Modulo Scheduling

- Modulo Scheduling

Modulo Scheduling

- Most popular software pipelining technique
  - Originally developed by B. Rau, later simplified by M. Lam
  - All commercial compilers include this technique
    - Another commercial technique is EPS

- Trial-and-error method to get a pipelined schedule
  - Compute the minimum initiation interval (*MII*) based on both precedence constraints (precedence MII) and resource constraints (resource MII)
  - Try to obtain a schedule with such an *MII*
    - Based on instruction placement, not code motion
  - If cannot find a schedule, try with *MII* + 1, and continue

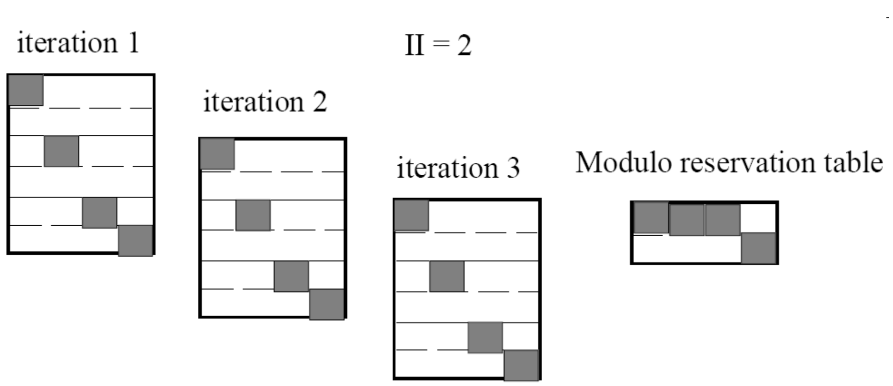
Determining *MII* : Resource Constraints

- Representation of resource constraints: reservation table

A reservation table for "MUL r1 r2 r3"

Resource Constraints and the *II*

- If an instruction is scheduled at cycle x, it will also execute at cycle *x* + *II*, *x* + *2* x *II*, and so on
- The resource requirements of a single iteration should not exceed the available resources
- The available resources of the kernel increase as the II increases



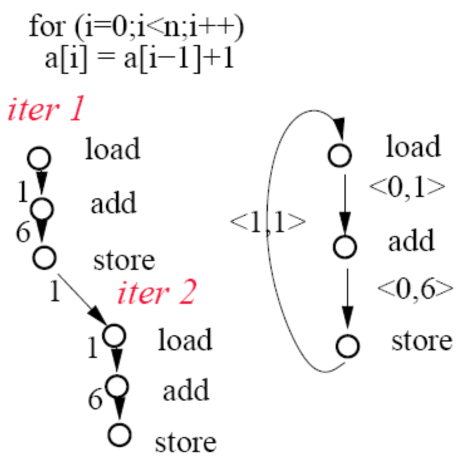
Resource MII *(RMII)*

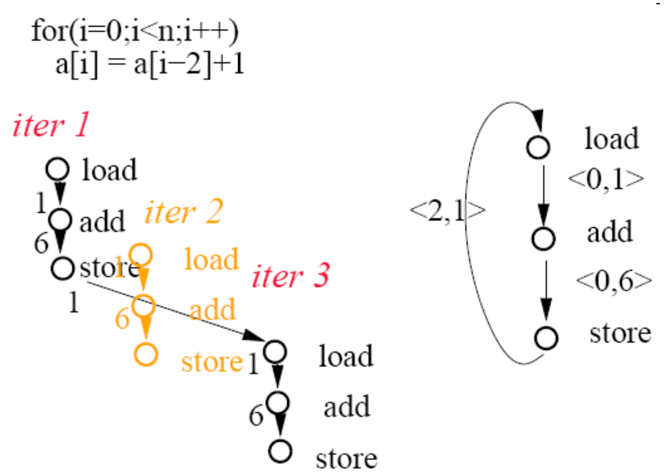
- For all resources *i*,
  - Number of units required by one iteration: r_i
  - Number of units in system: R_i
  - *RMII* = $\max_i \left\lceil \frac{r_i}{R_i} \right\rceil$

