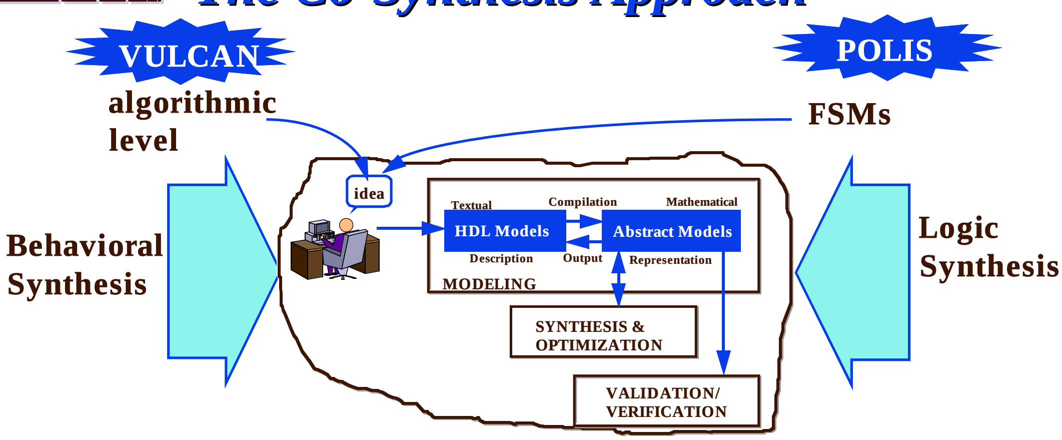
H/S Codesign:
A CAD Perspective

Margarida F. Jacome

The University of Texas at Austin

The Co-Synthesis Approach



The hardware software co-design problem is posed as an "evolution" of *existing synthesis methods*

The Cosynthesis Approach

- Working hypothesis: the overall system can be modeled consistently and be partitioned (either manually or automatically) into hardware and software components.
  - hardware components
    - performance
    - implemented using existing hardware synthesis tools
  - software components
    - low cost, flexibility
    - generated automatically (software compilation)
  - interfaces and synchronization

H/S Codesign: Research Issues

- Models and Specification Languages
- Design Space Exploration, Estimation, Partitioning
- Co-simulation/Verification
- Software, Hardware, and Interface Synthesis
- Scheduling, Real-time Operating Systems

Co-Synthesis

- Vulcan (Stanford - DeMicheli et al)
- Polis (UC Berkeley - Vicentelli et al)

Vulcan

- Leverages research in behavioral synthesis
  - modeling: flow graphs (⇐ sequencing graphs)
  - scheduling techniques (⇐ relative scheduling)
  - automatic path to synthesis (Olympus)
- Automatic partitioning
- Deterministic constraint analysis

Vulcan

- Modeling
- Constraint Analysis
- Software and Runtime Environment
- Target Architecture - H/S Interface
- Partitioning

Example - Algorithmic Description

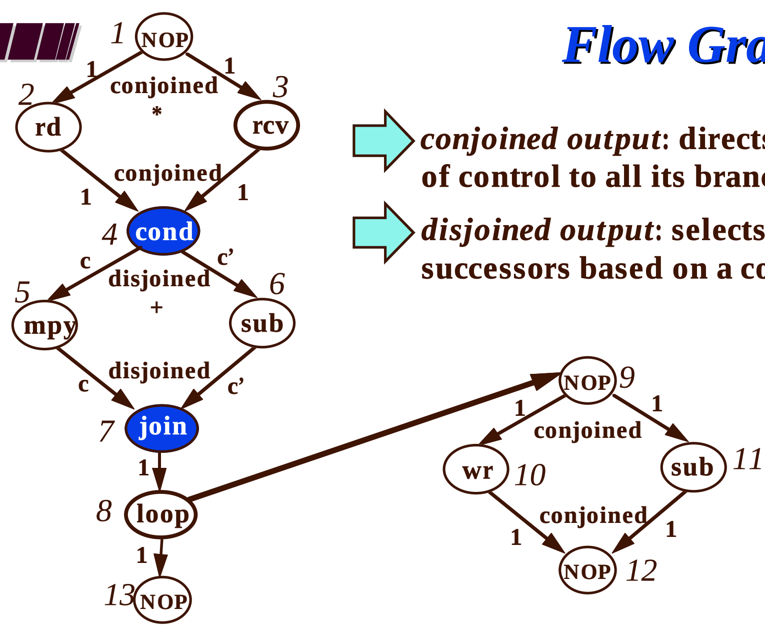
```
process example (a, b, c)
  in port a[8],
  in channel b[8];
  out port c[8];
{
  boolean x[8], y[8], z[8];
  x = read(a);
  y = receive(b);
  if (x > y)
      z = x - y;
  else
      z = x * y;
  while (z >= 0)
      { write c = y;
        z = z - 1; }
}
```

enables ***blocking*** *of the read operation based on the availability of data on the channel*

Flow Graph

- Hierarchical control/data-flow graph:
  - control flow primitives (*iteration* and *model call*) modelled through hierarchy
- *Acyclic*
  - models a partial order of tasks/operations
  - acyclic dependencies suffice ⇒ iteration is modeled outside the graph
- *Polar*
  - source and sink vertices model *No-Operations*

Flow Graph



- ***conjoined output***: directs the flow of control to all its branches
- ***disjoined output***: selects one of the successors based on a condition

Flow Graph

nodes ⇒

- *no-op*: no operation
- *cond*: conditional fork
- *join*: conditional join
- *op-logic*: logical operations
- *op-arithmetic*: arithmetic operations
- *op-relational*: relational operations
- *op-io:* I/O operations
- *wait*: wait on a signal variable (synchronization)
- *link*: hierarchical operations
  - *call*: procedure call (invocation times = 1)
  - *loop*: iteration (invocation times ≥ 1)

System Model

System Model: $\Phi = \{G_1^*, G_2^*, \ldots, G_n^*\}$
where

G_i^* : process graph model G_i and all the flow graphs hierarchically linked to G_i.

☆ Flow graph models can common to more than one hierarchy ⇒ *shared models*

Margarida Jacome - UT Austin - Spring 97

Implementation Attributes

- Implementation, *I(G)*, of a graph model *G* :
  - assignment of *delays* and *size* properties to *operations* in *G*
  - choice of a *runtime scheduler*, γ, that enables the execution of *source* operations in G

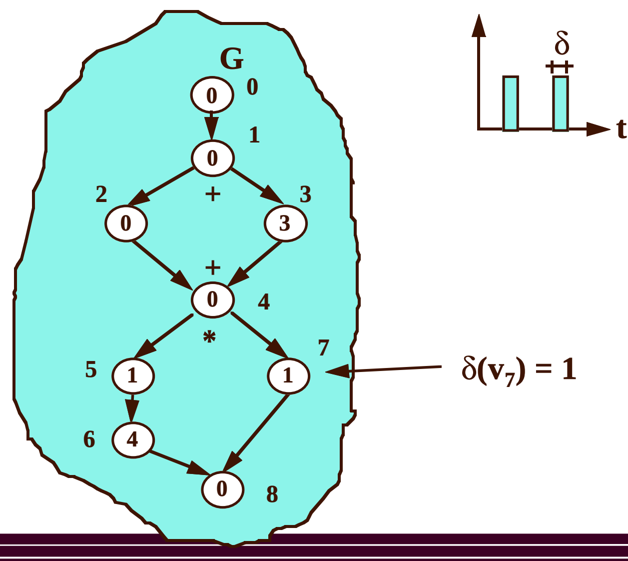
γ ⇨ enables source operation once:
- ✩ "top-level" graphs: the sink operation completes
- ✩ conditionally invoked graphs: the graph enabling condition is TRUE

