Instruction Scheduling - Part 3

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)

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

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

  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 $1 + \mathit{minReg}$ number of registers and can handle only one cycle load delay

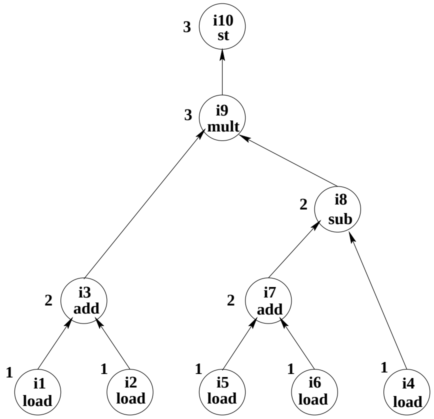
Sethi-Ullman minReg Computation Algorithm

```
if (isLeaf(node)) then {node.minReg = 1}
else
  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	←	load a	
i6:	r2	←	load b	
i7:	r1	←	r1 + r2	% 1 stall
i4:	r2	←	load c	
i8:	r1	←	r1 - r2	% 1 stall
i1:	r2	←	load a	
i2:	r3	←	load b	
i3:	r2	←	r2 + r3	% 1 stall
i9:	r1	←	r1 * r2	
i10:	d	←	st r1	

(a) Stalls in Sethi-Ullman Sequence

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

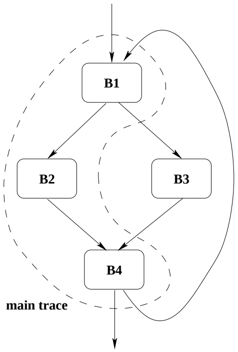
Trace Example

```
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

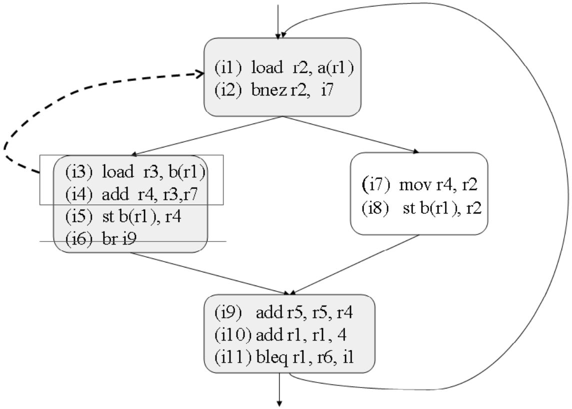
Trace - Basic Block Schedule

- 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 ← 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 Scheduling : Example



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		

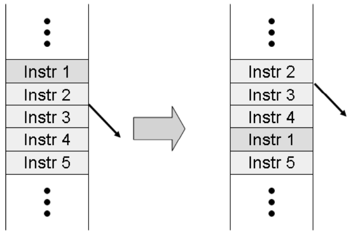
3	i7:	r4 ← r2	i8:	b(r1) ← r2
4	i12:	goto i9		

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

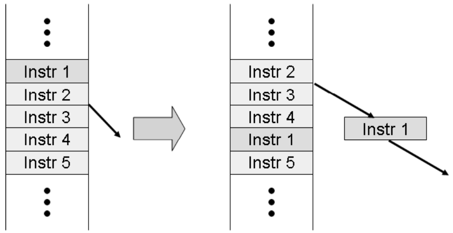
Compensation Code



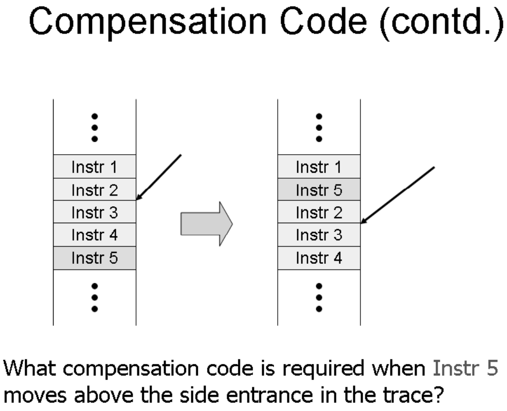
What compensation code is required when Instr 1 is moved below the side exit in the trace?

Compensation Code - Side Exit

Compensation Code (contd.)



Compensation Code - Side Entry



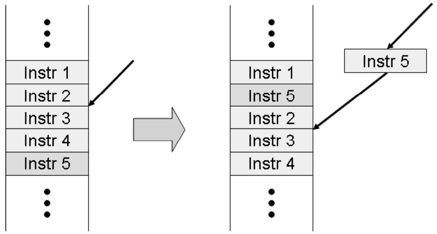
Compensation Code (contd.)