  - If the ratio is not integral, unrolling can improve the lower bound (e.g., 3 mem refs / 2mem ports = 1.5)

Determining *MII* : Precedence Constraints

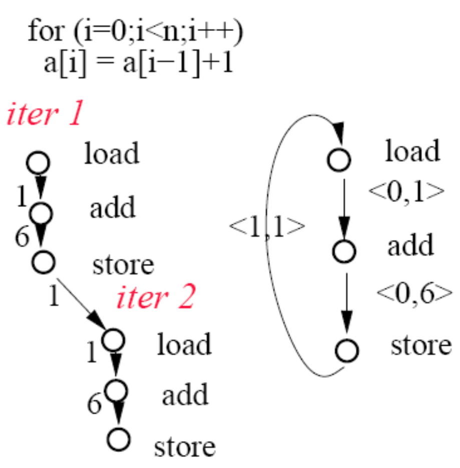
- Representation of precedence constraints: data dependence graph
  - Node: instruction, Edge: dependence relationship
- Observation of dependence relationship
  - Must show iteration difference as well as delay

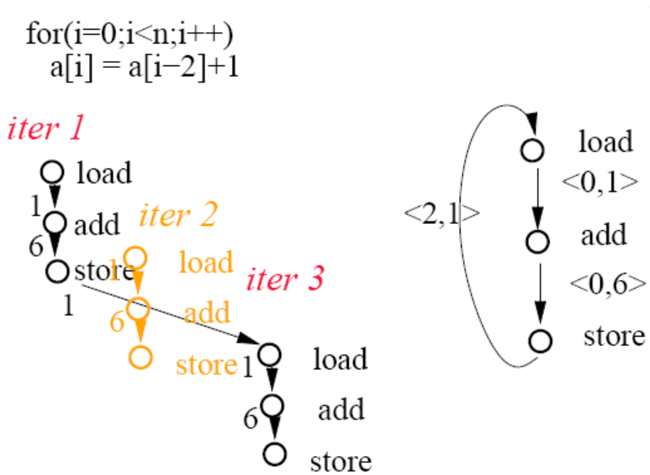




Determining *MII* : Precedence Constraints

- An edge $\overrightarrow{u,v}$ in data dependence graph has <*p*,*d*>
  - *d* : a delay value
    - *V* can start no earlier than *d* cycles after node *u* starts
  - *p* : a value representing minimum iteration distances
    - *p* = 0: intra-iteration dependence, *p* > 0: loop-carried dependence
- Some observation of precedence MII
  - 8 cycles in the left example but 4 cycles in the right example