Timing Properties

- Operation delay
- Graph Latency
- Rate of Execution (operations)

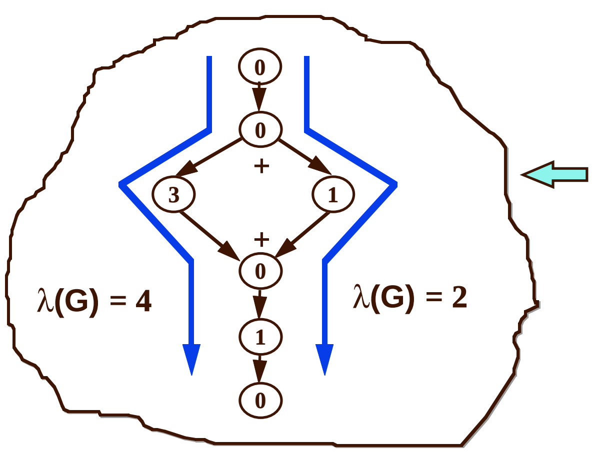
Margarida Jacome - UT Austin - Spring 97

Operation Delay



Latency

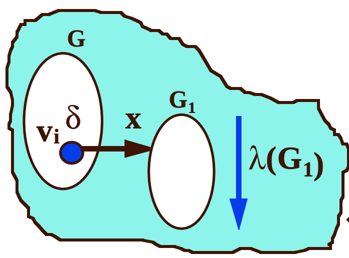
- <u>Latency</u>, $\lambda(G)$: execution delay of $G \Rightarrow$ $\lambda_k(G) = t_k(v_n) - t_k(v_0)$



latency of a non-hierarchical flow graph may be varible (conditional paths)

Execution Delay of Link Vertices

- $\delta(v_i) = \lambda(G_1) \bullet x$
  - can be
    - ⇨ variable
    - ⇨ unbounded (loop vertices with unbounded indices)



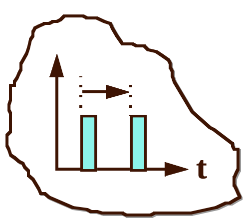
⇦ *Link* vertices: `call` and/or `loop` (point to other flow graphs in the hierarchy)

Rate of Execution (operations)

- assuming a *synchronous* execution model with cycle time τ,
  - ⇨ the rate of execution at invocation *k* of operation v_i is given by the time interval between its current and previous execution

$$\rho_i(k) := \frac{1}{t_k(v_i) - t_{k-1}(v_i)} \quad (\text{sec}^{-1})$$

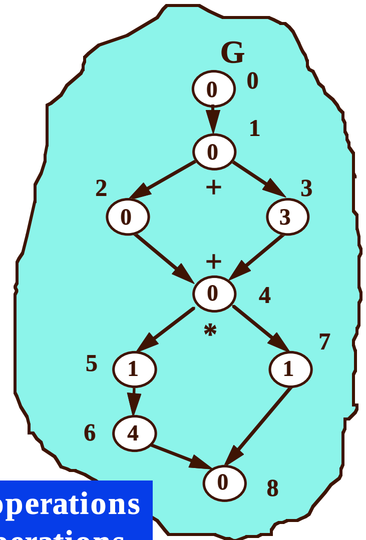
$$= \frac{\tau}{t_k(v_i) - t_{k-1}(v_i)} \quad (\text{cycle}^{-1})$$



Timing Properties

- Operation delay
- Graph Latency
- Rate of Execution (operations)

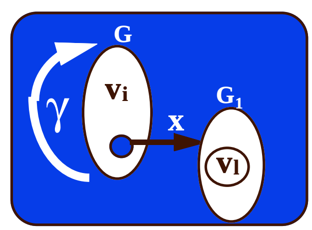
fixed, variable, bounded/unbounded



Data dependent loop and synchronization operations are termed *non-deterministic delay* or *ND* operations

Scheduling

- For each invocation of a flow graph model, an operation is invoked zero, one, or many times depending upon its *position on the hierarchy* of the flow model



The execution times $t_k(v)$ of an operation v are determined by two separate mechanisms

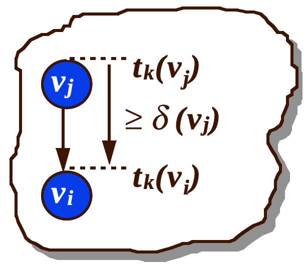
- **The runtime scheduler, γ**
  - **determines the invocation time of flow graphs**
- **The operation scheduler, Ω**

Scheduling of Operations

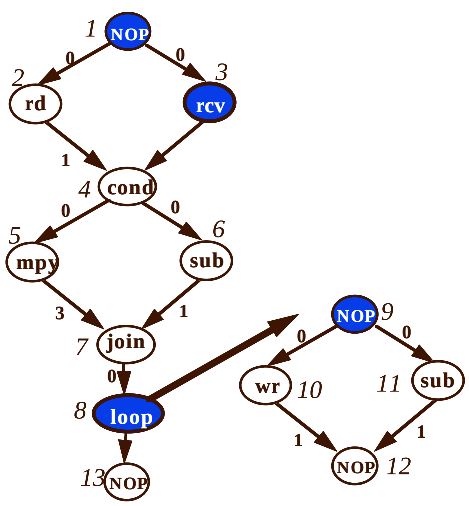
- Given a graph model G = (V, E), the selection of a *schedule* refers to the choice of a function Ω that determines the *start time of operations* such that

$$t_k(v_i) \ge \max_{j:(v_j, v_i) \in E} [\, t_k(v_j) + \delta(v_j) \,]$$

is satisfied for each invocation k>0 of operations v_i and v_j



Modified Relative Schedule



Vertex	Relative Offset v_1	v_3	v_8	v_9
v_1	--	--	--	
v_2	0	--	--	
v_3	0	--	--	
v_4	1	0	--	
v_5	1	0	--	
v_6	1	0	--	
v_7	(2,4)	(1,3)	--	
v_8	(2,4)	(1,3)	--	
v_9	(2,4)	(1,3)	--	
v_{10}	--	--	--	0
v_{11}	--	--	--	0
v_{12}	--	--	--	1
v_{13}	--	--	0	--

Margarida Jacome - UT Austin - Spring 97

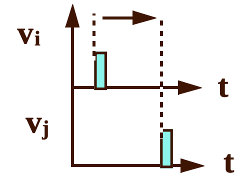
Vulcan

- Modeling
- Constraint Analysis
- Software and Runtime Environment
- Target Architecture - H/S Interface
- Partitioning

