Instruction Scheduling

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

Outline

- Instruction Scheduling
    - Simple Basic Block Scheduling
    - Automaton-based Scheduling
    - Integer programming based scheduling
    - Optimal Delayed-load Scheduling (DLS) for trees
    - Trace, Superblock and Hyperblock scheduling

Instruction Scheduling

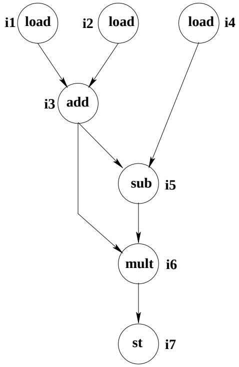
- Reordering of instructions so as to keep the pipelines of functional units full with no stalls
- NP-Complete and needs heuristcs
- Applied on basic blocks (local)
- Global scheduling requires elongation of basic blocks (super-blocks)

Instruction Scheduling - Motivating Example

- time: load - 2 cycles, op - 1 cycle
- This code has 2 stalls, at i3 and at i5, due to the loads

i1:	r1	←	load a
i2:	r2	←	load b
i3:	r3	←	r1 + r2
i4:	r4	←	load c
i5:	r5	←	r3 - r4
i6:	r6	←	r3 * r5
i7:	d	←	st r6

(a) Sample Code Sequence



(b) DAG

Scheduled Code - no stalls

- There are no stalls, but dependences are indeed satisfied

i1:	r1	←	load a
i2:	r2	←	load b
i4:	r4	←	load c
i3:	r3	←	r1 + r2
i5:	r5	←	r3 - r4
i6:	r6	←	r3 * r5
i7:	d	←	st r6

Definitions - Dependences

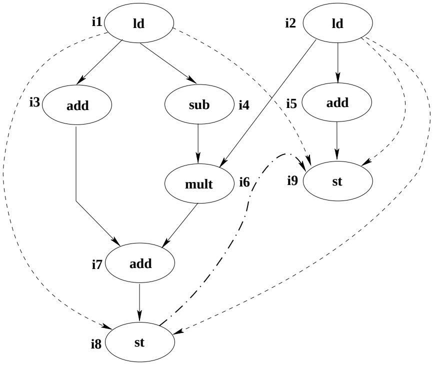
- Consider the following code:
  $i_1 : \; r1 \leftarrow load(r2)$
  $i_2 : \; r3 \leftarrow r1 + 4$
  $i_3 : \; r1 \leftarrow r4 + r5$
- The dependences are
  $i_1 \; \delta \; i_2$ (flow dependence) $i_2 \; \overline{\delta} \; i_3$ (anti-dependence)
  $i_1 \; \delta^o \; i_3$ (output dependence)
- anti- and ouput dependences can be eliminated by register renaming

Dependence DAG

- full line: *flow* dependence, dash line: *anti*-dependence
  dash-dot line: *output* dependence
- some anti- and output dependences are because memory disambiguation could not be done

i1:	t1 ← load a
i2:	t2 ← load b
i3:	t3 ← t1 + 4
i4:	t4 ← t1 - 2
i5:	t5 ← t2 + 3
i6:	t6 ← t4 ∗ t2
i7:	t7 ← t3 + t6
i8:	c ← st t7
i9:	b ← st t5

(a) Instruction Sequence



(b) DAG

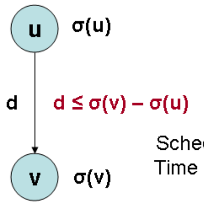
Basic Block Scheduling

- Basic block consists of micro-operation sequences (MOS), which are indivisible
- Each MOS has several steps, each requiring resources
- Each step of an MOS requires one cycle for execution
- Precedence constraints and resource constraints must be satisfied by the scheduled program
    - PC's relate to data dependences and execution delays
    - RC's relate to limited availability of shared resources

The Basic Block Scheduling Problem

- Basic block is modelled as a digraph, $G = (V, E)$
  - R: number of resources
  - Nodes (V): MOS; Edges (E): Precedence
  - Label on node v
    - resource usage functions, $\rho_v(i)$ for each step of the MOS associated with v
    - length $l(v)$ of node v
  - Label on edge e: Execution delay of the MOS, $d(e)$
- Problem: Find the shortest schedule $\sigma : V \rightarrow N$ such that $\forall e = (u, v) \in E,\ \sigma(v) - \sigma(u) \geq d(e)$ and $\forall i, \sum^{v \in V} \rho_v(i - \sigma(v)) \leq R$, where length of the schedule is $\max_{v \in V}\{\sigma(v) + l(v)\}$

Instruction Scheduling - Precedence and Resource Constraints



Precedence constraint

MOS substeps (time)

Schedule Time $\sigma(u)$ ↓

Schedule Time $\sigma(u)$	0	1	2	3	4
$\sigma(v_1)=0$	1	1	🔴 2	2	
$\sigma(v_2)=1$	2	🔴 3	1	🔵 1	2
$\sigma(v_3)=2$	🔴 3	1	🔵 2		
$\sigma(v_4)=3$	1	🔵 2	3	2	

Resource constraint

Consider R = 5. Each MOS substep takes 1 time unit.