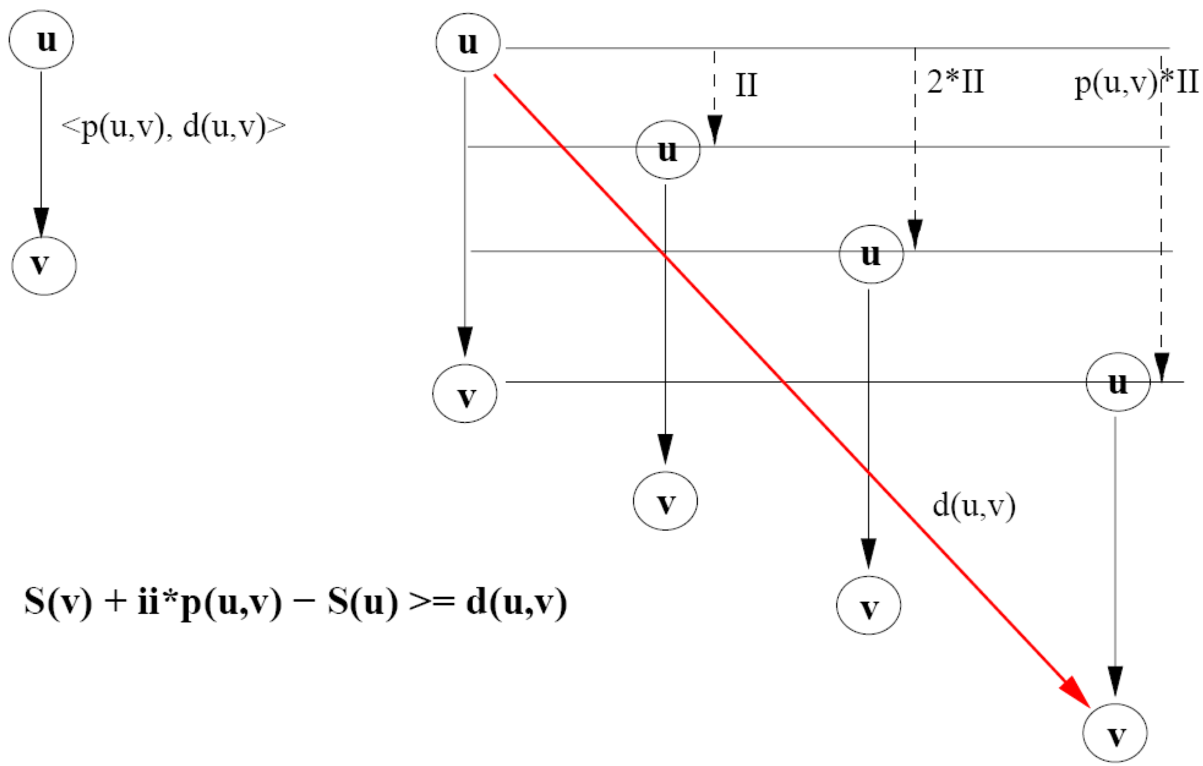




II and the Schedule

- Given an initiation interval *II* and S(x) is the cycle where x is scheduled,

$$S(v) - S(u) \geq d(u,v) - II \times p(u,v)$$



S(v) + ii*p(u,v) − S(u) >= d(u,v)

Precedence MII (PMII)

For all cycles c in the data dependence graph

$$PMII = \max_c \frac{cycle\_length}{iteration\_differences}$$

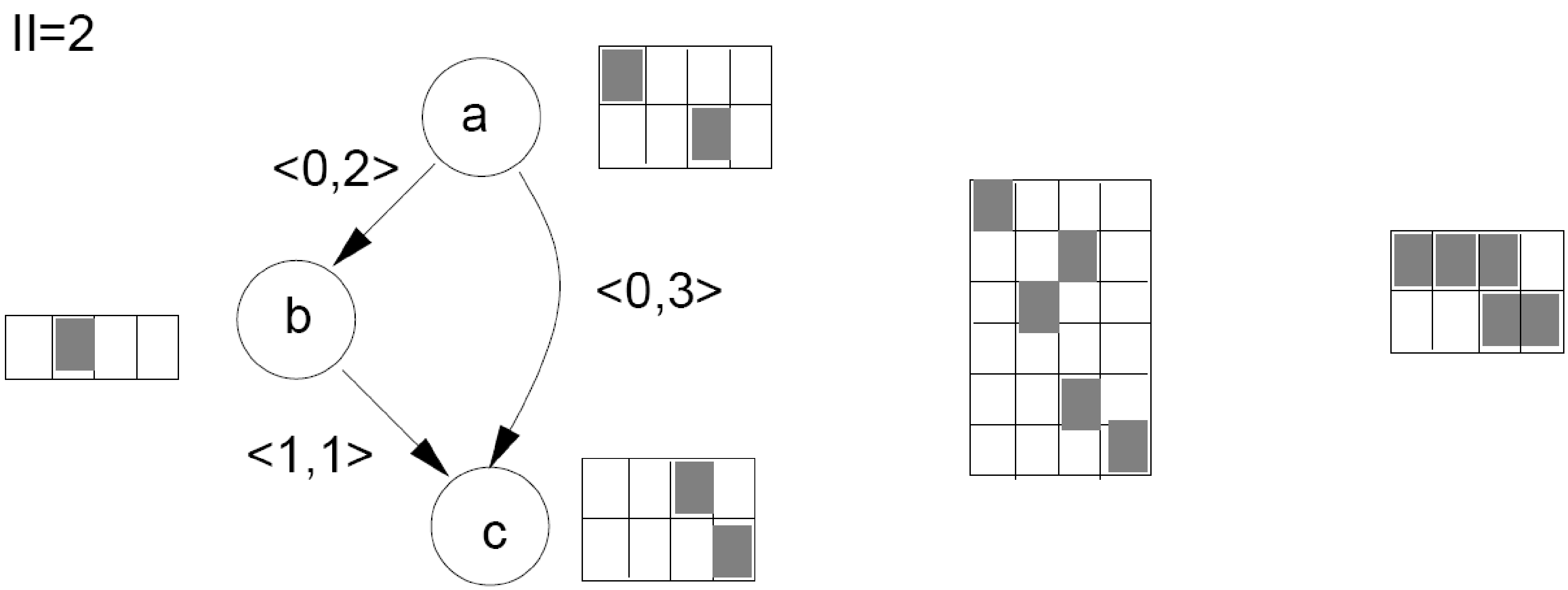
- Why? For each dependence edge in the cycle,
  - Represent S(v) – S(u) >= d(u, v) – p(u, v) * II
  - Then, sum them up, which will make
    - 0>= sum(d) – sum(p) * II, meaning II>=sum(d)/sum(p)
- If the PMII ratio is not integral, unrolling can improve the lower bound