Timing Constraints

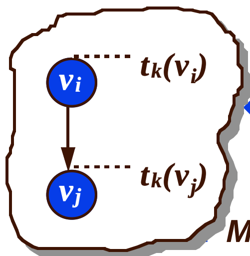
- *Operation delay* constraints
  - *unary*: bounds on the delay of an operation
  - *binary*: bounds on the delay between the starting time of two operations
- Execution rate constraints

Binary Delay Constraints



- *Minimum timing constraint*, $l_{ij} \geq 0$ from operation vertex v_i to v_j is defined as

$$t_k(v_j) \geq t_k(v_i) + l_{ij}$$



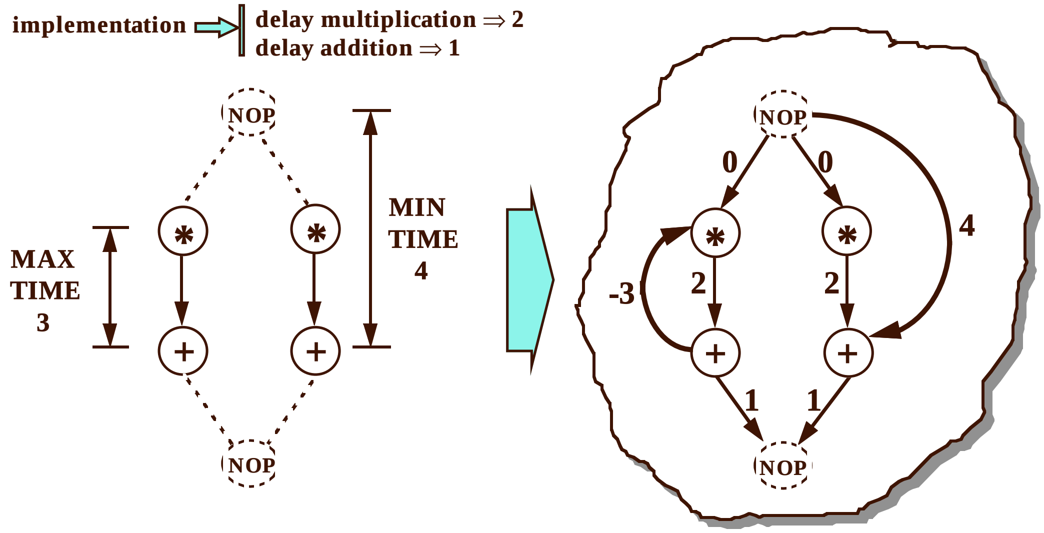
- sequencing dependencies between operations induce default minimum timing constraints

Maximum timing constraint, $u_{ij} \geq 0$ from operation vertex v_i to v_j is defined as

$$t_k(v_j) \leq t_k(v_i) + u_{ij}$$

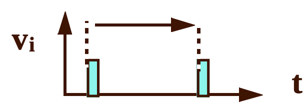
Constraint Graph

implementation ⇒ delay multiplication ⇒ 2
delay addition ⇒ 1



Timing Constraints

- Operation delay constraints
- Execution rate constraints
  - constraints on the interval of time between successive executions of an operation



Data Rate Constraints

- *Minimum data rate constraint, r_i* (cycles^{-1}) on operation v_i : **lower bound** on the execution rate of v_i

$$\rho_{vi}(k) \geq r_i \qquad \forall k > 0 \qquad \text{[min rate]}$$

$$\Rightarrow \; t_k(v_i) - t_{k-1}(v_i) \leq \tau . r_i^{-1} \qquad \forall k > 0$$

- *Maximum data rate constraint, R_i* (cycles^{-1}) on operation v_i: **upper bound** on the execution rate of v_i

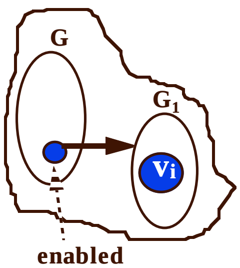
$$\rho_{vi}(k) \leq R_i \qquad \forall k > 0 \qquad \text{[max rate]}$$

$$\Rightarrow \; t_k(v_i) - t_{k-1}(v_i) \geq \tau . R_i^{-1} \qquad \forall k > 0$$

Relative Execution Rate Constraint

- *relative rate of execution* of v_i with respect to G:
  - ⇒ constraint on the rate of execution of v_i when G is continuously enabled and executing

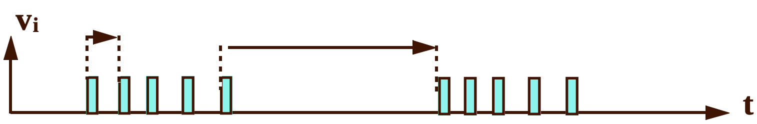
$$r_i^G \le \rho_{v_i}(k) \le R_i^G \qquad \forall k > 0$$



and, there exists an execution, j, of G such that

$$t_j(v_0(G)) \le t_{k-1}(v_i) \le t_k(v_i) \le t_j(v_N(G))$$

v_0 and v_N: source and sink nodes of G



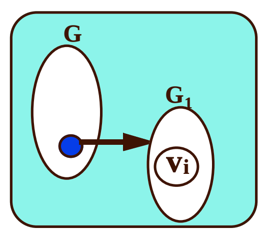
Ex.: Specification of Rate Constraints

```
process example (a,b,c)
      in port a[8],b[8];
      out port c[8];
{
       boolean x[8],y[8],z[8],w[8];
       tag A;
       x = read(a);
       y = read(b);
       z = x * y;
       w = x + y;
       while(z >= 0) {
          while(w >= 0)  {
A:                   write c = y;
                     w = w - 1; }
       z = z - w;
       write c = z;    }
 attribute “constraint minrate of A = 100 cycles/sample”
 attribute “constraint minrate 0 of A = 1 cycles/sample”
 attribute “constraint minrate 1 of A = 10 cycles/sample”
}
```

1

0

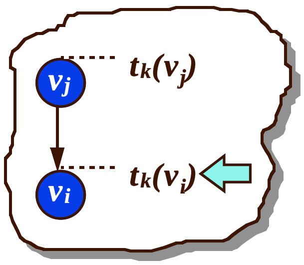
r = 0.01 per cycle



<u>relative</u> min constraints -- indexed by the corresponding loops

Timing Constraints and Scheduling

- Given a scheduling function, a timing constraint is considered *satisfied* if
  - the operation starting times determined by the scheduling function satisfy the inequalities



