# Concurrency properties of BSV methods and rules

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology



## One-rule-at-a-time semantics

- Given a program with a set of rules {rule $r_i$ $a_i$} and an initial state $S_0$ , S is a legal state if and only if there exists a sequence of rules $r_{j1}$,…., $r_{jn}$ such that $S= a_{jn}(\ldots(a_{j1}(S_0))\ldots)$

# Concurrent execution of two rules

- Concurrent execution of two rules, rule $r_1$ $a_1$ and rule $r_2$ $a_2$, means executing a rule whose body looks like ($a_1$; $a_2$), that is a rule which is a parallel composition of the actions of the two rules with the following restrictions to preserve the one-rule-at-a-time semantics:
  - Either $\forall S.\ (a_1;\ a_2)(S) = a_2(a_1(S))$
    or $\forall S.\ (a_1;\ a_2)(S) = a_1(a_2(S))$



# Concurrent scheduling of rules

- rule $r_1$ $a_1$ to rule $r_n$ $a_n$ can be scheduled concurrently, preserving one-rule-at-a-time semantics, if and only if there exists a permutation $(p_1,\ldots,p_n)$ of $(1,\ldots,n)$ such that
  - $\forall S.\ (a_1;\ldots;a_n)(S) = a_{pn}(\ldots(a_{p1}(S))$

How does a compiler decide which rules can be scheduled concurrently

Related question: what is a legal rule?

# Well formed actions (rules)

*informally*

- No possibility of *double write error.* In general, no double use of a method
  - The only exception is a value method without arguments, e.g., register read, fifo.first
- *No combinational cycles.* In general it means that it should be possible to put all the method calls in a sequential order consistent with their module definitions and data dependences



# Are these actions legal?

- x <= e1; x <= e2;
- x <= e1; if( p ) x <= e2;
- if( p ) x <= e1; else x <= e2;
- x[0] <= x[1]

- x[0] <= y[1]; y[0] <= x[1]
- if (x[1]) x[0] <= e;

# A critical example

- Example 1: Rule exchange x <= y; y <= x
- Example 2: Rule exchange' f() ; g()

  where Module foo

  register x, y etc

  method f() = (x <= y);

  method g() = (y <= x);
- Example 3:

  Rule fr;  x <= y;

  Rule gr; y <= x;

Is exchange legal?

Is exchange' legal?

Can rules fr and gr be executed together?



# Primitive module: Register

- read and write can happen in the same atomic action and don't affect each other; the effect of write is not visible until the atomic action has completed
- Legality of an action depends upon the permitted intra-rule behaviors
- Concurrent scheduling depends upon the inter-rule behavior

| | reg.r | reg.w |
|---|---|---|
| reg.r | CF | CF |
| reg.w | CF | C |

Intra-rule behavior

| | reg.r | reg.w |
|---|---|---|
| reg.r | CF | < |
| reg.w | > | C |

Inter-rule behavior

# Primitive module: EHR

Intra-rule behavior

| | EHR.r0 | EHR.w0 | EHR.r1 | EHR.w1 |
|---|---|---|---|---|
| EHR.r0 | CF | | | |
| EHR.w0 | | C | | |
| EHR.r1 | | | CF | |
| EHR.w1 | | | | C |

Inter-rule behavior

| | EHR.r0 | EHR.w0 | EHR.r1 | EHR.w1 |
|---|---|---|---|---|
| EHR.r0 | CF | | | |
| EHR.w0 | | C | | |
| EHR.r1 | | | CF | |
| EHR.w1 | | | | C |



## Intra-rule analysis
# "Happens before" (<) relation

- "happens before" relation between the methods of a module governs how the methods behave when called by a rule, action, method or exp
  - f < g : f happens before g (g cannot affect f within an action)
  - f > g : g happens before f
  - C : f and g conflict and cannot be called together
  - CF : f and g are conflict free and do not affect each other
- This relation is defined as a conflict matrix (CM) for the methods of primitive modules like registers and EHRs and derived for the methods of all other modules

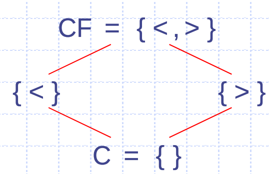
Inter-rule analysis

# "Happens before" (<) relation

- "happens before" relation between the methods of a module governs how the methods behave when called by two rules or methods
  - f < g : parallel: but g may affect f in a seq execution
  - f > g : parallel: but f may affect g in a seq execution
  - C : not parallel: f and g conflict
  - CF : parallel: f and g are conflict free and unrelated
- This relation is defined as a conflict matrix (CM) for the methods of primitive modules like registers and EHRs and derived for the methods of all other modules