🔵 At i=4, $\varsigma_{v4}(1)+\varsigma_{v3}(2)+\varsigma_{v2}(3)+\varsigma_{v1}(4) = 2+2+1+0 = 5 \le R$, satisfied

🔴 At i=2, $\varsigma_{v3}(0)+\varsigma_{v2}(1)+\varsigma_{v1}(2) = 3+3+2 = 8 > R$, NOT satisfied

A Simple List Scheduling Algorithm

Find the shortest schedule $\sigma : V \rightarrow N$, such that precedence and resource constraints are satisfied. Holes are filled with NOPs.

```
FUNCTION ListSchedule (V,E)
BEGIN
  $Ready$ = root nodes of V; $Schedule = \phi$;
  WHILE $Ready \neq \phi$ DO
  BEGIN
    v = highest priority node in $Ready$;
    $Lb$ = SatisfyPrecedenceConstraints ($v, Schedule, \sigma$);
    $\sigma$(v) = SatisfyResourceConstraints ($v, Schedule, \sigma, Lb$);
    $Schedule = Schedule + \{v\}$;
    $Ready = Ready - \{v\} + \{u \mid NOT\ (u \in Schedule)$
                 $AND\ \forall\ (w,u) \in E,\ w \in Schedule\}$;
  END
  RETURN $\sigma$;
END
```

List Scheduling - Ready Queue Update

$$\begin{tikzcd}
w_1 \arrow[d] & v \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & w_2 \arrow[d] & x_1 \arrow[d] \\
u_1 & u_2 & u_3 & x_2
\end{tikzcd}$$

Already scheduled nodes w

Unscheduled nodes which will get into the Ready queue now u

Currently scheduled node v

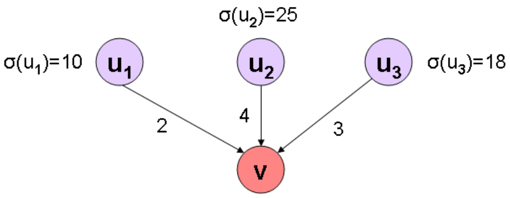
Unscheduled nodes x

Constraint Satisfaction Functions

```
FUNCTION SatisfyPrecedenceConstraint(v, Sched, $\sigma$)
BEGIN
   RETURN ($\max_{u \in Sched} \sigma(u) + d(u, v)$)
END

FUNCTION SatisfyResourceConstraint(v, Sched, $\sigma$, Lb)
BEGIN
   FOR i := Lb TO $\infty$ DO
      IF $\forall 0 \leq j < l(v),\ \rho_v(j) + \sum^{u \in Sched} \rho_u(i + j - \sigma(u)) \leq R$ THEN
          RETURN (i);
END
```

Precedence Constraint Satisfaction



Lower bound for σ(v) = 29

Already scheduled nodes

Node to be scheduled

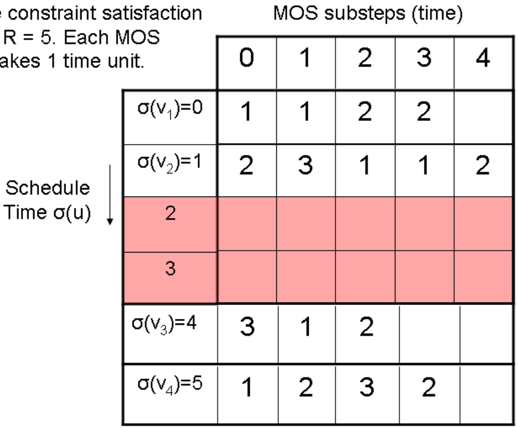
Precedence constraint satisfaction:

v can be scheduled only after all of u_1, u_2, and, u_3, finish

Lower bound for σ(v)
= max(10+2, 25+4, 18+3)
= max(12, 29, 21) = 29

Resource Constraint Satisfaction

Resource constraint satisfaction
Consider R = 5. Each MOS substep takes 1 time unit.



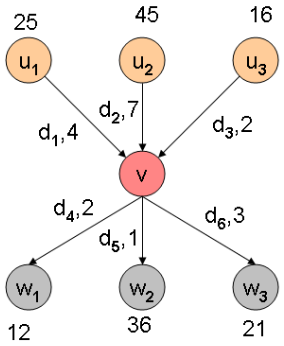
Schedule Time $\sigma(u)$ ↓	MOS substeps (time) 0	1	2	3	4
$\sigma(v_1)=0$	1	1	2	2	
$\sigma(v_2)=1$	2	3	1	1	2
2	🟥	🟥	🟥	🟥	🟥
3	🟥	🟥	🟥	🟥	🟥
$\sigma(v_3)=4$	3	1	2		
$\sigma(v_4)=5$	1	2	3	2	

Time slots 2 and 3 are vacant because scheduling node v_3 in either of them violates resource constraints

List Scheduling - Priority Ordering for Nodes

1. Height of the node in the DAG (*i.e.,* longest path from the node to a terminal node
2. *Estart*, and *Lstart*, the earliest and latest start times
   - Violating *Estart* and *Lstart* may result in pipeline stalls
   - $Estart(v) = \max_{i=1,\cdots,k}(Estart(u_i) + d(u_i, v))$ where $u_1, u_2, \cdots, u_k$ are predecessors of v. *Estart* value of the source node is 0.
   - $Lstart(u) = \min_{i=1,\cdots,k}(Lstart(v_i) - d(u, v_i))$ where $v_1, v_2, \cdots, v_k$ are successors of u. *Lstart* value of the sink node is set as its *Estart* value.
   - *Estart* and *Lstart* values can be computed using a top-down and a bottom-up pass, respectively, either statically (before scheduling begins), or dynamically during scheduling

Estart and Lstart Computation



$$Estart(v) = \max_{i=1,\ldots,3} (Esart(u_i) + d_i)$$
$$= \max(25+4, 45+7, 16+2)$$
$$= \max(29, 52, 18) = 52$$

