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 heuristics
- Applied on basic blocks (local)
- Global scheduling requires elongation of basic blocks (super-blocks)
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 $r$. 

Y.N. Srikant
Optimal Instruction Scheduling using Integer Linear Programming

- Let $\sigma_i$ 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

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

- 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
The schedule time $\sigma_i$ 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 $\sigma_i$’s as:

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

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

$$\sum_t k_{i,t} = 1, \quad \forall i$$
The resource constraint that no more than $R_r$ instructions are scheduled in any time step can be expressed as

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

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
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

(a) 3-Address Code

| 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 |

(b) Expression Tree
<table>
<thead>
<tr>
<th>i1:</th>
<th>r1 ← load a</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2:</td>
<td>r2 ← load b</td>
</tr>
<tr>
<td>i3:</td>
<td>r1 ← r1 + r2</td>
</tr>
<tr>
<td>i4:</td>
<td>r2 ← load c</td>
</tr>
<tr>
<td>i5:</td>
<td>r3 ← load a</td>
</tr>
<tr>
<td>i6:</td>
<td>r4 ← load b</td>
</tr>
<tr>
<td>i7:</td>
<td>r3 ← r3 + r4</td>
</tr>
<tr>
<td>i8:</td>
<td>r2 ← r3 - r2</td>
</tr>
<tr>
<td>i9:</td>
<td>r1 ← r1 * r2</td>
</tr>
<tr>
<td>i10:</td>
<td>d ← st r1</td>
</tr>
</tbody>
</table>

(a) Code Sequence using 4 Registers

<table>
<thead>
<tr>
<th>i5:</th>
<th>r1 ← load a</th>
</tr>
</thead>
<tbody>
<tr>
<td>i6:</td>
<td>r2 ← load b</td>
</tr>
<tr>
<td>i7:</td>
<td>r1 ← r1 + r2</td>
</tr>
<tr>
<td>i4:</td>
<td>r2 ← load c</td>
</tr>
<tr>
<td>i8:</td>
<td>r1 ← r1 - r2</td>
</tr>
<tr>
<td>i1:</td>
<td>r2 ← load a</td>
</tr>
<tr>
<td>i2:</td>
<td>r3 ← load b</td>
</tr>
<tr>
<td>i3:</td>
<td>r2 ← r2 + r3</td>
</tr>
<tr>
<td>i9:</td>
<td>r1 ← r1 * r2</td>
</tr>
<tr>
<td>i10:</td>
<td>d ← st r1</td>
</tr>
</tbody>
</table>

(b) Optimal Code Sequence with 3 Registers
### DLS Computation Example

<table>
<thead>
<tr>
<th>i5:</th>
<th>r1 ← load a</th>
<th>% 1 stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>i6:</td>
<td>r2 ← load b</td>
<td></td>
</tr>
<tr>
<td>i7:</td>
<td>r1 ← r1 + r2</td>
<td>% 1 stall</td>
</tr>
<tr>
<td>i4:</td>
<td>r2 ← load c</td>
<td></td>
</tr>
<tr>
<td>i8:</td>
<td>r1 ← r1 - r2</td>
<td>% 1 stall</td>
</tr>
<tr>
<td>i1:</td>
<td>r2 ← load a</td>
<td></td>
</tr>
<tr>
<td>i2:</td>
<td>r3 ← load b</td>
<td></td>
</tr>
<tr>
<td>i3:</td>
<td>r2 ← r2 + r3</td>
<td></td>
</tr>
<tr>
<td>i9:</td>
<td>r1 ← r1 * r2</td>
<td></td>
</tr>
<tr>
<td>i10:</td>
<td>d ← st r1</td>
<td></td>
</tr>
</tbody>
</table>

