Instruction Scheduling and Software Pipelining - 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline

- Instruction Scheduling
  - Simple Basic Block Scheduling
  - Trace, Superblock and Hyperblock scheduling
- Software pipelining

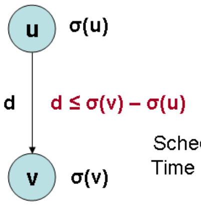
Basic Block Scheduling

- 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 σ(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

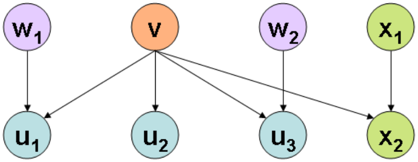
🔴 At i=2, $\varsigma_{v3}(0)+\varsigma_{v2}(1)+\varsigma_{v1}(2) = 3+3+2 = 8 > R$, NOT satisfied

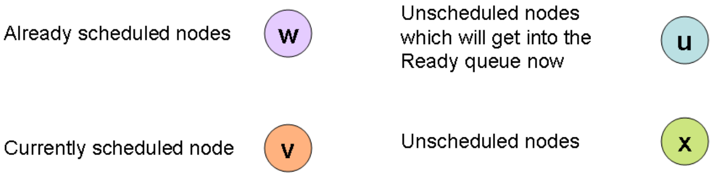
A Simple List Scheduling Algorithm

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



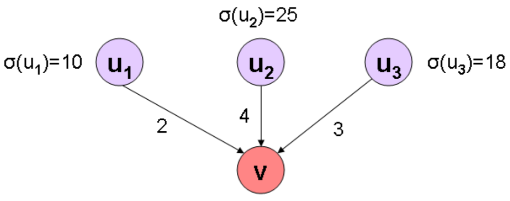


Constraint Satisfaction Functions

```
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 $\sigma(v)$ = 29

Already scheduled nodes

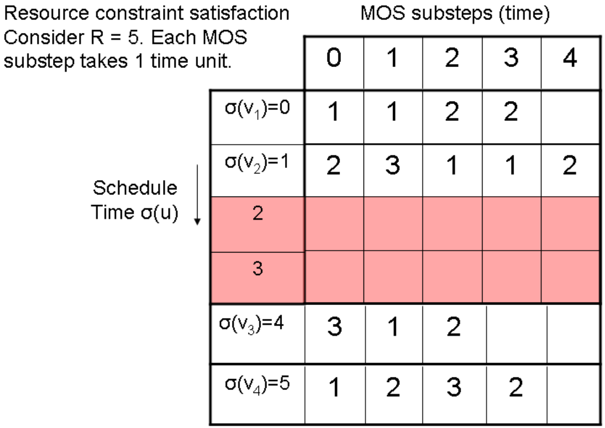
Node to be scheduled

Precedence constraint satisfaction:

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

Lower bound for $\sigma(v)$
= 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.

MOS substeps (time)

Schedule Time $\sigma(u)$

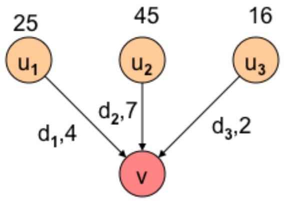
	0	1	2	3	4
$\sigma(v_1)=0$	1	1	2	2	
$\sigma(v_2)=1$	2	3	1	1	2
2					
3					
$\sigma(v_3)=4$	3	1	2		
$\sigma(v_4)=5$	1	2	3	2	

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

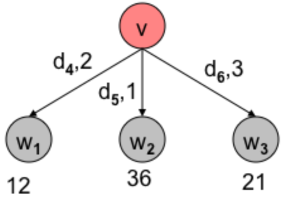
1. Height of the node in the DAG (*i.e.,* longest path from the node to a terminal node
2. *Estart*, and *Lstart*, the earliest and latest start times
   - Violating *Estart* and *Lstart* may result in pipeline stalls
   - $Estart(v) = \max_{i=1,\cdots,k} (Estart(u_i) + d(u_i, v))$ where $u_1, u_2, \cdots, u_k$ 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 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 Computation



$$Lstart\ (v) = \min_{i=4,\ldots,6} (Lsart\ (w_i) - d_i)$$

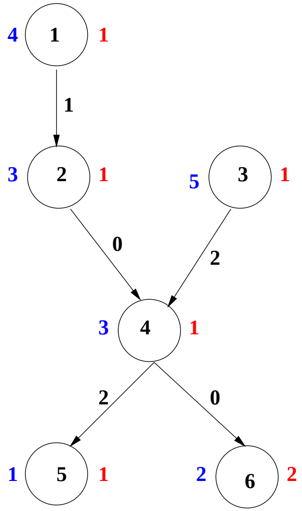
$$= \min(12-2, 36-1, 21-3)$$
$$= \min(10, 35, 18) = 10$$

List Scheduling - Slack

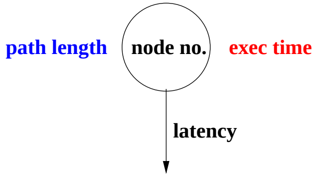
1. A node with a lower *Estart* (or *Lstart*) value has a higher priority
2. $Slack = Lstart - Estart$
    - Nodes with lower slack are given higher priority
    - Instructions on the critical path may have a slack value of zero and hence get priority

Simple List Scheduling - Example - 1

INSTRUCTION SCHEDULING - EXAMPLE



LEGEND



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

= max { latency (n,m) + path length (m) }
m ε succ (n)

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

Simple List Scheduling - Example - 2

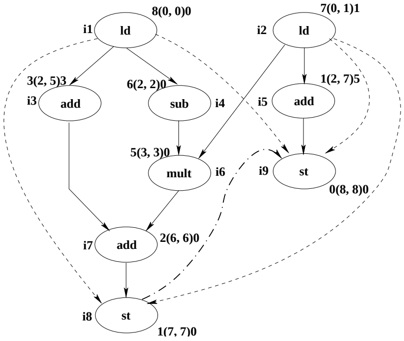
- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
- *path length* and *slack* are shown on the left side and right side of the pair of numbers in parentheses

