Instruction Scheduling - Part 2

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

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)

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

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
```

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
```

List Scheduling - Priority Ordering for Nodes

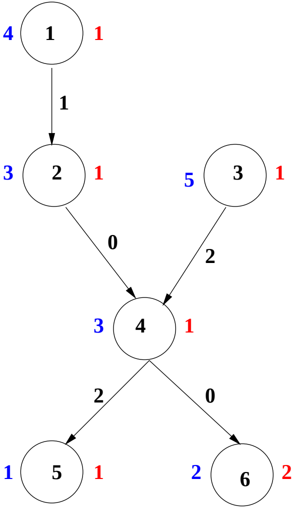
1. Height of the node in the DAG (*i.e.,* longest path from the node to a terminal node
2. *Estart*, and *Lstart*, the earliest and latest start times
    - Violating *Estart* and *Lstart* may result in pipeline stalls
    - $Estart(v) = \max_{i=1,\cdots,k} (Estart(u_i) + d(u_i, v))$ where $u_1, u_2,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

List Scheduling - Slack

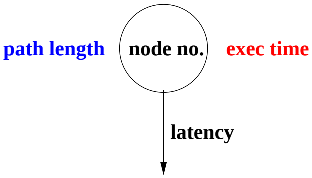
1. A node with a lower *Estart* (or *Lstart*) value has a higher priority
2. $Slack = Lstart - Estart$
    - Nodes with lower slack are given higher priority
    - Instructions on the critical path may have a slack value of zero and hence get priority

Simple List Scheduling - Example - 1

INSTRUCTION SCHEDULING - EXAMPLE



LEGEND



$$\textbf{path length}(n) = \textbf{exec time}(n)\text{, if } n \text{ is a leaf}$$

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

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

Simple List Scheduling - Example - 2

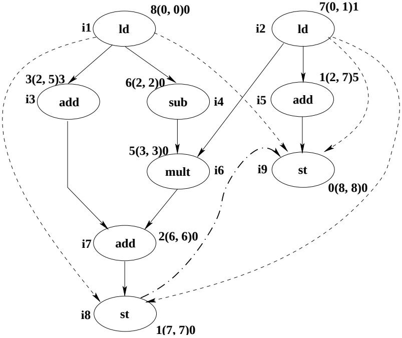
- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
- *path length* and *slack* are shown on the left side and right side of the pair of numbers in parentheses

```
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

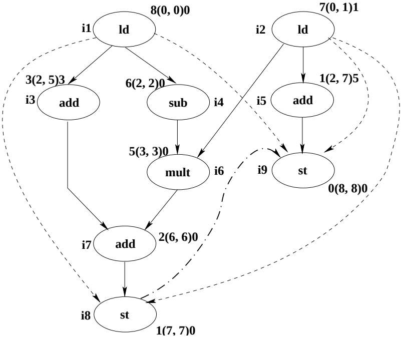
- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
  - 2 Integer units and 1 Multiplication unit, all capable of load and store as well

```
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 × 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

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

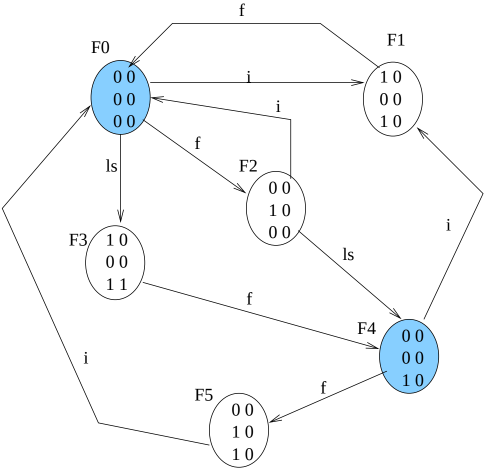
int/inop
(i class)

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

fp/fnop
(f class)

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

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 to 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