(a) Stalls in Sethi-Ullman Sequence

<table>
<thead>
<tr>
<th>i5:</th>
<th>r1 ← load a</th>
</tr>
</thead>
<tbody>
<tr>
<td>i6:</td>
<td>r2 ← load b</td>
</tr>
<tr>
<td>i4:</td>
<td>r3 ← load c</td>
</tr>
<tr>
<td>i1:</td>
<td>r4 ← load a</td>
</tr>
<tr>
<td>i7:</td>
<td>r1 ← r1 + r2</td>
</tr>
<tr>
<td>i2:</td>
<td>r2 ← load b</td>
</tr>
<tr>
<td>i8:</td>
<td>r1 ← r1 - r3</td>
</tr>
<tr>
<td>i3:</td>
<td>r4 ← r4 + r2</td>
</tr>
<tr>
<td>i9:</td>
<td>r1 ← r1 * r4</td>
</tr>
<tr>
<td>i10:</td>
<td>d ← st r1</td>
</tr>
</tbody>
</table>

(b) DLS Sequence with No Stalls
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
}
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); }
}
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.
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
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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>i1</td>
<td>r2 ← load a(r1)</td>
</tr>
<tr>
<td></td>
<td>i2</td>
<td>if (r2 != 0) goto i7</td>
</tr>
<tr>
<td>B2</td>
<td>i3</td>
<td>r3 ← load b(r1)</td>
</tr>
<tr>
<td></td>
<td>i4</td>
<td>r4 ← r3 + r7</td>
</tr>
<tr>
<td></td>
<td>i5</td>
<td>b(r1) ← r4</td>
</tr>
<tr>
<td></td>
<td>i6</td>
<td>goto i9</td>
</tr>
<tr>
<td>B3</td>
<td>i7</td>
<td>r4 ← r2</td>
</tr>
<tr>
<td></td>
<td>i8</td>
<td>b(r1) ← r2</td>
</tr>
<tr>
<td>B4</td>
<td>i9</td>
<td>r5 ← r5 + r4</td>
</tr>
<tr>
<td></td>
<td>i10</td>
<td>r1 ← r1 + 4</td>
</tr>
<tr>
<td></td>
<td>i11</td>
<td>if (r1 &lt; r6) goto i1</td>
</tr>
</tbody>
</table>

(b) Assembly Code

(c) Control Flow Graph
- 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

<table>
<thead>
<tr>
<th>Time</th>
<th>Int. Unit 1</th>
<th>Int. Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i1: r2 ← load a(r1)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>i2: if (r2 ≠ 0) goto i7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i3: r3 ← load b(r1)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>i4: r4 ← r3 + r7</td>
<td>i6: goto i9</td>
</tr>
<tr>
<td>4</td>
<td>i5: b(r1) ← r4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>i7: r4 ← r2</td>
<td>i8: b(r1) ← r2</td>
</tr>
<tr>
<td>6</td>
<td>i9: r5 ← r5 + r4</td>
<td>i10: r1 ← r1 + 4</td>
</tr>
<tr>
<td>7 (4)</td>
<td>i11: if (r1 &lt; r6) goto i1</td>
<td></td>
</tr>
<tr>
<td>8 (5)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Trace Scheduling: Example

(i1) load r2, a(r1)
(i2) bnez r2, i7

(i3) load r3, b(r1)
(i4) add r4, r3, r7
(i5) st b(r1), r4
(i6) br i9

(i7) mov r4, r2
(i8) st b(r1), r2

(i9) add r5, r5, r4
(i10) add r1, r1, 4
(i11) breq r1, r6, i1
Trace Schedule

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Int. Unit 1</th>
<th>Int. Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i1: r2 ← load a(r1)</td>
<td>i3: r3 ← load b(r1)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i2: if (r2 != 0) goto i7</td>
<td>i4: r4 ← r3 + r7</td>
</tr>
<tr>
<td>3</td>
<td>i5: b(r1) ← r4</td>
<td></td>
</tr>
<tr>
<td>4 (5)</td>
<td>i9: r5 ← r5 + r4</td>
<td>i10: r1 ← r1 + 4</td>
</tr>
<tr>
<td>5 (6)</td>
<td>i11: if (r1 &lt; r6) goto i1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>i7: r4 ← r2</td>
<td>i8: b(r1) ← r2</td>
</tr>
<tr>
<td>4</td>
<td>i12: goto i9</td>
<td></td>
</tr>
</tbody>
</table>
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

What compensation code is required when Instr 1 is moved below the side exit in the trace?
Compensation Code (contd.)