Modulo Scheduling

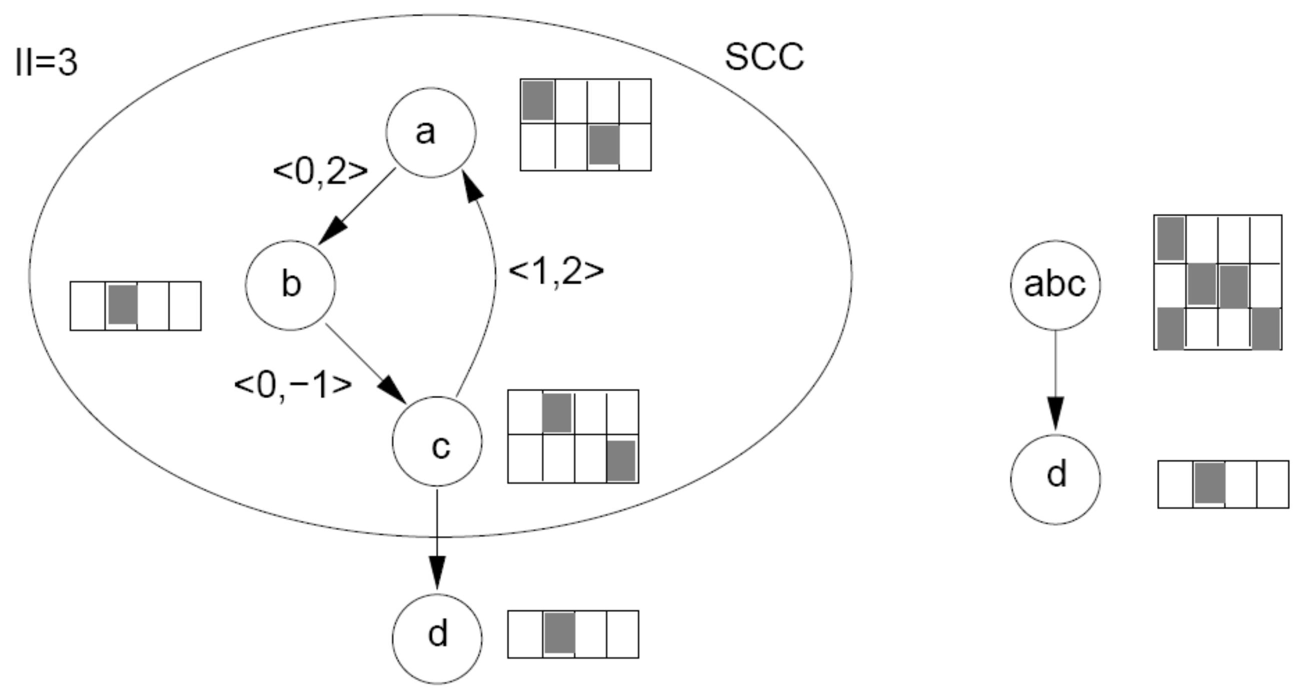
- Determining Minimum Initiation Interval (MII)
  - *MII = min(RMII, PMII)*
- Interdependence between *II* and constraints
  - Constraints determine the minimum *II*
  - *II* affects modulo reservation tables, scheduling functions, S(x), etc
  - Minimizing *II* is NP-complete
- Goal of scheduling
  - Determine *II*
  - Solve the scheduling function S(x) for each instruction in the loop

