Modular scheduling of guarded atomic actions

Muralidaran Vijayaraghavan, Nirav Dave, Arvind

vmurali@mit.edu, ndave@csl.sri.com, arvind@csail.mit.edu

MemoCODE 2013
A processor example

Module Processor (iReq, iRes, dReq, dRes)
redirect <- Fifo
f2d <- Fifo
d2e <- Fifo
rf <- RegFile
sb <- Scoreboard
fetch<-Fetch(iReq, iRes, f2d.enq, redirect.deq, redirect.first)

Instantiate other modules

A module with interface parameters
Module in traditional HDL vs Bluespec

• Modules in traditional HDL (e.g., Verilog) represent FSMs with input/output wires.
  – Composition of FSMs is easy
  – But timing refinements is hard
• Modules in Bluespec have object-like methods interface
  – Permits flexible modular/timing refinements
  – But separate compilation in the presence of interface parameters has issues
Why separate compilation?

• Improves compilation times
• Separate compilation/synthesis of modules with external parameters is sometimes inevitable
  – For example, DRAM cannot be instantiated inside a processor!
The goals of this work

• Spell out the compositional interface generated by compiling a module
• Give a modular scheduling scheme

All this in the presence of interface parameters
Outline

• Motivation

• Background
  – KBS0 – Bluespec without modules
  – KBS1 = KBS0 + modules

• KBS2 = KBS1 + method parameters for modules
KBSO – Bluespec without modules

[Hoe, TCAD 2004]

• A system is designed as a collection of rules or (guarded) atomic actions each of which changes the state, i.e. collection of registers, of the system

• One-rule-at-a-time semantics: The overall state change sequence can be understood as if each of the rules executed one by one

![Diagram showing state transitions with rules Ri and Rj over time]

Initial State  \[\rightarrow\] \(r_i\) \[\rightarrow\] Next state \(r_j\) \[\rightarrow\] Next state ...
Need to execute multiple rules concurrently

• For efficiency, multiple rules are executed by a scheduler in each clock cycle
  – For example, pipelining

• One-rule-at-a-time semantics should not be violated
Executing multiple rules in one clock cycle

Sequential composition preserves the one-rule-at-a-time semantics but generally increases the critical combinational path.

Parallel composition does not create longer combinational paths but may not preserve the one-rule-at-a-time semantics.

Scheduler only permits parallel composition!
Conflict analysis in KBSO

- Rules $r_1$ and $r_2$ can be scheduled concurrently (as in parallel composition) $\iff$ either $r_1 < r_2$ or $r_2 < r_1$
  - where $r_a < r_b \iff r_b$ does not read or write any register that $r_a$ writes

\[
\begin{align*}
\text{rule } ra & \ x := x+1 \\
\text{rule } rb & \ y := y+2 \\
\text{rule } ra & \ x := y+1 \\
\text{rule } rb & \ y := x+2
\end{align*}
\]
Outline

• Motivation

• Background
  – KBS0 – Bluespec without modules
  – KBS1 = KBS0 + modules

• KBS2 = KBS1 + method parameters for modules
KBS0 + Modules = KBS1

[Rosenband, DAC 2004]
KBS1 Example: A FIFO

Module Fifo
valid <- Reg(False)
data <- Reg(undefined)
method enq(x) = if !valid 
    (data := x | valid := True)
method deq() = if valid
    valid := False
method first() = if valid
    data

Module Top
f <- Fifo
rule r1 f.enq(20)
rule r2 f.deq()

Inlining semantics:
Module Top
f_valid <- Reg(False)
f_data <- Reg(undefined)

rule r1 if !f_valid
    ( f_data := 20
    | f_valid := True)

rule r2 if f_valid
    f_valid := False

Can rules r1 and r2 be scheduled concurrently?
Conflict Matrix for defined methods

- When a module is compiled, it exports the conflict information between every pair of its defined methods \( (CM_{DM}) \)
- Conflict information can be viewed as a lattice:
  - Conflicting (C) = {}
  - Conflict-Free (CF) = {<, >}
- \( CM_{DM} \) is predefined for primitive modules
- \( CM_{DM} \) is calculated by the compiler for user defined modules

<table>
<thead>
<tr>
<th>( CM_{DM} ) (row, column)</th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>CF</td>
<td>{&lt;}</td>
</tr>
<tr>
<td>write</td>
<td>{&gt;}</td>
<td>C</td>
</tr>
</tbody>
</table>

Register
**CM<sub>DM</sub> for FIFO**

- Determine methods called by \texttt{enq, deq}
  - Methods called by \texttt{enq}: \{\texttt{data.write, valid.read, valid.write}\} = \texttt{enqS}
  - Methods called by \texttt{deq}: \{\texttt{valid.read, valid.write}\} = \texttt{deqS}