```
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		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 - Instruction Reservation Table

	r_0	r_1	r_2	r_3	r_4
t_0	1	0	1	2	0
t_1	1	1	0	0	1
t_2	0	0	0	2	1
t_3	0	1	0	0	1

No. of resources in the machine: 4

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

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

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

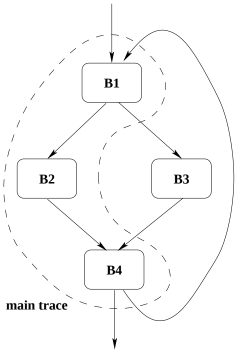
Trace Example

```
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



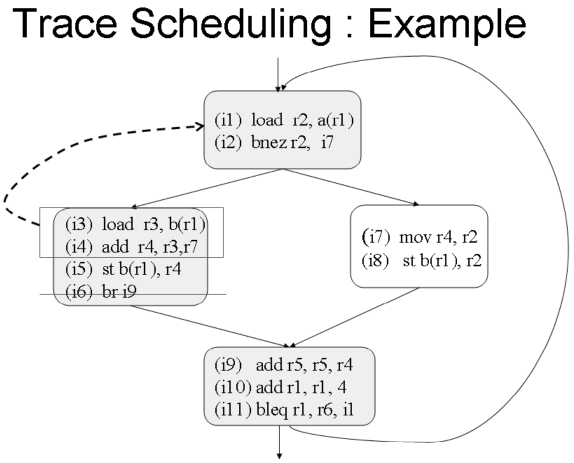
(c) Control Flow Graph

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 ← load a(r1)		
1				
2	i2:	if (r2 != 0) goto i7		
3	i3:	r3 ← load b(r1)		
4				
5	i4:	r4 ← r3 + r7		
6	i5:	b(r1) ← r4	i6:	goto i9
3	i7:	r4 ← r2	i8:	b(r1) ← r2
7 (4)	i9:	r5 ← r5 + r4	i10:	r1 ← 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		

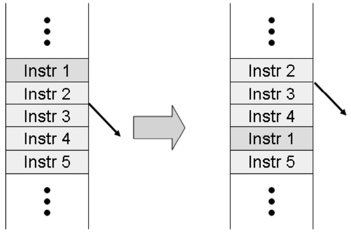
3	i7:	r4 ← r2	i8:	b(r1) ← r2
4	i12:	goto i9		

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!

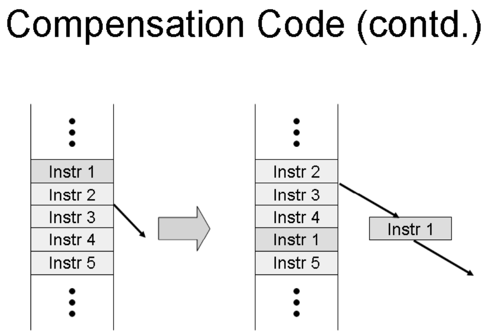
Compensation Code - Side Exit

Compensation Code

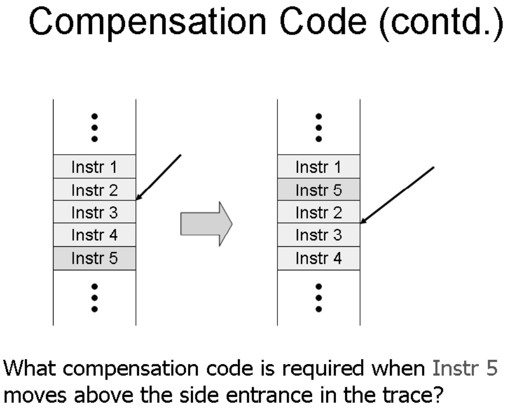


What compensation code is required when Instr 1 is moved below the side exit in the trace?

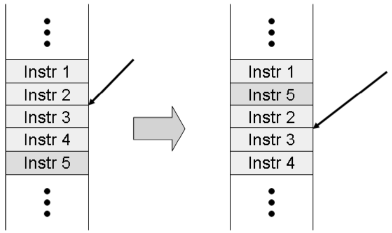
Compensation Code - Side Exit



Compensation Code - Side Entry



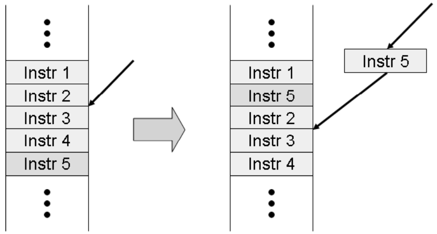
Compensation Code (contd.)



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

Compensation Code - Side Entry

Compensation Code (contd.)


Instr 1
Instr 2
Instr 3
Instr 4
Instr 5
Instr 1
Instr 5
Instr 2
Instr 3
Instr 4
Instr 5