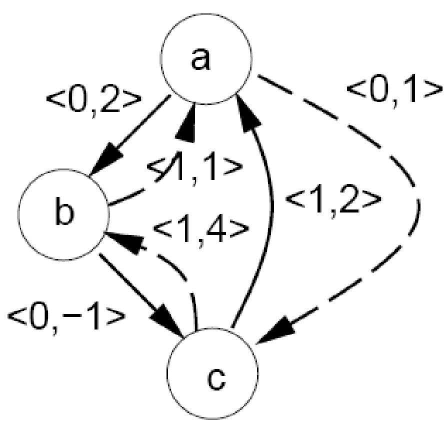
Generating Pipelined Schedules

- Scheduling Acyclic Data Dependence Graph: List Scheduling



Scheduling Cyclic Graph





$$S(b) >= S(a) + 2$$

$$S(a) + ii >= S(b) + 1$$

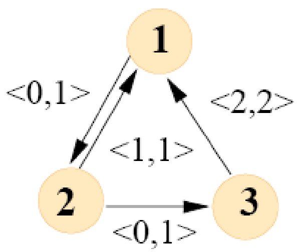
$$S(a) + ii - 1 >= S(b) >= S(a) + 2$$

Cyclic Precedence Constraints

- Observation: scheduling a node make the schedules of all other nodes from above and below
  - Depends on the II
- Implementation: pre-compute longest path lengths between all points based on II value
  - Once for all II by using a symbolic value for *II*
  - Why longest paths? Meeting worst-case constraints

Why Longest Path? An Example

(1) a[i] = c[i] + b[i]	T(2) − T(1) >= 1		
(2) b[i−1] = a[i]	T(1) + ii − T(2) >= 1	T(1) − T(2) >= 1 − ii	
(3) c[i−2] = b[i−1]	T(1) + 2*ii − T(2) >= 3	T(1) − T(2) >= 3 − 2*ii	