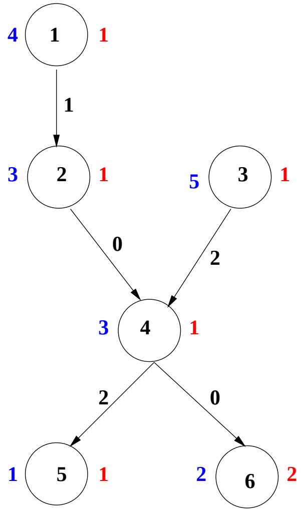
$$Lstart(v) = \min_{i=4,\ldots,6} (Lsart(w_i) - d_i)$$
$$= \min(12-2, 36-1, 21-3)$$
$$= \min(10, 35, 18) = 10$$

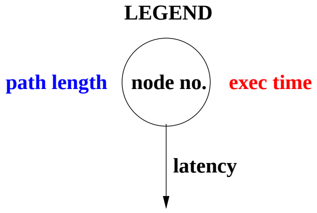
List Scheduling - Slack

1. A node with a lower *Estart* (or *Lstart*) value has a higher priority
2. $Slack = Lstart - Estart$
    - Nodes with lower slack are given higher priority
    - Instructions on the critical path may have a slack value of zero and hence get priority

Simple List Scheduling - Example - 1

INSTRUCTION SCHEDULING - EXAMPLE





path length (n) = exec time (n) , if n is a leaf

$$= \max_{m \in succ(n)} \{ \text{latency}(n,m) + \text{path length}(m) \}$$

Schedule = {3, 1, 2, 4, 6, 5}

Simple List Scheduling - Example - 2

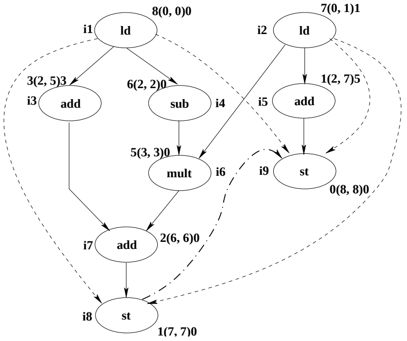
- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
- *path length* and *slack* are shown on the left side and right side of the pair of numbers in parentheses

```
c = (a+4)+(a-2)*b;
b = b+3;
```

(a) High-Level Code

i1:	t1	←	load a
i2:	t2	←	load b
i3:	t3	←	t1 + 4
i4:	t4	←	t1 - 2
i5:	t5	←	t2 + 3
i6:	t6	←	t4 * t2
i7:	t7	←	t3 + t6
i8:	c	←	st t7
i9:	b	←	st t5

(b) 3-Address Code



(c) DAG with *(Estart, Lstart)* Values

Simple List Scheduling - Example - 2 (contd.)

- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
  - 2 Integer units and 1 Multiplication unit, all capable of load and store as well
- Heuristic used: height of the node or slack

int1	int2	mult	Cycle #	Instr.No.	Instruction
1	1	0	0	i1, i2	$t_1 \leftarrow load\ a, t_2 \leftarrow load\ b$
1	1	0	1		
1	1	0	2	i4, i3	$t_4 \leftarrow t_1 - 2, t_3 \leftarrow t_1 + 4$
1	0	1	3	i6, i5	$t_5 \leftarrow t_2 + 3, t_6 \leftarrow t_4 * t_2$
0	0	1	4		i6/i5 not sched. in cycle 2
0	0	1	5		due to shortage of *int* units
1	0	0	6	i7	$t_7 \leftarrow t_3 + t_6$
1	0	0	7	i8	$c \leftarrow st\ t_7$
1	0	0	8	i9	$b \leftarrow st\ t_5$

Resource Usage Models - Reservation Table

Resources	Time Steps			
	0	1	2	3
r_0	1	0	0	0
r_1	0	1	1	0
r_2	0	0	0	1

(a) Reservation Table for I_1

Resources	Time Steps			
	0	1	2	3
r_0	1	0	0	0
r_3	0	1	0	0
r_4	0	0	1	1

(b) Reservation Table for I_2

Resource Usage Models - Global Reservation Table

	r_0	r_1	r_2	$\cdots$	r_M
t_0	1	0	1		0
t_1	1	1	0		1
t_2	0	0	0		1
t_T					

M: No. of resources in the machine
T: Length of the schedule

Resource Usage Models - Global Reservation Table

- GRT is constructed as the schedule is built (cycle by cycle)
- All entries of GRT are initialized to 0
- GRT maintains the state of all the resources in the machine
- GRTs can answer questions of the type:
  "can an instruction of class I be scheduled in the current cycle (say t_k)?"
- Answer is obtained by ANDing RT of I with the GRT starting from row t_k
  - If the resulting table contains only 0's, then YES, otherwise NO
- The GRT is updated after scheduling the instruction with a similar OR operation

Operation Scheduling

- List scheduling discussed so far schedules instructions on a cycle-by-cycle basis
- Operation scheduling attempts to schedule instructions one after another
- Tries to find the first cycle at which each instruction can be scheduled
- After choosing an operation i of highest priority, an attempt is made to schedule it at time t between $Estart(i)$ and $Lstart(i)$ that does not have any resource conflict
- This scheduling may affect the $Estart$ and $Lstart$ values of unscheduled instructions
- Priorities may have to be recomputed for these instructions

Operation Scheduling

- If no time slot as above can be found for instruction *i*, an already scheduled instruction *j*, which has resource conflicts with instruction *i* is *de-scheduled*
- Instruction *i* is placed in this slot and instruction *j* is placed in the ready list once again
- In order to ensure that the algorithm does no get into an infinite loop (a group of instructions mutually de-schedule each other), a threshold on the number of de-scheduled instructions is kept
- Once the threshold is crossed, the partial schedule is abandoned, the *Lstart* value of the sink node is increased, new value of *Lstart* is computed, and the whole process is restarted