$t_k(v_j) \geq t_k(v_i) + l_{ij}$ **[min delay]**

$t_k(v_j) \leq t_k(v_i) + u_{ij}$ **[max delay]**

$\rho_{v_i}(k) \leq R_i$ **[max rate]**

$\rho_{v_i}(k) \geq r_i$ **[min rate]**

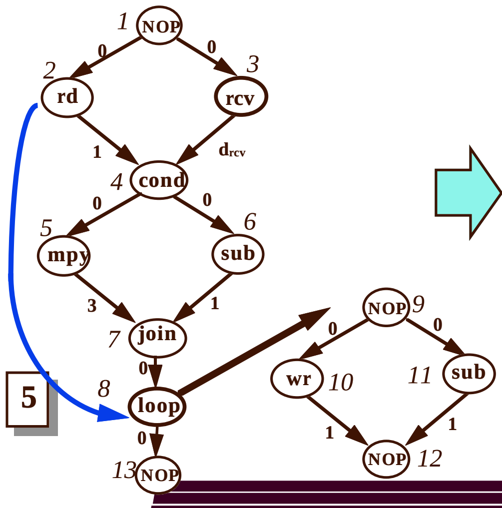
Satisfiability - Delay Constraints

- **A minimum delay constraint is always satisfiable**

$$\theta_{v_j}(v_i) \geq \max(l(v_j, v_j), l_{ij})$$

- **A maximum delay constraint may not always be satisfiable**

Modified Relative Schedule



Vertex	Relative Offset v_1	v_3	v_8	v_9
v_1	--	--	--	
v_2	0	--	--	
v_3	0	--	--	
v_4	1	0	--	
v_5	1	0	--	
v_6	1	0	--	
v_7	4	3	--	
v_8	~~4~~ 5	3	--	
v_9	~~4~~ 5	3	--	
v_{10}	--	--	--	0
v_{11}	--	--	--	0
v_{12}	--	--	--	1
v_{13}	--	--	0	--

Satisfiability - Delay Constraints

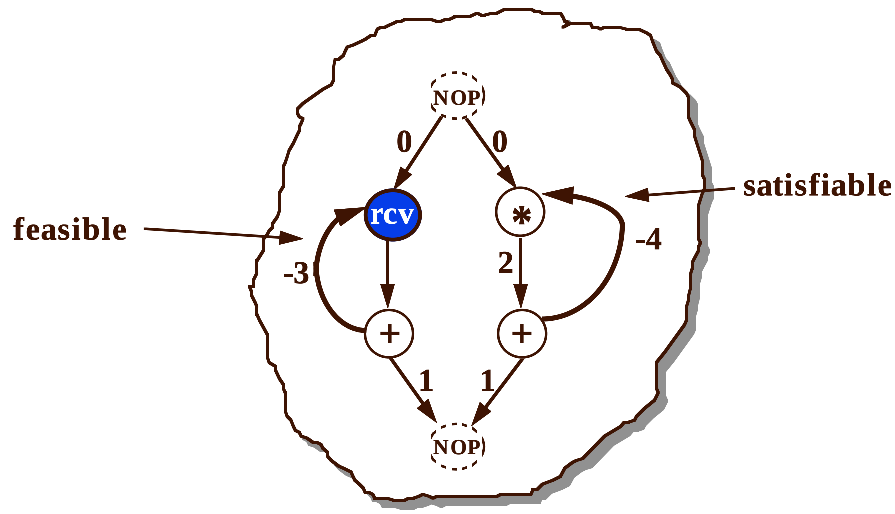
Feasibility:

- A constraint graph is considered ***feasible*** if it contains *no positive cycle* when the delay of ND operations is assigned to zero.

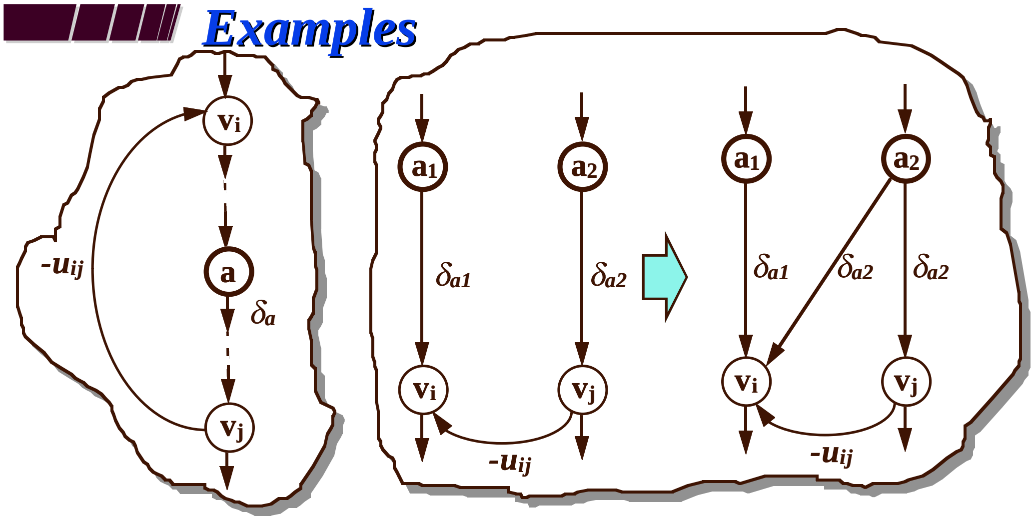
Condition <u>necessary and sufficient</u> to determine the *satisfiability* of constraints in the presence of *ND* operations:

- Operation delay constraints are *satisfiable* if and only if
  - the constraint graph is <u>feasible</u>
  - there exists <u>no cycles with ND operations</u>

Example



Examples

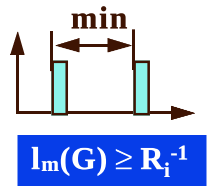


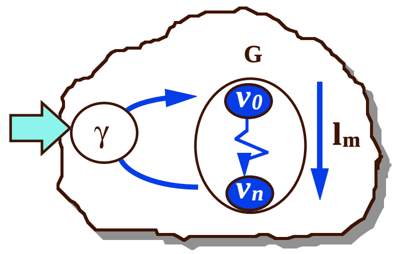
Constraints are not satisfiable (maybe feasible)

can be modified such that... ⇨ Constraints are satisfiable

Satisfiability - Rate Constraints

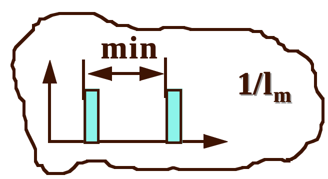
➡ **Maximum rate constraints are always satisfiable**

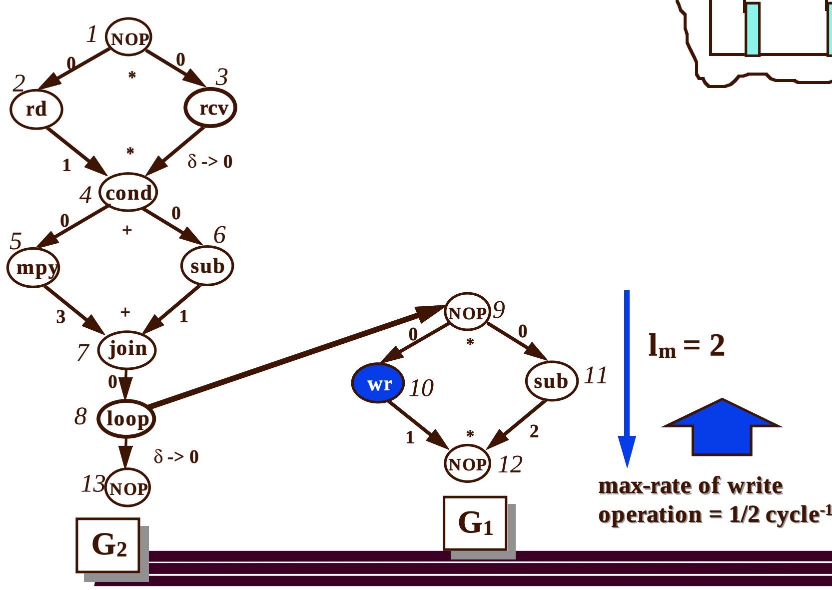




- **appropriate choice of *overhead delay* (γ) applicable to every execution of G**