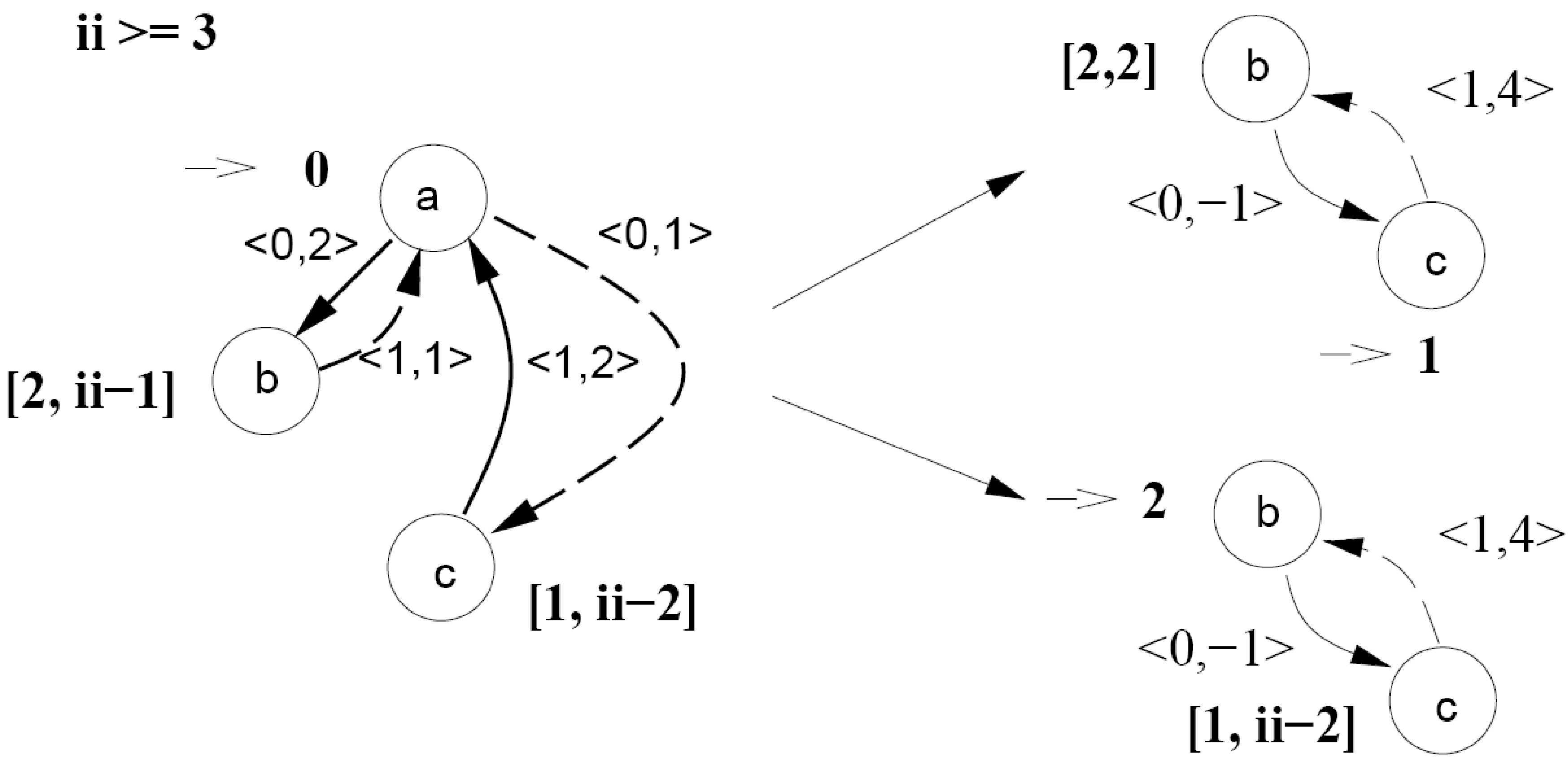
longest path from 2 to 1	when ii = 2	when ii = 3
bigger one between 1−ii and 3−2ii	T(1) = 0	T(1) = 0
	T(2) = 1	T(2) <= 2 or T(2) <= 3

	ii=3 T(2) <= 2		ii=3 T(2) <= 3	
0	a[i] = c[i] + b[i]		a[i] = c[i] + b[i]	
1				
2	b[i−1] = a[i]			
3	c[i−2] = b[i−1]	a[i] = c[i] + b[i]	b[i−1] = a[i]	a[i] = c[i] + b[i]
4			c[i−2] = b[i−1]	
5		b[i−1] = a[i]		
6		c[i−2] = b[i−1] a[i] = c[i] + b[i]		b[i−1] = a[i]
7				c[i−2] = b[i−1]

Longest Paths

- The closure of the cost of a path *e*
  - $d(e) - ii \times p(e)$
  - To capture all possible maximum costs between two nodes, the longest path is represented as a set