Simple List Scheduling - Operation Scheduling

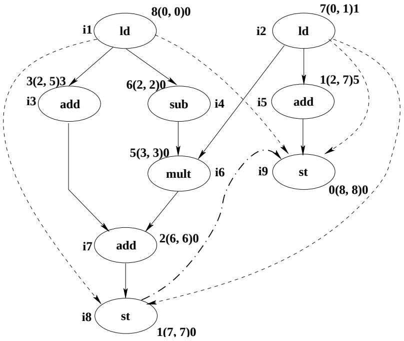
- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
  - 2 Integer units and 1 Multiplication unit, all capable of load and store as well

```
c = (a+4)+(a-2)*b;
b = b+3;
```

(a) High-Level Code

i1:	t1	←	load a
i2:	t2	←	load b
i3:	t3	←	t1 + 4
i4:	t4	←	t1 - 2
i5:	t5	←	t2 + 3
i6:	t6	←	t4 * t2
i7:	t7	←	t3 + t6
i8:	c	←	st t7
i9:	b	←	st t5

(b) 3-Address Code



(c) DAG with *(Estart, Lstart)* Values

Simple List Scheduling - Operation Scheduling (contd.)

- Instructions sorted on slack, with $(Estart, Lstart)$ values
  slack 0: $i_1(0,0), i_4(2,2), i_6(3,3), i_7(6,6), i_8(7,7), i_9(8,8)$
  slack 1: $i_2(0,1)$, slack 3: $i_3(2,5)$, slack 5: $i_5(2,7)$

Cycle #	Instr.No.	Instruction
0	i1, i2	$t_1 \leftarrow load\ a, t_2 \leftarrow load\ b$
1		
2	i4, i3	$t_4 \leftarrow t_1 - 2, t_3 \leftarrow t_1 + 4$
3	i6, i5	$t_5 \leftarrow t_2 + 3, t_6 \leftarrow t_4 * t_2$
4		
5		
6	i7	$t_7 \leftarrow t_3 + t_6$
7	i8	$c \leftarrow st\ t_7$
8	i9	$b \leftarrow st\ t_5$

Simple List Scheduling - Disadvantages

- Checking resource constraints is inefficient here because it involves repeated ANDing and ORing of bit matrices for many instructions in each scheduling step
- Space overhead may become considerable, but still manageable

Automaton Based Scheduling

- Constructs a collision automaton which indicates whether it is legal to issue an instruction in a given cycle (*i.e.,* no resource contentions)
- Collision automaton recognises legal instruction sequences
- Avoids extensive searching that is needed in list scheduling
- Uses the same topological ordering and ready queue as in list scheduling, to handle precedence constraints
- Automaton can be constructed offline using resource reservation tables

Collision Automaton

- Uses a collision matrix *for each state*
  - Size: #instruction classes $\times$ length of the longest pipeline
  - $S[i,j] = 1$, iff i^{th} instruction class creates a conflict with the current pipeline state S, if issued j cycles after the machine enters the current state S
- Each instruction class I also has a similar collision matrix
  - $I[i,j] = 1$, iff instruction of class i would create a conflict with instruction class I in cycle j, if launched in the current cycle
  - These collision matrices are created using resource vectors
- For the example, consider a *dual issue* machine

Collision Automaton - Example

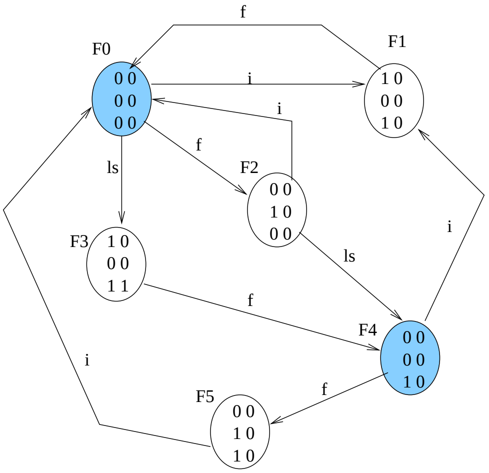
Resource Usage Vectors

instr class	pipeline cycle 0	1
i	id	
f	fd	
ls	id+mem	mem

Collision Matrices

$$\begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 0 & 0 \\ f & 1 & 0 \\ ls & 0 & 0 \end{array} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline i & 1 & 0 \\ f & 0 & 0 \\ ls & 1 & 1 \end{array}$$

int/inop (i class) — fp/fnop (f class) — ld/st (ls class)



COLLISION AUTOMATON

Transitions in a Collision Automaton

- Given a state S and any instruction *i* from an instruction class *I*
  - $S[I, 1] = 0$ implies that it is *legal* to issue *i* from *S*
  - Only legal issues have edges in the automaton
  - The collision matrix of the target state S' is produced by OR-ing collision matrices of *S* and *I*
  - When no instruction is legal to be issued from *S*, *S* is said to be *cycle-advancing*
- In any state, a NOP instruction can be issued
  - such a state behaves as a cycle-advancing state, only when a NOP is issued (not otherwise)

Cycle-advancing State

- Collision matrix is produced by left-shifting by one column, the collision matrix of *S*
- Such a state represents start of a new clock tick in all pipelines
- In single instruction issue processors, all states are *cycle-advancing*
- Start state is *cycle-advancing*
- States in which NOP is issued behave like a cycle-advancing state

Instruction Scheduling with Collision Automaton

1. Start at the *Start* state of the automaton
2. Pick instructions one by one, in priority order from the ready list
3. If it is legal to issue the picked instruction in the current state (i.e., cycle), issue it; there is no advancement of the cycle counter
4. Change state, compute collision matrix, update ready list and repeat the steps 2-3-4
5. If no instructions in the ready list are legal to be issued in a state, then insert a NOP in the output and compute the collision matrix as explained above for cycle-advancing states, and advance the cycle counter; goto step 2

