing the schedule

- ◆ The command-line flag `-show-schedule` can be used to dump the schedule
- ◆ Three groups of information:
 - method scheduling information
 - rule scheduling information
 - the static execution order of rules and methods

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-3

Method scheduling info

- ◆ For each method, there is an entry like this:

```
Method: imem_get
Ready signal: 1
Conflict-free: dmem_get, dmem_put, start, done
Sequenced before: imem_put
Conflicts: imem_get
```

name of the method

expression for the ready signal
(1 for always ready)

conflict relationships
with other methods

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-4

Types of conflicts

- ◆ Conflict-free
 - Any methods which can execute in the same clock cycle as the current method, in any execution order
- ◆ Sequenced before
 - Any methods which can execute in the same clock cycle, but only if they sequence before the current method in the execution order
- ◆ Sequenced after
 - Any methods which can execute in the same clock cycle, but only if they sequence after the current method
- ◆ Conflicts
 - Any methods which cannot execute in the same clock cycle as this method

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-5

Rule scheduling info

- ◆ For each rule, there is an entry like this:

Rule: fetch
Predicate: the_bf.i_notFull_ && the_started.get
Blocking rules: imem_put, start

name of the rule
expression for the rule's condition
more urgent rules which can block the execution of this rule (more on urgency later)

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-6

Static execution order

- ◆ When multiple rules execute in a single clock cycle, they must *appear* to execute in sequence
- ◆ This execution sequence is fixed at compile-time. All rule conditions are evaluated in this order during every clock cycle
- ◆ The final part of the schedule output is this order

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-7

Urgency

- ◆ The compiler performs aggressive analysis of rule boolean conditions and is therefore aware of mutual exclusion (i.e., when it is impossible for two rules to be enabled simultaneously)
 - Thus, typically the compiler does not often need to choose between competing rules
 - The compiler produces informational messages about scheduling choices only where necessary

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-8

Viewing conflict information

- ◆ The `-show-schedule` flag will inform you that a rule is blocked by a conflicting rule
 - The output won't show you *why* the rules conflict
- ◆ The output will show you that one rule was sequenced before another rule
 - The output won't tell you whether the other order was not possible due to a conflict
- ◆ For conflict information, use the `-show-rule-rel` flag
 - See *User Guide section 8.2.2*

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-9

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 enabled, only one will fire
- ◆ Which one?
 - The compiler chooses (and informs you, during compilation)
 - The "descending_urgency" attribute allows the designer to control the choice

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-10

Demo Example 2: Concurrent Updates

- ◆ Process 0 increments register x;
- ◆ Process 1 transfers a unit from register x to register y;
- ◆ Process 2 decrements register y



```
rule proc0 (cond0);
  x <= x + 1;
endrule
```

```
rule proc1 (cond1);
  y <= y + 1;
  x <= x - 1;
endrule
```

```
rule proc2 (cond2);
  y <= y - 1;
endrule
```

```
(* descending_urgency = "proc2, proc1, proc0" *)
```

show what happens under different urgency annotations

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-11

Example2.bsv Demo

- ◆ Compile (bsc Example2.bsv)
- ◆ Generate Verilog (bsc -verilog -g mkExample2 Example2.bsv)
- ◆ Run in vcs (See lab3 handout)
- ◆ Examine WILL_FIRE
- ◆ -keep-fires (Examine CAN_FIRE)
- ◆ -show-schedule
- ◆ -show-rule-rel (See manual)

March 4, 2005 ◆ Changing the priorities to true?

BST-12

Conditionals and rule-splitting

◆ In Rule Semantics this rule:

```
rule r1 (p1);  
  if (q1) f.enq(x);  
  else   g.enq(y);  
endrule
```

rule r1 won't fire
unless both f and g
queues are not full!

◆ Is equivalent to the following two rules:

```
rule r1a (p1 && q1);  
  f.enq(x);  
endrule  
  
rule r1b (p1 && ! q1);  
  g.enq(y);  
endrule
```

but not quite because
of the compiler treats
implicit conditions
conservatively

March 4, 2005

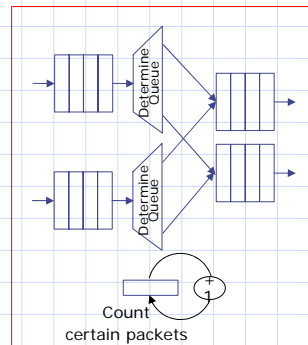
<http://csg.csail.mit.edu/6.884/>

BST-13

Demo rule splitting:

Example 3

```
(* descending_urgency = "r1, r2" *)  
// Moving packets from input FIFO i1  
rule r1;  
  Tin x = i1.first();  
  if (dest(x) == 1) o1.enq(x);  
  else               o2.enq(x);  
  i1.deq();  
  if (interesting(x)) c <= c + 1;  
endrule  
  
// Moving packets from input FIFO i2  
rule r2;  
  Tin x = i2.first();  
  if (dest(x) == 1) o1.enq(x);  
  else               o2.enq(x);  
  i2.deq();  
  if (interesting(x)) c <= c + 1;  
endrule
```



This example won't
work properly
without rule splitting

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-14

Example3.bsv Demo

- ◆ Compiling
- ◆ Examining FIFO signals, enables
- ◆ Examining conservative conditions
 - What are the predicates for R1, R2?
- ◆ -aggressive-conditions
 - What are the predicates now?
- ◆ -expand-if
 - Why can certain generated rules never fire?