Scheduling Order



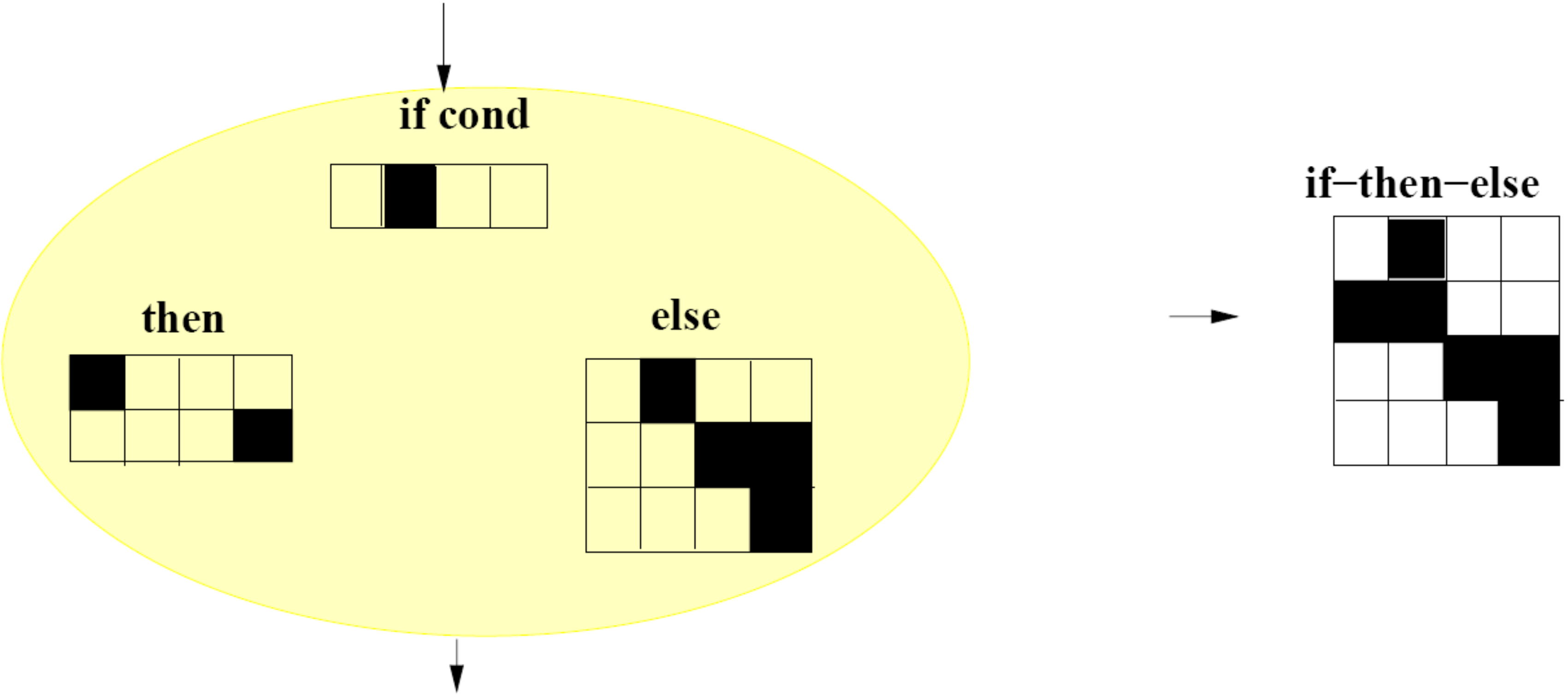
- Topological ordering on intra-iteration constraints
- Upper bounds increases with *ii*