Note: If step 5 is executed repeatedly, start state will be reached at some point and in the start state, all resources will be available

Optimal Instruction Scheduling using Integer Linear Programming

- This is useful for the evaluation of instruction scheduling heuristics that do not generate optimal schedules
- Careful implementation may enable these methods to be deployed even in production quality compilers
- Assume a simple resource model in which all the functional units are fully pipelined
- Assume an architecture with integer ALU, FP add unit, FP mult/div unit, and load/store unit with possibly differing execution latencies
- Assume that there are R_r instances of the functional unit r

Optimal Instruction Scheduling using Integer Linear Programming

- Let σ_i be the time at which instruction i is scheduled
- Let $d_{(i,j)}$ be the weight of the edge (i,j) of the DAG
- To satisfy dependence constraints, for each arc (i,j) of the DAG

$$\sigma_j \geq \sigma_i + d_{(i,j)} \tag{1}$$

- A matrix $K_{n \times T}$, where n is the number of instructions in the DAG and T is an estimate of the worst case execution time of the schedule, is used
  - T can be estimated by summing up the execution times of all the instructions in the DAG
- $K[i,t]$ is 1, if instruction i is scheduled at time step t and 0 otherwise

Optimal Instruction Scheduling using Integer Linear Programming

- The schedule time σ_i of instruction i can be expressed as

$$\sigma_i = k_{i,0} \cdot 0 + k_{i,1} \cdot 1 + \cdots + k_{i,T-1} \cdot (T-1)$$

  where exactly one of the $k_{i,j}$ is 1
- This can be written in matrix form for all σ_i's as:

$$\begin{bmatrix} \sigma_0 \\ \sigma_1 \\ \vdots \\ \sigma_{n-1} \end{bmatrix} = \begin{bmatrix} k_{0,0} & k_{0,1}, & \cdots & k_{0,T-1} \\ k_{1,0} & k_{1,1} & \cdots & k_{1,T-1} \\ \vdots & \vdots & \vdots & \vdots \\ k_{n-1,0} & k_{n-1,1} & \cdots & k_{n-1,T-1} \end{bmatrix} * \begin{bmatrix} 0 \\ 1 \\ \vdots \\ T-1 \end{bmatrix} \tag{2}$$

- To express that each instruction is scheduled exactly once, we include the constraint

$$\sum_t k_{i,t} = 1, \quad \forall i \tag{3}$$

Optimal Instruction Scheduling using Integer Linear Programming

- The resource constraint that no more than R_r instructions are scheduled in any time step can be expressed as

$$\sum_{i \in F(r)} k_{i,t} \leq R_r, \quad \forall\, t \;\text{ and } \forall\, r \tag{4}$$

  where $F(r)$ represents the set of instructions that can be executed in functional unit type r.
- The objective function is to minimize the execution time or schedule length, subject to the constraints in equations 1-4 above. This can be represented as:

$$\text{minimize}(\max_i(\sigma_i + d_{(i,j)}))$$

Delayed Load Scheduling Algorithm for Trees

- RISC load/store architecture with delayed loads
- Single cycle issue/execution, with only loads pipelined (load delay = 1 cycle)
- Generates optimal code without any interlocks for expression trees
- Three phases
    - Computation of *minReg* as in Sethi-Ullman code generation algorithm
    - Ordering of loads and operations as in the SU algorithm
    - Emitting code in canonical DLS order
- Uses $1 + minReg$ number of registers and can handle only one cycle load delay

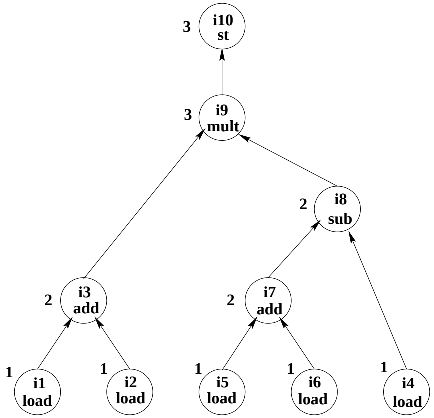
Sethi-Ullman minReg Computation Algorithm

```
procedure label (node){
if (isLeaf(node)) then {node.minReg = 1}
else { label(node.left); label(node.right);
  if (node.left.minReg == node.right.minReg) then
     {node.minReg = node.left.minReg + 1}
  else {node.minReg = MAX(node.left.minReg,
                        node.right.minReg)}
}
}
```

Sethi-Ullman minReg Computation Example

i1:	t1	←	load a
i2:	t2	←	load b
i3:	t3	←	t1 + t2
i4:	t4	←	load c
i5:	t5	←	load a
i6:	t6	←	load b
i7:	t7	←	t5 + t6
i8:	t8	←	t7 - t4
i9:	t9	←	t3 * t8
i10:	d	←	st t9

(a) 3-Address Code



(b) Expression Tree

Sethi-Ullman Algorithm Code Gen Example

i1:	r1	←	load a
i2:	r2	←	load b
i3:	r1	←	r1 + r2
i4:	r2	←	load c
i5:	r3	←	load a
i6:	r4	←	load b
i7:	r3	←	r3 + r4
i8:	r2	←	r3 - r2
i9:	r1	←	r1 * r2
i10:	d	←	st r1

(a) Code Sequence using 4 Registers

i5:	r1	←	load a
i6:	r2	←	load b
i7:	r1	←	r1 + r2
i4:	r2	←	load c
i8:	r1	←	r1 - r2
i1:	r2	←	load a
i2:	r3	←	load b
i3:	r2	←	r2 + r3
i9:	r1	←	r1 * r2
i10:	d	←	st r1