Example - max rate





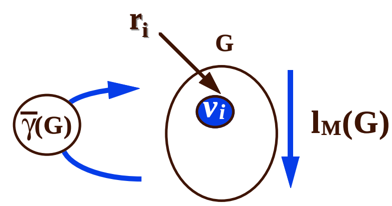
Min Rate Constraint

➩ A minimum rate constraint r_i on an operation $v_i \in V(G)$, where G contains no ND operations is satisfiable if

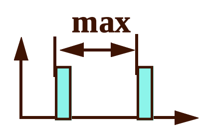
$$\bar{\gamma}(G) + l_M(G) \leq (\tau / r_i)$$

bound on latency

overhead delay



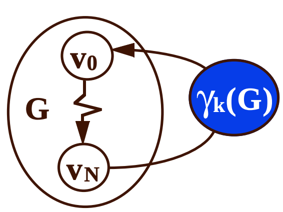
- A minimum rate constraint places an upper bound on the interval of successive executions of an operation



Overhead Delay

- $\gamma_k(G)$: reinvocation delay for G
  - may be a fixed quantity: overhead due to a run time scheduler
  - may be variable: in case of conditional invocation of G

Overhead Delay ➩ $\gamma_k(G) = t_{k+1}(v_0(G)) - t_k(v_N(G))$

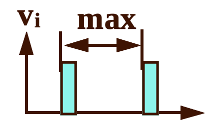


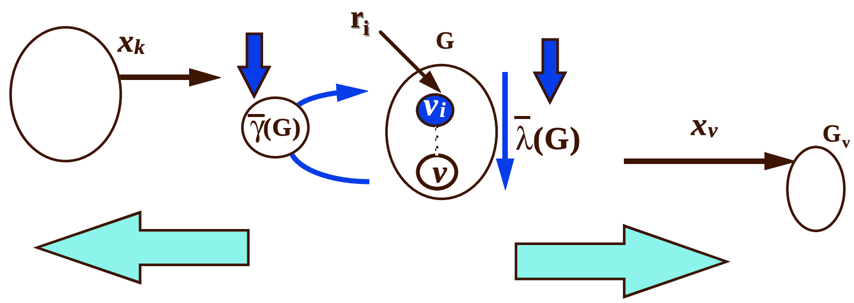
additional delay operation in series with the sink operation $v_N(G)$

Min Rate Constraints

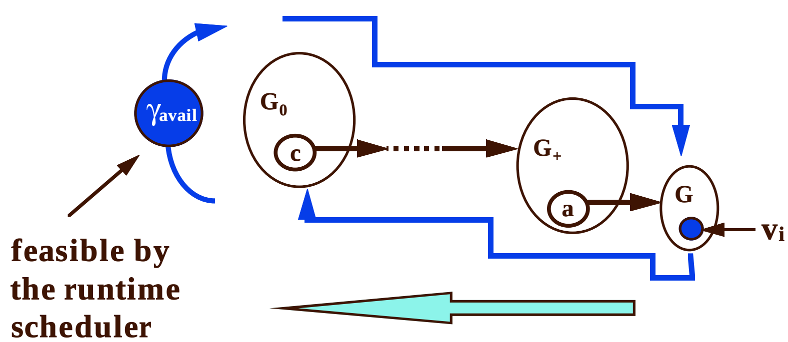
- General case: involves two bounds

$\overline{\gamma}(G) + \overline{\lambda}(G) \leq (\tau / r_i)$





Min Rate: satisfiability



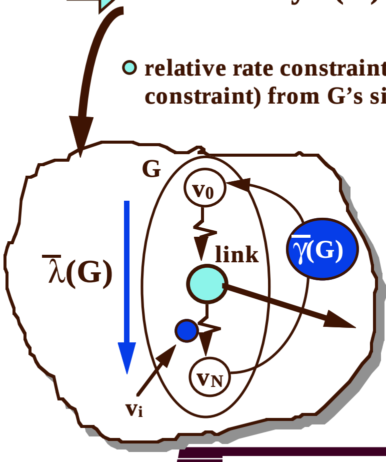
Min Rate : satisfiability

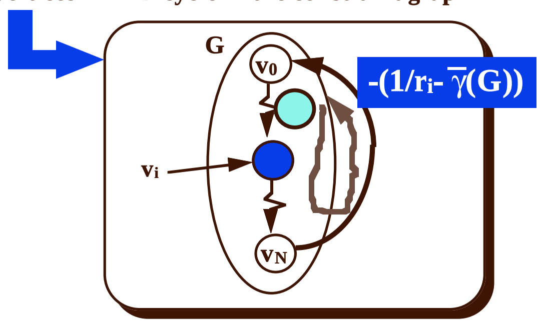
In the presence of *ND* operations in G:

$\overline{\gamma}(G) + \overline{\lambda}(G) \leq (1/r_i)$

The latency $\lambda(G)$ is not bounded

- **relative rate constraints -- represented as a backward edge (i.e., max delay constraint) from G's sink to source vertices => ND cycle in the constraint graph**