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

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

Compensation Code - Side Entry

Compensation Code (contd.)

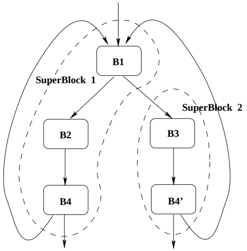


Superblock Scheduling

- 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
           }
```

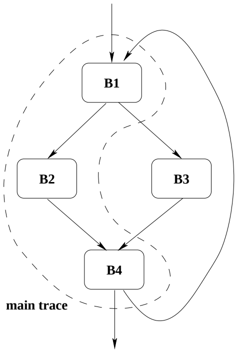
Hyperblock Example Code

```
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

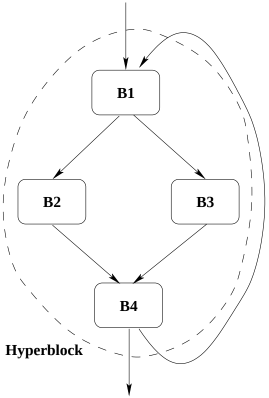
(b) Assembly Code



(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

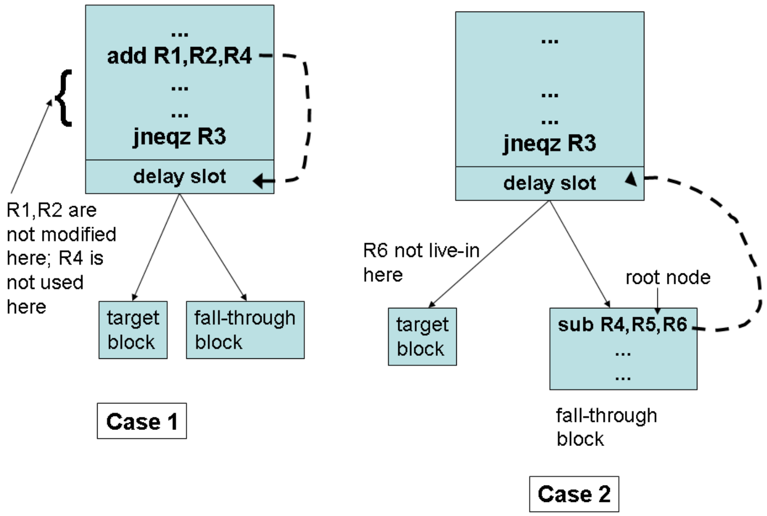
Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2':	p1 ← (r2 == 0)	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4, if p1	i8:	b(r1) ← r2, if !p1
4	i10:	r1 ← r1 + 4	i7:	r4 ← r2, if !p1
5	i9:	r5 ← r5 + r4	i11:	if (r1<r6) goto i1

(b) Hyperblock Schedule

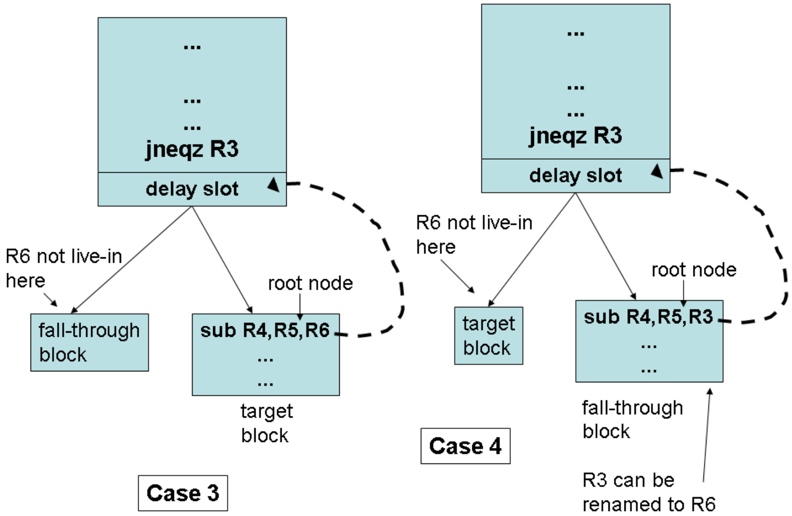
Delayed Branch Scheduling

- 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
R3 can be renamed to R6
Case 4