(b) Optimal Code Sequence with 3 Registers

DLS Computation Example

i5:	r1	$\leftarrow$	load a	
i6:	r2	$\leftarrow$	load b	
i7:	r1	$\leftarrow$	r1 + r2	% 1 stall
i4:	r2	$\leftarrow$	load c	
i8:	r1	$\leftarrow$	r1 - r2	% 1 stall
i1:	r2	$\leftarrow$	load a	
i2:	r3	$\leftarrow$	load b	
i3:	r2	$\leftarrow$	r2 + r3	% 1 stall
i9:	r1	$\leftarrow$	r1 * r2	
i10:	d	$\leftarrow$	st r1	

(a) Stalls in Sethi-Ullman Sequence

i5:	r1	$\leftarrow$	load a
i6:	r2	$\leftarrow$	load b
i4:	r3	$\leftarrow$	load c
i1:	r4	$\leftarrow$	load a
i7:	r1	$\leftarrow$	r1 + r2
i2:	r2	$\leftarrow$	load b
i8:	r1	$\leftarrow$	r1 - r3
i3:	r4	$\leftarrow$	r4 + r2
i9:	r1	$\leftarrow$	r1 * r4
i10:	d	$\leftarrow$	st r1

(b) DLS Sequence with No Stalls

DLS Algorithm - Main Program

```
Procedure Generate(root: ExprNode)
{ label(root); //Calculate minReg values
  opSched = loadSched = emptyList(); //Initialize
  order(root, opSched, loadSched);
  //Find load and operation order
  schedule(opSched, loadSched, root.minReg+1);
  //Emit canonical order
}
```

DLS Algorithm - Finding SU Order

```
Procedure Order(root: ExprNode;
                var opSched, loadSched: NodeList)
{ if (not(isLeaf(root))
    { if (root.left.minReg < root.right.minReg)
        { order(root.right, opSched, loadSched);
          order(root.left, opSched, loadSched);
        } else
           {order(root.left, opSched, loadSched);
            order(root.right, opSched, loadSched);
           }
        append(root, opSched);
    }
  else { append(root, loadSched);
}
```

DLS Algorithm - Scheduling

```
Procedure schedule(opSched, loadSched: NodeList;
                          Regs: integer)
{ for i = 1 to MIN(Regs, length(loadSched)) do
  // loads first
  { ld = popHead(loadSched);
    ld.reg = getReg(); gen(Load, ld.name, ld.Reg)}
  while (not Empty(loadSched))
  // (Operation,Load) pairs next
  { op = popHead(opSched); op.reg = op.left.reg;
    gen(op.op, op.left.reg, op.right.reg, op.reg);
    ld = popHead(loadSched); ld.reg = op.right.reg;
    gen(Load, ld.name, ld.reg) }
  while (not Empty(opSched)) //Remaining Operations
  { op = popHead(opSched); op.reg = op.left.reg;
    gen(op.op, op.left.reg, op.right.reg, op.reg);
    freeReg(op.right.reg) }
}
```

Global Acyclic Scheduling

- Average size of a basic block is quite small (5 to 20 instructions)
    - Effectiveness of instruction scheduling is limited
    - This is a serious concern in architectures supporting greater ILP
        - VLIW architectures with several function units
        - superscalar architectures (multiple instruction issue)
- Global scheduling is for a set of basic blocks
    - Overlaps execution of successive basic blocks
    - Trace scheduling, Superblock scheduling, Hyperblock scheduling, Software pipelining, etc.

Trace Scheduling

- A Trace is a frequently executed acyclic sequence of basic blocks in a CFG (part of a path)
- Identifying a trace
  - Identify the most frequently executed basic block
  - Extend the trace starting from this block, forward and backward, along most frequently executed edges
- Apply list scheduling on the trace (including the branch instructions)
- Execution time for the trace may reduce, but execution time for the other paths may increase
- However, overall performance will improve

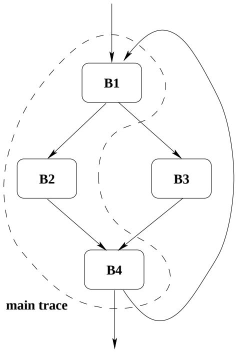
Trace Example

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

		%% r1 ← 0
		%% r5 ← 0
		%% r6 ← 400
		%% r7 ← s
B1:	i1:	r2 ← load a(r1)
	i2:	if (r2 != 0) goto i7
B2:	i3:	r3 ← load b(r1)
	i4:	r4 ← r3 + r7
	i5:	b(r1) ← r4
	i6:	goto i9
B3:	i7:	r4 ← r2
	i8:	b(r1) ← r2
B4:	i9:	r5 ← r5 + r4
	i10:	r1 ← r1 + 4
	i11:	if (r1 < r6) goto i1

(b) Assembly Code



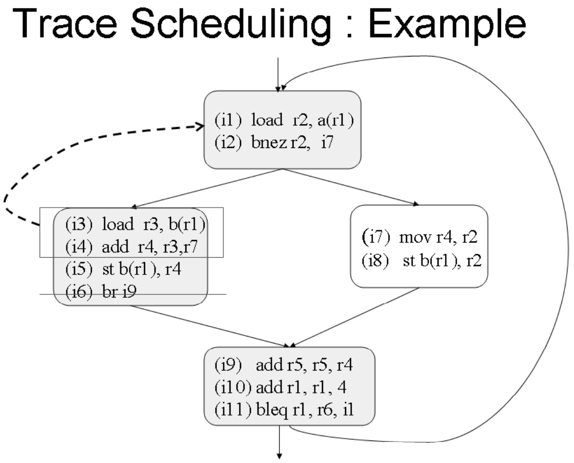
(c) Control Flow Graph

Trace - Basic Block Schedule

- 2-way issue architecture with 2 integer units
- *add, sub, store*: 1 cycle, *load*: 2 cycles, *goto*: no stall
- 9 cycles for the main trace and 6 cycles for the off-trace