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-15

Summary of performance tuning

- ◆ If the schedule of rules is not as you expected or desire, we have seen several ways to adjust the schedule for improved performance:
 - Remove rule conflicts by splitting rules
 - Change rule urgency
- ◆ Sometimes, an urgency warning or a conflict can be due to a mistake or oversight by the designer
 - A rule may accidentally include an action which shouldn't be there
 - A rule may accidentally write to the wrong state element
 - A rule predicate might be missing an expression which would make the rule mutually exclusive with a conflicting rule

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-16

Rule attributes

- ◆ We have already seen the **descending_urgency** attribute on rules
- ◆ There are two other useful attributes which can be applied to rules:
 - **fire_when_enabled**
 - **no_implicit_conditions**
- ◆ These attributes are assertions about the rule which bsc verifies
- ◆ Does not change generated RTL

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-17

fire_when_enabled

- ◆ Asserts that the rule will always execute when its condition is applicable
 - i.e., there are no (more urgent) conflicting rules
- ◆ Can be used to guarantee that a rule will handle some condition, by guaranteeing that the rule fires when the condition arises
- ◆ Examples:
 - To handle an unbuffered input on the interface
 - ◆ particularly in a time-based or synchronous module and particularly when the interface is "always_enabled"
 - To handle transient situations e.g., interrupts

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-18

no_implicit_conditions

- ◆ Asserts that rule actions do not introduce any implicit conditions
 - That the rule's condition is exactly as the user has written, and nothing more

- ◆ Can be combined with the attribute **fire_when_enabled** to guarantee that the rule will fire when its explicit condition is true

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-19

Matching to external interfaces

... the external interface may not use the same RDY/EN protocol as Bluespec; interface attributes are available to handle this situation ...

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-20

Interface attributes

- ◆ Useful attributes
 - `always_ready`
 - `always_enabled`
- ◆ Attributes attach to a *module*
- ◆ They apply to the interface provided by that module – when the module is synthesized
- ◆ The attributes apply to all methods in the interface

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-21

`always_ready`

This attribute has two effects:

- ◆ Asserts that the ready signal for all methods is True
 - It is an error if the tool cannot prove this
- ◆ Removes the associated port in the generated RTL module
 - Any users of the module will assume a value of True for the ready signals
 - No `RDY_method` signal are found

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-22

always_enabled

- ◆ Ties to True the enable signal for all action methods
 - If the method cannot be executed on every cycle (due to internal conflicts), bsc reports an error
- ◆ Removes the associated port in the generated RTL module
 - Any user of the module must execute the method on every cycle, or it is an error
- ◆ E.g. `EN_method` is assumed True and removed

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-23

Interface attributes

- ◆ These attributes are used to match externally-specified port lists which do not have RDY and EN wires
- ◆ Or for a synchronous module which should receive input on every cycle

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-24

Synchronous Binary Multiplier Interface

```
interface Design_IFC;
    method Action setInput (Bit#(16) x,
        Bit#(16) y, Bool start);
    method Bit#(32) prod();
    method Bool ready();
endinterface : Design_IFC

(* always_ready,always_enabled *)
module mkDesign (Design_IFC);
```

```
module mkDesign(clk,
                reset,
                setInput_x,
                setInput_y,
                setInput_start,
                prod,
                ready);
```

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-25

Demo Example 1:

```
module mkMult1 (Mult_ifc);
    Reg#(Tout) product <- mkReg (0);
    Reg#(Tout) d <- mkReg (0);
    Reg#(Tin) r <- mkReg (0);

    rule cycle (r != 0);
        if (r[0] == 1) product <= product + d;
        d <= d << 1;
        r <= r >> 1;
    endrule: cycle

    method Action start (Tin d_init, Tin r_init) if (r == 0);
        d <= zeroExtend(d_init);
        r <= r_init; product <= 0;
    endmethod

    method Tout result () if (r == 0);
        return product;
    endmethod
endmodule: mkMult1
```

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-26

Test bench for Example 1

```
module mkTest (Empty);
// arrays a, b contain the numbers to be multiplied and
// array ab contains the correct answers.

    Mult_ifc m      <- mkMult1();
    Reg#(Bool) busy <- mkReg(False);
    Reg#(int) i     <- mkReg(0);   Reg#(int) j <- mkReg(0);

    rule data_in (!busy);
        m.start (a[i], b[i]);
        i <= i+1;                busy <= True;
    endrule

    rule data_out (busy);
        Tout x = m.result();
        $display ("%0.h X %0.h = %0.h Status: %0.d",
                a[j], b[j], x, x==ab[j] );
        j <= j+1;                busy <= False;
    endrule
endmodule: mkTest
```

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-27

Example1.bsv Demo

- ◆ Compiling with `-u`
- ◆ The `(* synthesize *)` pragma
- ◆ Method RDY and EN
- ◆ Making the multiplier synchronous
- ◆ `(* always_ready *)`
- ◆ Altering the testbench
- ◆ `(* always_enabled *)`
- ◆ Examining the final verilog ports

March 4, 2005

<http://csg.csail.mit.edu/6.884/>

BST-28