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

- Instr 2
- Instr 3
- Instr 4
- Instr 1
- Instr 5
What compensation code is required when Instr 5 moves above the side entrance in the trace?
Compensation Code (contd.)

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5

Instr 1
Instr 5
Instr 2
Instr 3
Instr 4
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
5 cycles for the main trace and 6 cycles for the off-trace

(a) Control Flow Graph

(b) Superblock Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Int. Unit 1</th>
<th>Int. Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i1: r2 ← load a(r1)</td>
<td>i3: r3 ← load b(r1)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i2: if (r2!=0) goto i7</td>
<td>i4: r4 ← r3 + r7</td>
</tr>
<tr>
<td>3</td>
<td>i5: b(r1) ← r4</td>
<td>i10: r1 ← r1 + 4</td>
</tr>
<tr>
<td>4</td>
<td>i9: r5 ← r5 + r4</td>
<td>i11: if (r1&lt;r6) goto i1</td>
</tr>
</tbody>
</table>

3  i7: r4 ← r2
4  i9’: r5 ← r5 + r4
5  i11’: if (r1<r6) goto i1
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.
for $l = 1$ to 100 do {
    if ($A(l) \leq 0$) then continue
    $A(l) = B(l) + 3$
}

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

for $l = 1$ to 100 do {
    $p = (A(l) \leq 0)$
    ($p$) $A(l) = B(l) + 3$
}

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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>i1:</td>
<td>r2 ← load a(r1)</td>
<td>i2:</td>
<td>if (r2 != 0) goto i7</td>
</tr>
<tr>
<td>B2</td>
<td>i3:</td>
<td>r3 ← load b(r1)</td>
<td>i4:</td>
<td>r4 ← r3 + r7</td>
</tr>
<tr>
<td>B3</td>
<td>i5:</td>
<td>b(r1) ← r4</td>
<td>i6:</td>
<td>goto i9</td>
</tr>
<tr>
<td>B4</td>
<td>i7:</td>
<td>r4 ← r2</td>
<td>i8:</td>
<td>b(r1) ← r2</td>
</tr>
<tr>
<td></td>
<td>i9:</td>
<td>r5 ← r5 + r4</td>
<td>i10:</td>
<td>r1 ← r1 + 4</td>
</tr>
<tr>
<td></td>
<td>i11:</td>
<td>if (r1 &lt; r6) goto i1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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

(b) Hyperblock Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Int. Unit 1</th>
<th>Int. Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i1: ( r2 \leftarrow \text{load a(r1)} )</td>
<td>i3: ( r3 \leftarrow \text{load b(r1)} )</td>
</tr>
<tr>
<td>1</td>
<td>i2': ( p1 \leftarrow (r2 == 0) )</td>
<td>i4: ( r4 \leftarrow r3 + r7 )</td>
</tr>
<tr>
<td>2</td>
<td>i5: ( b(r1) \leftarrow r4, \text{if } p1 )</td>
<td>i8: ( b(r1) \leftarrow r2, \text{if } !p1 )</td>
</tr>
<tr>
<td>3</td>
<td>i10: ( r1 \leftarrow r1 + 4 )</td>
<td>i7: ( r4 \leftarrow r2, \text{if } !p1 )</td>
</tr>
<tr>
<td>4</td>
<td>i9: ( r5 \leftarrow r5 + r4 )</td>
<td>i11: if (r1&lt;r6) goto i1</td>
</tr>
</tbody>
</table>

Y.N. Srikant
Instruction Scheduling
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

Case 1

- add R1, R2, R4
- ...
- jneqz R3

R1, R2 are not modified here; R4 is not used here

target block
fall-through block

Case 2

- jneqz R3

R6 not live-in here

root node

sub R4, R5, R6
- ...

fall-through block
Delay Branch Scheduling Conditions - 2

Case 3

R6 not live-in here

fall-through block

root node

sub R4, R5, R6

... 

... 

delay slot

Case 4

R6 not live-in here

root node

sub R4, R5, R3

... 

... 

delay slot

R3 can be renamed to R6