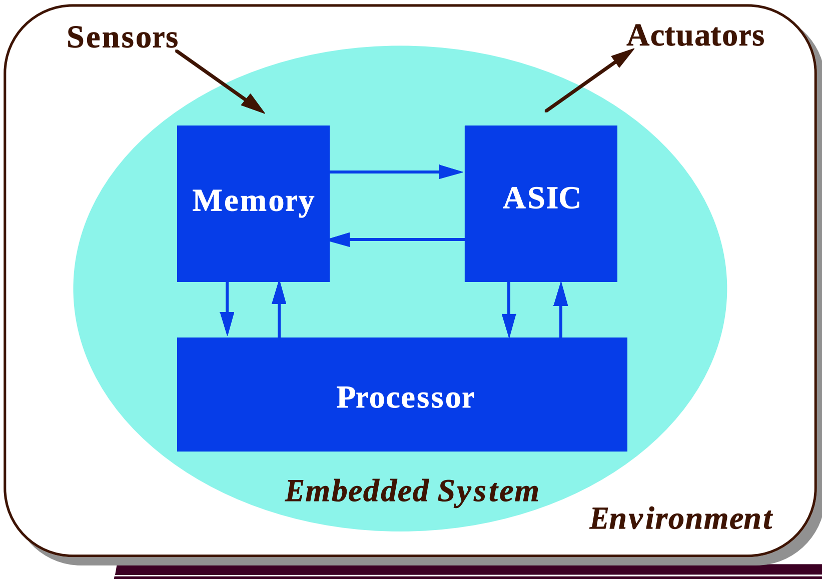


min rate constraint ⇒ bound on loop index, $\overline{x}$

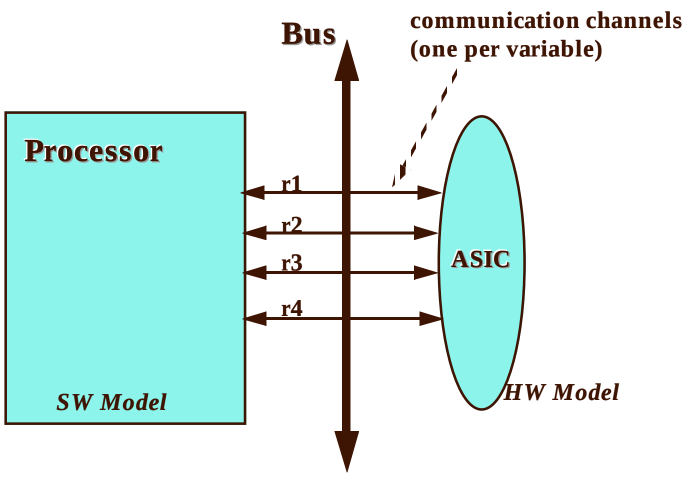
Vulcan

- Specification
- Modeling
- Constraint Analysis
- Software and Runtime Environment
- Target Architecture - H/S Interface
- Partitioning

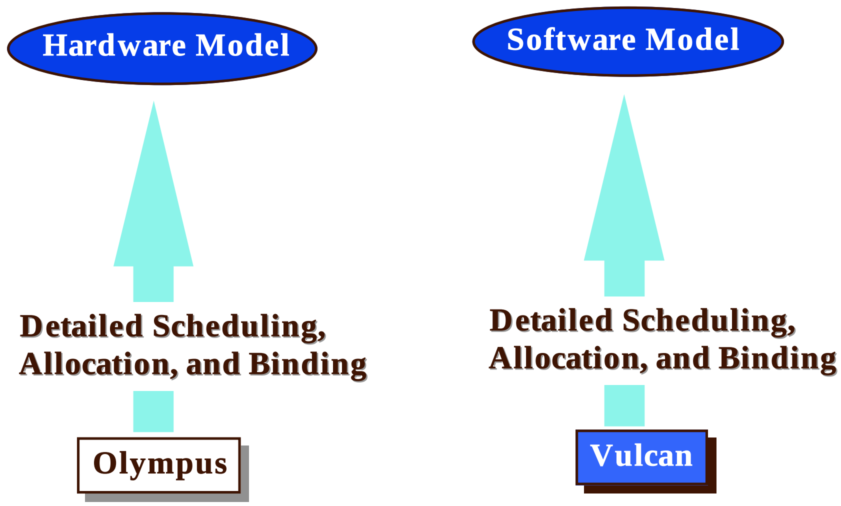
Micro-controller Architecture



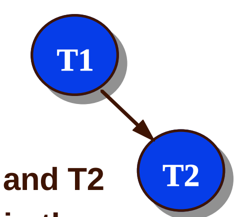
Target Architecture



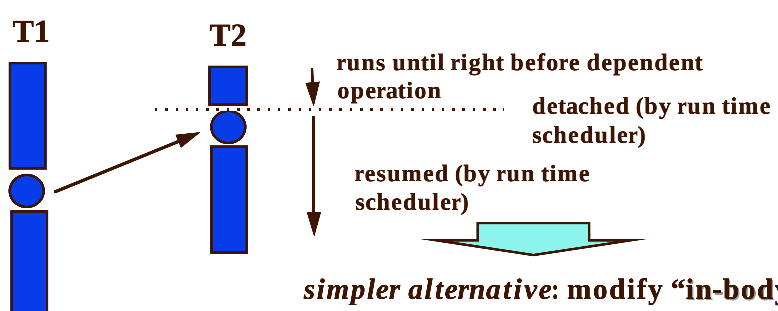
Runtime System



Software Model



- Dependence between two threads T1 and T2
  - dependencies between operations in the bodies of T1 and T2



simpler alternative: **modify "in-body" dependencies ⇒ make graphs *<u>convex</u>***

Convex Graph

- A (sub)graph is defined to be convex if it has only one single entry and exit operations.

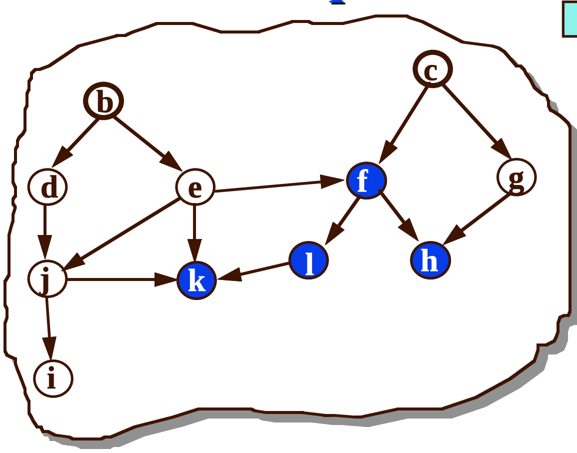
The corresponding program thread, once invoked, can run to completion without need to detach in order to observe dependencies

"cost"

- Potential loss of concurrency
- Constraint analysis must be performed on the modified graphs

"benefit" All routines can be implemented as independent programs with statically embedded control dependencies

Example



Flow graph to be implemented in software

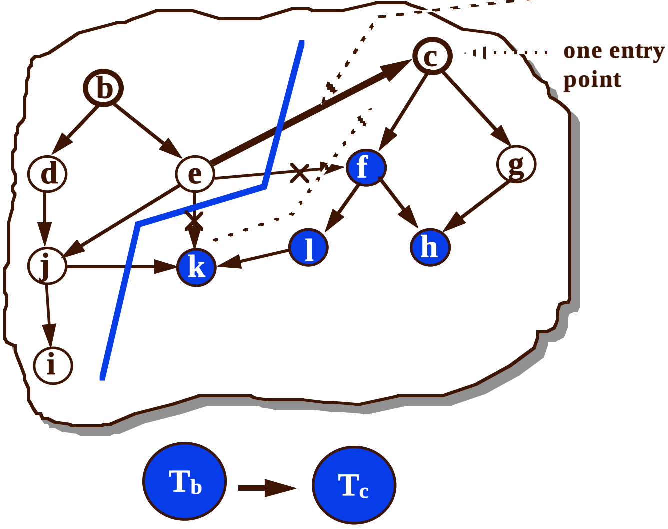
2 ND operations (b and c)
⇒ 2 threads

Operations with the same anchor set belong to the same thread

operations k, l, f, h: anchor set
⇒ b and c...