Scheduling Algorithm Summary

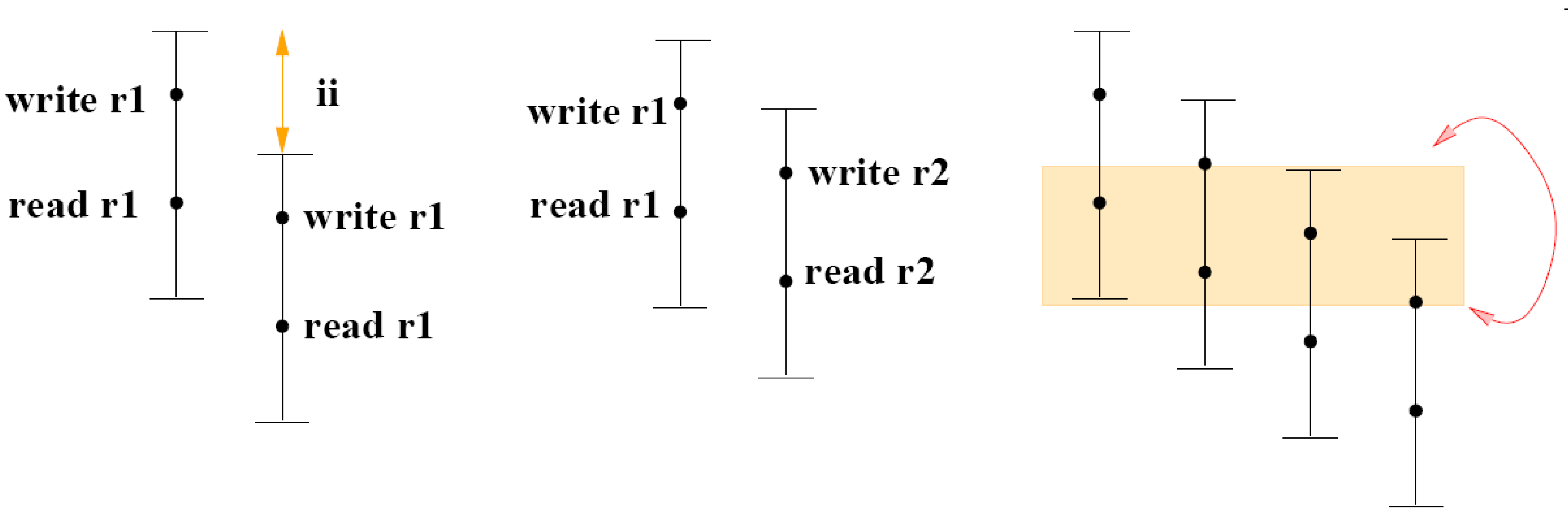
- Given *II*, Schedule SCCs first
  - Scheduling a node bounds rest nodes from below and above
  - Satisfy precedence constraints
    - Pre-compute all-points longest paths
    - Allow fast update on the range of each node
  - Satisfy resource constraints
    - Topological ordering on intra-iteration edges
    - Upper bounds increases with initiation interval
  - Schedule reduced acyclic graphs
    - Schedule node in topological ordering
    - Find conflict slots within initiation window

Hierachical Reduction

- For hammock-type conditional statements
  - Union both resource and precedence constraints



Modulo Variable Expansion



Modulo Variable Expansion Algorithm

- Schedule without considering cross-iteration anti-dependences
- lifetime > *II* : multiple registers are needed
- Assign registers
  - *L(v)* = lifetime of variable *v*
  - *ii* = initiation interval
  - Suppose *L(u)* = 3 x *ii* and *L(v)* = 2 x *ii*
  - Unroll three times, use three registers for each *u* and *v*
  - If only two registers for *v*, unroll 6 times

Rotating Register (RR)

- Can use rotating registers instead of unrolling
  - Architectural mechanism for renaming
  - Employed in Intel's P6 Itanium processors
  - Composed of n registers (RR(0) ~ RR(n))
  - Includes a brtop (or brexit) instruction
    - If a brtop is taken, RR(i) is accessible via RR((i+1)%n)
    - That is, a block shift of registers, RR(i+1) = RR(i), occur
  - We allocate RR to those whose lifetime > II, we can avoid cross-iteration register overwrites