- \(\text{CM}_{\text{DM}}(\texttt{enq}, \texttt{deq}) = \)
  
  Least Upper Bound of
  \[
  \{\text{Conflict}(p, q) \mid (p, q) \text{ is in } (\texttt{enqS} \times \texttt{deqS})\}
  \]

\[
\text{Conflict(data.\*, valid.\*)} = \text{CF}
\]

Methods in different modules (or registers) share no state and hence do not conflict

\[
\text{Conflict(valid.write, valid.write)} = \text{C}
\]

\[\Rightarrow \text{CM}_{\text{DM}}(\texttt{enq}, \texttt{deq}) = \text{C}\]
Outer scheduler schedules rules of the outer module, which calls methods of the instance.

Instance scheduler schedules its internal rules after knowing which methods are being called, to avoid its internal rules from conflicting with the outer module’s rules.
Outline

• Motivation
• Background
  – KBS0 – Bluespec without modules
  – KBS1 = KBS0 + modules
• KBS2 = KBS1 + method parameters for modules
KBS1 + method parameters in modules = KBS2

Methods defined in different instances are no longer CF!

If f1 calls p1, since ap1 is instantiated with m2.f2, m1.f1 and m2.f2 interact!
Example of KBS2 module

Do rules r1 and r2 conflict?  
If g2 calls p2 in M1 and g3 calls p3 in M2?

Yes, since m1.p2 = f.enq and m2.p3 = f.deq

Module compilation produces FPU(g) for every defined method g
(Formal Parameters Used)
Calculating $\text{CM}_{\text{DM}}$

$\text{CM}_{\text{DM}}(h_1, h_2) = \text{Conflict}(m_1.g_2, m_2.g_3)$

$= C$
Calculating $\text{CM}_{\text{DM}}$

$\text{CM}_{\text{DM}}(h_1, h_2) = \text{Conflict}(\text{m1.g1, pt2})$

$= \text{Conflict}(\text{m1.p1, pt2})$

$= \text{Conflict}(\text{pt1, pt2})$

$= \text{CM}_{\text{FP}}(\text{pt1, pt2})$

**Designer has to supply $\text{CM}_{\text{FP}}(\text{pt1, pt2})$**
Outer scheduler prevents scheduling of some of the passed-in parameters used in the internal rules of an instance to preserve one-rule-at-a-time semantics.
Conclusion

• Modular refinement is of limited value without separate compilation of modules
• Interface parameters are essential for expressing complex designs in a natural way in Bluespec

Thanks!
Questions?
A system is designed as a collection of *rules* or *(guarded) atomic actions* each of which changes the state of the system.

\[
\begin{align*}
\text{rule } ::= & \text{ rule name } a \\
\text{ } & \\
\text{ } & e ::= r \\
\text{ } & | c \\
\text{ } & | t \\
\text{ } & | \text{op}(e, e) \\
\text{ } & | \text{let } t = e \text{ in } e \\
\text{a } ::= & r := e \\
\text{ } & | a \mid a \\
\text{ } & | \text{if } e \text{ then } a \\
\text{ } & | \text{let } t = e \text{ in } a
\end{align*}
\]
Guard Lifting Axioms
without Let-blocks

• All the guards can be “lifted” to the top of a rule
  – \((a_1 \text{ when } p) \mid a_2 \Rightarrow (a_1 \mid a_2) \text{ when } p\)
  – \(a_1 \mid (a_2 \text{ when } p) \Rightarrow (a_1 \mid a_2) \text{ when } p\)
  – \(\text{if } (p \text{ when } q) \ a \Rightarrow (\text{if } (p) \ a) \text{ when } q\)
  – \(\text{if } (p) (a \text{ when } q) \Rightarrow (\text{if } (p) \ a) \text{ when } (q \ | | \ !p)\)
  – \((a \text{ when } p_1) \text{ when } p_2 \Rightarrow a \text{ when } (p_1 \ & \ & p_2)\)
  – \(\text{m.g}_B(e \text{ when } p) \Rightarrow \text{m.g}_B(e) \text{ when } p\)

similarly for expressions ...
  – Rule r \((a \text{ when } p) \Rightarrow \text{Rule r } (\text{if } (p) \ a)\)

We will call this guard lifting transformation WIF, for when-to-if

A complete guard lifting procedure also requires rules for let-blocks
An example: FIFO module

Module Fifo
valid <- Reg(False)
data <- Reg(undefined)
method enq(x) =
  if !valid
    ( data := x
     | valid := True)
method deq() =
  if valid
    valid := False
method first() =
  if valid
    data