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 $\leftarrow$ load a(r1)		
1				
2	i2:	if (r2 != 0) goto i7		
3	i3:	r3 $\leftarrow$ load b(r1)		
4				
5	i4:	r4 $\leftarrow$ r3 + r7		
6	i5:	b(r1) $\leftarrow$ r4	i6:	goto i9
3	i7:	r4 $\leftarrow$ r2	i8:	b(r1) $\leftarrow$ r2
7 (4)	i9:	r5 $\leftarrow$ r5 + r4	i10:	r1 $\leftarrow$ r1 + 4
8 (5)	i11:	if (r1 $<$ r6) goto i1		

Trace Schedule



Trace Schedule

- 6 cycles for the main trace and 7 cycles for the off-trace

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2:	if (r2 != 0) goto i7	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4		
4 (5)	i9:	r5 ← r5 + r4	i10:	r1 ← r1 + 4
5 (6)	i11:	if (r1 < r6) goto i1		

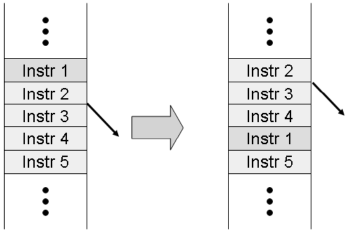
3	i7:	r4 ← r2	i8:	b(r1) ← r2
4	i12:	goto i9		

Trace Scheduling - Issues

- *Side exits* and *side entrances* are ignored during scheduling of a trace
- Required compensation code is inserted during book-keeping (after scheduling the trace)
- Speculative code motion - *load* instruction moved ahead of conditional branch
  - Example: Register r3 should not be live in block B3 (off-trace path)
  - May cause unwanted exceptions
    - Requires additional hardware support!

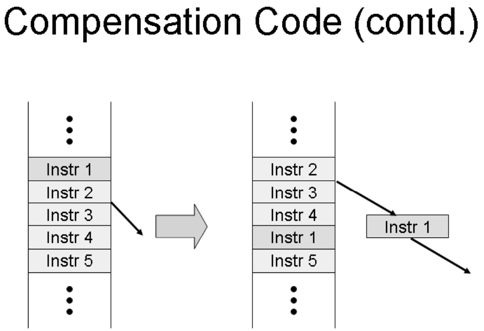
Compensation Code - Side Exit

Compensation Code



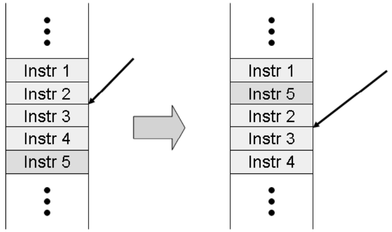
What compensation code is required when Instr 1 is moved below the side exit in the trace?

Compensation Code - Side Exit


Compensation Code (contd.)

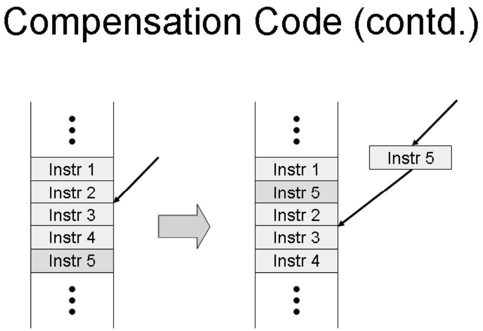
Compensation Code - Side Entry

Compensation Code (contd.)



What compensation code is required when Instr 5 moves above the side entrance in the trace?

Compensation Code - Side Entry

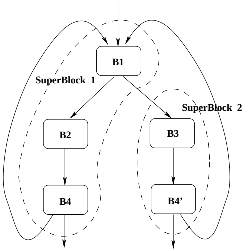


Superblock Scheduling

- A Superblock is a trace without side entrances
  - Control can enter only from the top
  - Many exits are possible
  - Eliminates several book-keeping overheads
- Superblock formation
  - Trace formation as before
  - Tail duplication to avoid side entrances into a superblock
  - Code size increases

Superblock Example

- 5 cycles for the main trace and 6 cycles for the off-trace



(a) Control Flow Graph

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2:	if (r2!=0) goto i7	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4	i10:	r1 ← r1 + 4
4	i9:	r5 ← r5 + r4	i11:	if (r1<r6) goto i1

3	i7:	r4 ← r2	i8:	b(r1) ← r2
4	i9':	r5 ← r5 + r4	i10':	r1 ← r1 + 4
5	i11':	if (r1<r6) goto i1		

(b) Superblock Schedule

Hyperblock Scheduling

- Superblock scheduling does not work well with control-intensive programs which have many control flow paths
- Hyperblock scheduling was proposed to handle such programs
- Here, the control flow graph is IF-converted to eliminate conditional branches
- IF-conversion replaces conditional branches with appropriate predicated instructions
- Now, control dependence is changed to a data dependence

IF-Conversion Example

```
for I = 1 to 100 do {
   if (A(I) <= 0) then contnue
   A(I) = B(I) + 3
}
```

```
for I = 1 to 100 do {
   p = (A(I) <= 0)
   (p)  A(I) = B(I) + 3
}
```

```
        for I = 1 to N do {
S1:        A(I) = D(I) + 1
S2:        if (B(I) > 0) then
S3:           C(I) = C(I) + A(I)
S4:         else   D(I+1) = D(I+1) + 1
           end if
        }
```

```
        for I = 1 to N do {
S1:        A(I) = D(I) + 1
S2:        p = (B(I) > 0)
S3:        (p)  C(I) = C(I) + A(I)
S4:        (!p)  D(I+1) = D(I+1) + 1
        }
```