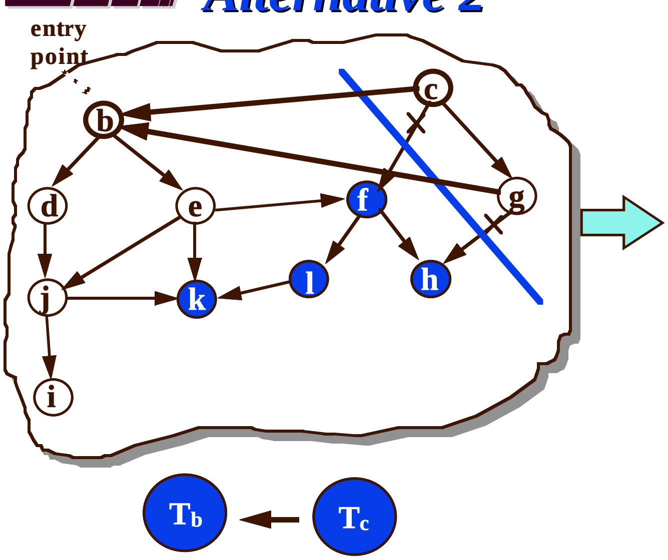
according with the choice of thread for these operations ≠ control dependencies will exist between the two resulting threads

Alternative 1



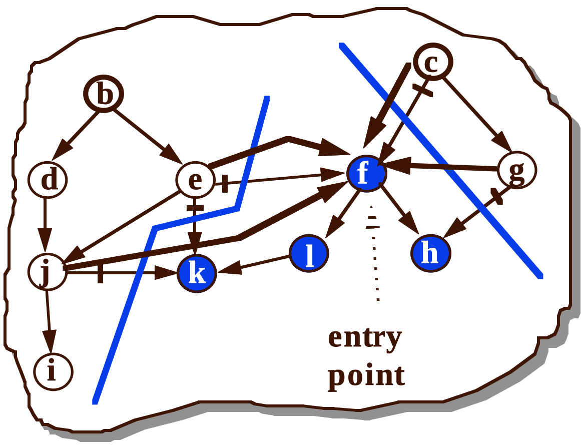
dependency
created due to
convexity
serialization

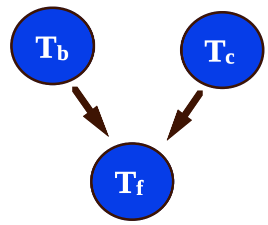
Alternative 2



(c, f) substituted by (c,b)
(g, h) substituted by (g,c)

Alternative 3





A Model for Software

- Software is constructed as a set of *concurrent program threads*

A tread is defined as a linearization of operations that may or may not begin by an *ND operation*

the latency of a thread (λ) is defined as the sum of the delay of its operations without including the ND operation

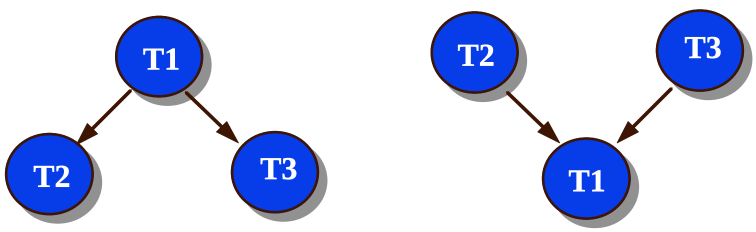
merged into the delay of the runtime scheduler

Margarida Jacome - UT Austin - Spring 97

Control Flow in the Software

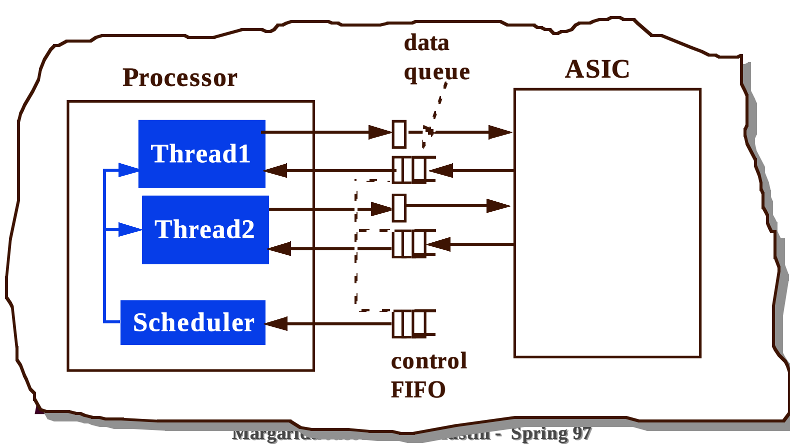
- There may be dependencies between operations that belong to separate threads

known statically ⇒ programs threads are constructed to observe these dependencies

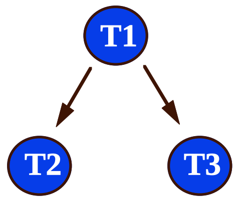


Non-prioritized FIFO Scheduler

- A thread is enabled when its “id” is in the control FIFO
- Before detaching, a thread performs one or more enqueue operations to the FIFO, for its dependent threads



Inter-thread Control Dependencies

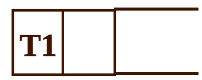



```
Thread T1
<body>
enqueue(T2)on cFIFO
enqueue(T3)on cFIFO
detach
```

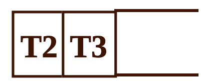
linearized set of operations

Control FIFO

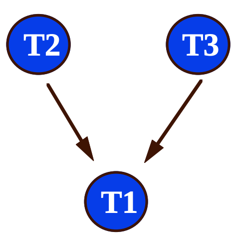
Before T1:



After T1:



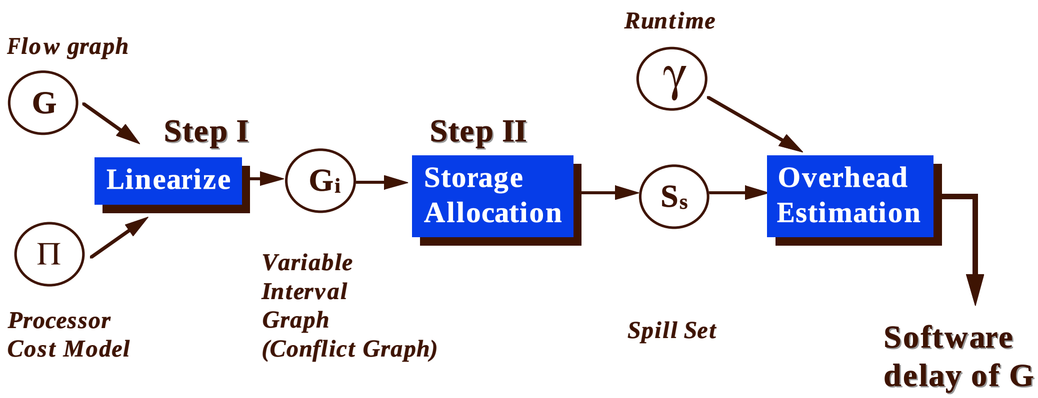
Thread with Multiple Control Dependencies




```
Thread T1
while (count != 1)
{
  count = count + 1;
  detach
}
<body>
count = 0;
enqueue(successor threads)on cFIFO
detach
```

synchronization preamble code

Software Size/Delay Estimation



Vulcan

- Specification
- Modeling
- Constraint Analysis
- Software and Runtime Environment
- Target Architecture - H/S Interface
- ⇨ Partitioning
- Co-simulation