# Conflict ordering




- This permits us to take intersections of conflict information, e.g.,
  - $\{>\}\cap\{<,>\} = \{>\}$
  - $\{>\}\cap\{<\} = \{\}$

# Some definitions

- mcalls(x) is the set of method called by x
- mcalls(x) $<_s$ mcalls(y) means
  a $\in$ mcalls(x), b $\in$ mcalls(y) =>
  (a < b) | (a CF b) | (a ME b)

  To be explained later
- we often overload < and use it in place of $<_s$



# Deriving the Conflict Matrix (CM) of a module

- Let g1 and g2 be the two methods defined by a module, such that

  mcalls(g1)={g11,g12...g1n}

  mcalls(g2)={g21,g22...g2m}
- conflict(x,y) = if x and y are methods of the same module then CM[x,y] else CF
- Derivation
  - CM[g1,g2] = conflict(g11,g21) $\cap$ conflict(g11,g22) $\cap$...
    $\cap$ conflict(g12,g21) $\cap$ conflict(g12,g22) $\cap$...
    ...
    $\cap$ conflict(g1n,g21) $\cap$ conflict(g12,g22) $\cap$...

Compiler can derive the CM for a module by starting with the innermost modules in the module instantiation tree

## Data-dependence constraints

- if (e) a $\Rightarrow$ NewDependences(mcalls(e), mcalls(a))
- m.g(e) $\Rightarrow$ NewDependences(mcalls(e), {m.g})
- t = e ; a $\Rightarrow$ NewDependences(mcalls(e) , {f | f $\in$ mcalls(a) & f uses t})

An action is *legal* if the data-dependence imposed constraints together with method definition constraints can be placed in a total order (no cycles)



## Real legal-rule analysis is more complicated: Predicated calls

- The analysis we presented would reject the following rule because of method conflicts

  if (p) m.g(e1) ; if (!p) m.g(e2)

- We need to keep track of the predicates associated with each method call

  m.g is called with predicates p and !p which are disjoint – therefore no conflict

## Mutually exclusive actions

- a1 and a2 are *mutually exclusive* if in all possible states the effect of one of them is "no action"
  - example: a1 = if (p) b1 a2 = if (!p) b2
- Mutual exclusivity of actions and methods reduces the number of conflicts at the cost of complicating the analysis
- In computing the conflict matrix (CM) one can ignore entries corresponding to mutually exclusive methods
- In determining the legality of an action (rule) one can ignore the ordering constraints between mutually exclusive sub-actions



## Some examples to intra-rule and inter-rule analysis

# CM for One-Element FIFO

```
module mkPipelineFifo(Fifo#(1, t)) provisos(Bits#(t, tSz));
  Reg#(t) d <- mkRegU;
  Reg#(Bool) v <- mkReg(False);

  method Action enq(t x) if (!v);
    d <= x;
    v <= True;
  endmethod

  method Action deq if (v);
    v <= False;
  endmethod

  method t first if (v);
    return d;
  endmethod
endmodule
```

mcalls(enq) =

mcalls(deq) =

mcalls(first) =

Notice enq and deq are mutually exclusive



# CM for One-Element FIFO

mcalls(enq) = {v.r, d.w, v.w}
mcalls(deq) = {v.r, v.w}
mcalls(first) = {v.r, d.r}

Intra-rule CM

| | enq | deq | first |
|---|---|---|---|
| enq | C | ME | ME |
| deq | ME | C | CF |
| first | ME | CF | CF |

Inter-rule CM

| | enq | deq | first |
|---|---|---|---|
| enq | C | ME | ME |
| deq | ME | C | > |
| first | ME | < | CF |

# CM for One-Element Pipeline FIFO

```
module mkPipelineFifo(Fifo#(1, t)) provisos(Bits#(t, tSz));
  Reg#(t) d <- mkRegU;
  Ehr#(2, Bool) v <- mkEhr(False);

  method Action enq(t x) if (!v[1]);
    d <= x;
    v[1] <= True;
  endmethod

  method Action deq if (v[0]);
    v[0] <= False;
  endmethod

  method t first if (v[0]);
    return d;
  endmethod
endmodule
```

mcalls(enq) =

mcalls(deq) =

mcalls(first) =



# CM for One-Element Pipeline FIFO

mcalls(enq) = {v.r1, d.w, v.w1}
mcalls(deq) = {v.r0, v.w0}
mcalls(first) = {v.r0, d.r}

CM[enq,deq] = conflict[v.r1,v.r0] ∩ conflict[v.r1,v.w0]
∩ conflict[d.w,v.r0] ∩ conflict[d.w,v.w0]
∩ conflict[v.w1,v.r0] ∩ conflict[v.w1,v.w0]

Intra-rule derivation

| | enq | deq | first |
|---|---|---|---|
| enq | C | | |
| deq | | C | |
| first | | | CF |