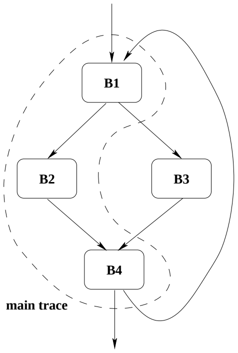
Hyperblock Example Code

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

		%% r1 ← 0
		%% r5 ← 0
		%% r6 ← 400
		%% r7 ← s
B1:	i1:	r2 ← load a(r1)
	i2:	if (r2 != 0) goto i7
B2:	i3:	r3 ← load b(r1)
	i4:	r4 ← r3 + r7
	i5:	b(r1) ← r4
	i6:	goto i9
B3:	i7:	r4 ← r2
	i8:	b(r1) ← r2
B4:	i9:	r5 ← r5 + r4
	i10:	r1 ← r1 + 4
	i11:	if (r1 < r6) goto i1

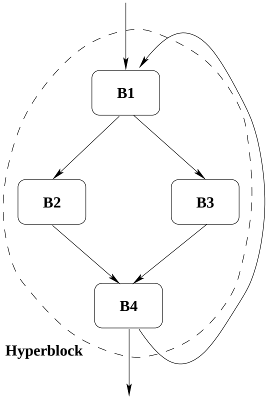
(b) Assembly Code



(c) Control Flow Graph

Hyperblock Example

- 6 cycles for the entire set of predicated instructions
- Instructions i3 and i4 can be executed speculatively and can be moved up, instead of being scheduled after cycle 2



(a) Control Flow Graph

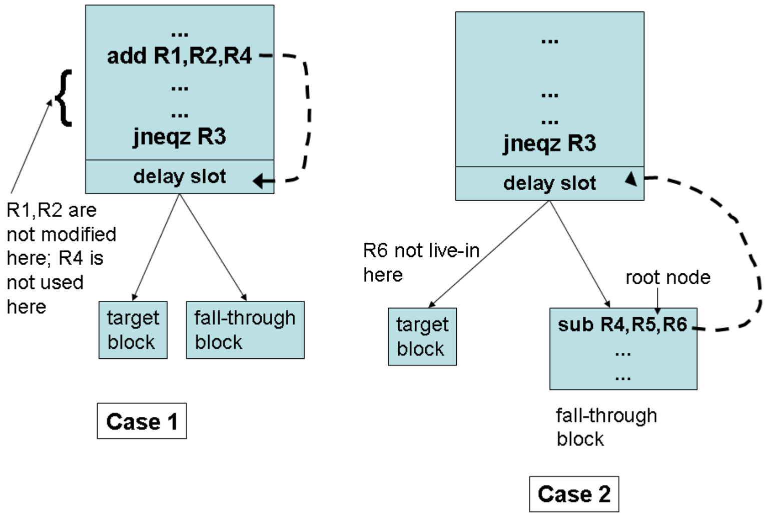
Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2':	p1 ← (r2 == 0)	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4, if p1	i8:	b(r1) ← r2, if !p1
4	i10:	r1 ← r1 + 4	i7:	r4 ← r2, if !p1
5	i9:	r5 ← r5 + r4	i11:	if (r1<r6) goto i1

(b) Hyperblock Schedule

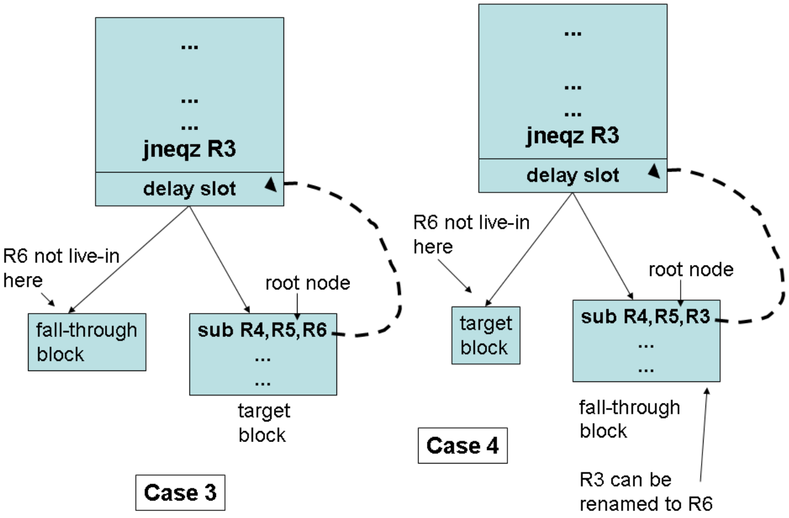
Delayed Branch Scheduling

- Delayed branching
  - One instruction immediately following the delayed branch instruction will be executed before the branch is taken
  - The instruction occupying the delay slot should be *independent* of the branch instruction
- It is best to fill the branch delay slot with an instruction from the basic block that the branch terminates
- Otherwise, an instruction from either the target block or the fall-through block, whichever is most likely to be executed, is selected
  - The selected instruction should either be a *root node* of the DAG of the basic block (target of fall-through)
  - and has a destination register that is not live-in in the other block
  - or has a destination register that can be renamed

Delay Branch Scheduling Conditions - 1



Delay Branch Scheduling Conditions - 2



...
...
...
jneqz R3
delay slot
R6 not live-in here
fall-through block
root node
sub R4,R5,R6
...
...
target block
Case 3
...
...
...
jneqz R3
delay slot
R6 not live-in here
target block
root node
sub R4,R5,R3
...
...
fall-through block
Case 4
R3 can be renamed to R6