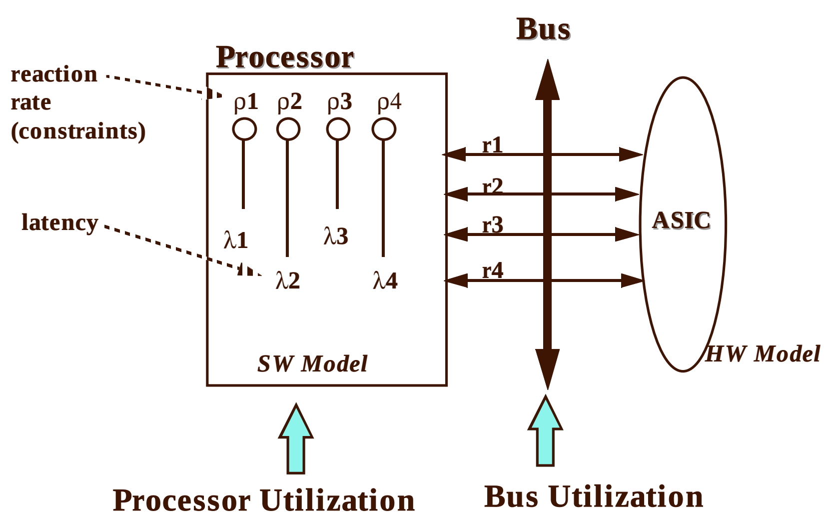
Problem Formulation

For a given set of flow graph models and timing constraints, create two sets of flow graph models such that one can be implemented in hardware and the other in software and the following is true:

- **Timing constraints are satisfied**
- **Processor utilization, $P \leq 1$**
- **Bus utilization, $B \leq \overline{B}$** (cumulative size of variables transferred across the partition)
- **A cost function $f(S_H, S_{S\Pi}, B, P^{-1}, m)$ is minimized...**

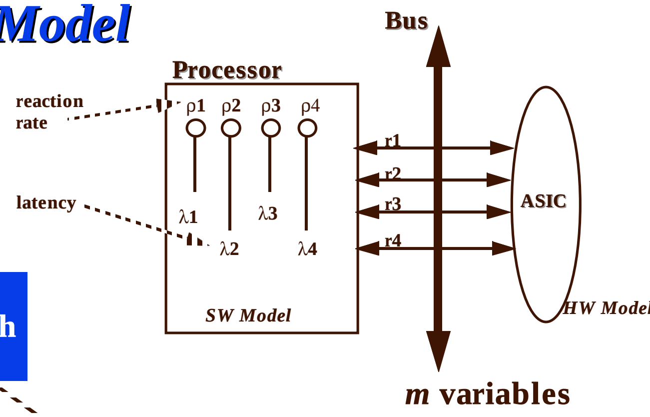
weights: represent a desired tradeoffs between size of the hardware, processor and bus utilization, and communication overhead

Software Model



Margarida Jacome - UT Austin - Spring 97

Software Model



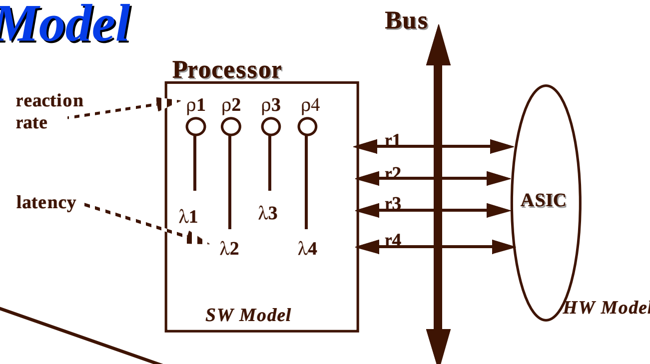
Bus Utilization

$$B := \sum_{j=1}^{m} r_j \leq \text{bus bandwidth}$$

bounded by bus bandwidth: function of the bus cycle time and memory access time

inverse of the time interval between two consecutive samples for a variable "j"

Software Model



Processor Utilization

$$P := \sum_{i=1}^{n} \rho_i \bullet \lambda_i$$

***n* threads**

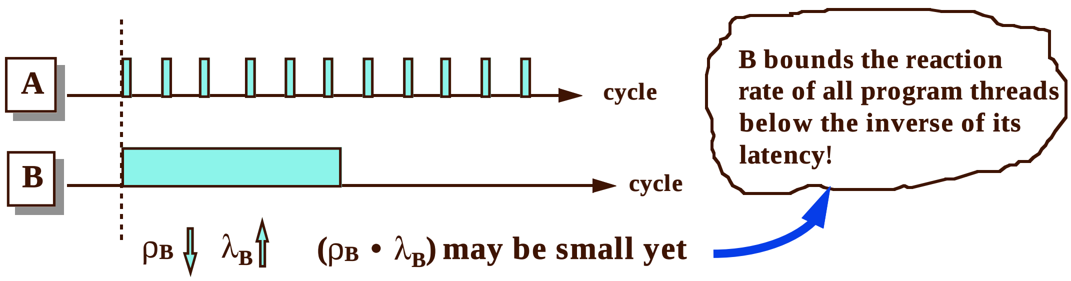
Satisfiability to reaction rates of program threads

Bound on Processor Utilization

considering case where all threads are enabled simultaneously

$$P := \sum_{i=1}^{n} \rho_i \bullet \lambda_i \leq 1$$

necessary but not sufficient!



Satisfiability to reaction rates of program threads

Sufficient condition for a program thread (for non-preemptive non-prioritized runtime scheduler)

$$(1/\rho_{max}) \geq \sum_{\text{all threads } k} \lambda_k$$

necessary and sufficient for independent threads (can be weakened for dependent threads)

maximum reaction rate over all program threads

Margarida Jacome - UT Austin - Spring 97

Partitioning Feasibility

- Determined based on a worst case scenario
  - ensure that worst case scenario is handled

Timing constraints: min/max delay and execution rate

Performance constraints: processor and bus utilization, run-time scheduler (software)

"Greedy" Partition Algorithm

```
graph_partition(G) {
  VH = V(G);
  VS = {};
  for v ∈ V(G) {                                  /* initialization */
     if v is a ND link operation                  /* All ND Loops go to Software */
        VS = VS + {v};
  }
  create software threads (VS);                   /* serialization, etc. */
  compute reaction rates for each thread;         /* based on rate constraints */
  if not check_feasibility (VH,VS)                /* timing cnstr, processor and bus utilization */
    exit;
  fmin = f (VH,VS) ;                              /* initialize cost function */
  repeat {
    for v ∈ VH and v is not ND                    /* pick HW operation --> SW */
    fmin = move(v);                               /* move(v) calls check_feasibility */
    } until no further reduction in fmin
  return (VH,VS) ;
```