Intra-rule

| | enq | deq | first |
|---|---|---|---|
| enq | C | | |
| deq | | C | |
| first | | | CF |

Inter-rule

# CM for One-Element Bypass FIFO

```
module mkBypassFifo(Fifo#(1, t)) provisos(Bits#(t, tSz));
  Ehr#(2, t) d <- mkEhr(?);
  Ehr#(2, Bool) v <- mkEhr(False);

  method Action enq(t x) if !v[0];
    d[0] <= x;
    v[0] <= True;
  endmethod

  method Action deq if v[1];
    v[1] <= False;
  endmethod

  method t first if v[1];
    return d[1];
  endmethod
endmodule
```

mcalls(enq) =

mcalls(deq) =

mcalls(first) =



# CM for One-Element Bypass FIFO

mcalls(enq) = {d.w0, v.w0, v.r0}
mcalls(deq) = {v.r1, v.w1}
mcalls(first) = {v.r1, d.r1}

CM[enq,deq] =

Intra-rule derivation

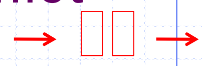
| | Enq | Deq | First |
|---|---|---|---|
| Enq | C | | |
| Deq | | C | |
| First | | | CF |

Intra-rule

| | Enq | Deq | First |
|---|---|---|---|
| Enq | C | | |
| Deq | | C | |
| First | | | CF |

Inter-rule

# CM for Two-Element Conflict-free FIFO




```
module mkCFFifo(Fifo#(2, t)) provisos(Bits#(t, tSz));
  Ehr#(2, t) da <- mkEhr(?);
  Ehr#(2, Bool) va <- mkEhr(False);
  Ehr#(2, t) db <- mkEhr(?);
  Ehr#(2, Bool) vb <- mkEhr(False);

  rule canonicalize(vb[1] && !va[1]);
    da[1] <= db[1];
    va[1] <= True; vb[1] <= False; endrule

  method Action enq(t x) if va[0];
    db[0] <= x; vb[0] <= True; endmethod
  method Action deq if !vb[0];
    va[0] <= False endmethod
  method t first if !vb[0];
    return da[0]; endmethod
endmodule
```

Derive the CM



# Extending CM to rules

## using inter-rule CMs

- CM between two rules is computed exactly the same way as CM for the methods of a module
  - Given rule r1 a1 and rule r2 a2 such that

    mcalls(a1)={g11,g12...g1n}

    mcalls(a2)={g21,g22...g2m}
  - Compute
    - Conflict(x,y) = if x and y are methods of the same module then CM[x,y] else CF
    - CM[r1,r2] = conflict(g11,g21) ∩ conflict(g11,g22) ∩...
      ∩ conflict(g12,g21) ∩ conflict(g12,g22) ∩...
      ...
      ∩ conflict(g1n,g21) ∩ conflict(g12,g22) ∩...
  - Conflict relation is not transitive
    - r1 < r2, r2 < r3 does not imply r1 < r3

# Using CMs for concurrent scheduling of rules

Two rules that are conflict free can be scheduled together without violating the one-rule-at-a-time semantics. In general, use the following theorem

*Theorem:* Given a set of rules {rule $r_i$ $a_i$}, if there exists a permutation {$p_1$, $p_2$ … $p_n$} of {1..n} such that

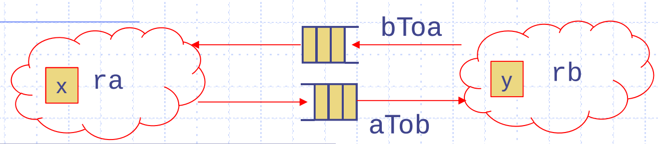
$\forall$ i < j. CM($a_{pi}$, $a_{pj}$) is CF or < or ME

then the rules $r_1$, $r_2$ … $r_n$ can be scheduled concurrently with the effect $\forall$ i, j. $r_{pi}$ < $r_{pj}$

A compiler can perform the analysis needed for concurrent scheduling of rules James Hoe, 2000



# Scheduling constraints due to multiple modules




Can ra and rb be scheduled concurrently?

```
rule ra;
  aTob.enq(fa(x));
  x <= ga(bToa.first);
  bToa.deq;
endrule
rule rb;
   y <= gb(aTob.first);
   aTob.deq;
   bToa.enq(fb(y));
Endrule
// assume both fifos are
not empty
```

fifos are initially empty

| aTob fifo | bToa fifo | Concurrent scheduling? |
|---|---|---|
| CF | CF | |
| CF | pipeline | |
| CF | bypass | |
| pipeline | CF | |
| pipeline | pipeline | |
| pipeline | bypass | |
| bypass | CF | |
| bypass | pipeline | |
| bypass | bypass | |