CS415 Compilers

Instruction Scheduling and Lexical Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time.

The Concept

Machine description

slow code → Scheduler → fast code

The task

- Produce correct code
- Minimize wasted (idle) cycles
- Operate efficiently
Dependences ⇒ defined on memory locations / registers and not values

Statement/instruction \( b \) depends on statement/instruction \( a \) if there exists:

- **true** of flow dependence
  
  \( a \) writes a location/register that \( b \) later reads  
  (RAW conflict)

- **anti** dependence
  
  \( a \) reads a location/register that \( b \) later writes  
  (WAR conflict)

- **output** dependence
  
  \( a \) writes a location/register that \( b \) later writes  
  (WAW conflict)

Dependences define ORDER CONSTRAINTS that need to be respected in order to generate correct code.

<table>
<thead>
<tr>
<th>true</th>
<th>anti</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = )</td>
<td>( = a )</td>
<td>( a = )</td>
</tr>
<tr>
<td>( = a )</td>
<td>( a = )</td>
<td>( a = )</td>
</tr>
</tbody>
</table>
To capture properties of the code, build a dependence graph $G$

- Nodes $n \in G$ are operations with $\text{type}(n)$ and $\text{delay}(n)$
- An edge $e = (n_1, n_2) \in G$ iff $n_2$ depends on $n_1$

The Code

- $a$: loadAl $r_0, @w$ $\Rightarrow r_1$
- $b$: add $r_1, r_1$ $\Rightarrow r_1$
- $c$: loadAl $r_0, @x$ $\Rightarrow r_2$
- $d$: mult $r_1, r_2$ $\Rightarrow r_1$
- $e$: loadAl $r_0, @y$ $\Rightarrow r_3$
- $f$: mult $r_1, r_3$ $\Rightarrow r_1$
- $g$: loadAl $r_0, @z$ $\Rightarrow r_2$
- $h$: mult $r_1, r_2$ $\Rightarrow r_1$
- $i$: storeAl $r_1$ $\Rightarrow r_0, @w$

The Dependence Graph

(all output dependences are covered, i.e., are satisfied through other dependences)
Instruction Scheduling

The big picture
1. Build a dependence graph, $P$
2. Compute a priority function over the nodes in $P$
3. Use list scheduling to construct a schedule, one cycle at a time
   (can only issue/schedule at most one instructions per cycle)
   a. Use a queue of operations that are ready
   b. At each cycle
      I. Choose a ready operation and schedule it
      II. Update the ready queue

Local list scheduling
• The dominant algorithm for twenty years
• A greedy, heuristic, local technique
Local (Forward) List Scheduling

Cycle ← 1
Ready ← leaves of P
Active ← Ø

while (Ready ∪ Active ≠ Ø)
    if (Ready ≠ Ø) then
        remove an op from Ready
        S(op) ← Cycle
        Active ← Active ∪ op
    Cycle ← Cycle + 1
    for each op ∈ Active
        if (S(op) + delay(op) ≤ Cycle) then
            remove op from Active
            for each successor s of op in P
                if (s is ready) then
                    Ready ← Ready ∪ s

Removal in priority order
op has completed execution
If successor’s operands are ready, put it on Ready
### Scheduling Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>3</td>
</tr>
<tr>
<td>loadl</td>
<td>1</td>
</tr>
<tr>
<td>loadAl</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>3</td>
</tr>
<tr>
<td>storeAl</td>
<td>3</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
</tr>
<tr>
<td>mult</td>
<td>2</td>
</tr>
<tr>
<td>fadd</td>
<td>1</td>
</tr>
<tr>
<td>fmult</td>
<td>2</td>
</tr>
<tr>
<td>shift</td>
<td>1</td>
</tr>
<tr>
<td>branch</td>
<td>0 to 8</td>
</tr>
</tbody>
</table>

- **Loads & stores may or may not block**
  - Non-blocking ⇒ fill those issue slots
- **Branches typically have delay slots**
  - Fill slots with operations unrelated to branch condition evaluation
  - Percolates branch upward
- **Branch Prediction may hide branch latencies** (hardware feature)

**Build a simple local scheduler (basic block)**

- non-blocking loads & stores
- different latencies load/store vs. arith. etc. operations
- different heuristics
- forward / backward scheduling
1. Build the dependence graph

The Code

The Dependence Graph

⇒ 20 cycles
1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

The Code

```
a:   loadAI  r0,@w  ⇒  r1
b:   add      r1,r1  ⇒  r1
c:   loadAI  r0,@x  ⇒  r2
d:   mult     r1,r2  ⇒  r1
e:   loadAI  r0,@y  ⇒  r3
f:   mult     r1,r3  ⇒  r1
g:   loadAI  r0,@z  ⇒  r2
h:   mult     r1,r2  ⇒  r1
i:   storeAI r1      ⇒  r0,@w
```

The Dependence Graph
1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

a: loadAI r0,@w \(\Rightarrow\) r1
b: add r1,r1 \(\Rightarrow\) r1
c: loadAI r0,@x \(\Rightarrow\) r2
d: mult r1,r2 \(\Rightarrow\) r1
e: loadAI r0,@y \(\Rightarrow\) r3
f: mult r1,r3 \(\Rightarrow\) r1
g: loadAI r0,@z \(\Rightarrow\) r2
h: mult r1,r2 \(\Rightarrow\) r1
i: storeAI r1 \(\Rightarrow\) r0,@w

Note: Here we assume that operation has to finish to satisfy an anti dependence. Our ILOC simulator takes only one cycle to satisfy an anti dependence since read-stage is executed before write stage.
1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

The Code

\begin{verbatim}
a:  loadAl   r0,@w  ⇒  r1
b:  add       r1,r1  ⇒  r1
c:  loadAl    r0,@x  ⇒  r2
d:  mult      r1,r2  ⇒  r1
e:  loadAl    r0,@y  ⇒  r3
f:  mult      r1,r3  ⇒  r1
g:  loadAl    r0,@z  ⇒  r2
h:  mult      r1,r2  ⇒  r1
i:  storeAl   r1     ⇒  r0,@w
\end{verbatim}

The Dependence Graph

Note: Here we assume that operation has to finish to satisfy an anti dependence. Our ILOC simulator takes only one cycle to satisfy an anti dependence since read-stage is executed before write stage (EaC).
1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

The Code

- a: loadAI r0, @w → r1
- b: add r1, r1 → r1
- c: loadAI r0, @x → r2
- d: mult r1, r2 → r1
- e: loadAI r0, @y → r3
- f: mult r1, r3 → r1
- g: loadAI r0, @z → r2
- h: mult r1, r2 → r1
- i: storeAI r1 → r0, @w

The Dependence Graph
1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

```plaintext
1. a: loadAl r0,@w ⇔ r1
2. c: loadAl r0,@x ⇔ r2
3. e: loadAl r0,@y ⇔ r3
4. b: add r1,r1 ⇔ r1
5. d: mult r1,r2 ⇔ r1
7. g: loadAl r0,@z ⇔ r2
8. f: mult r1,r3 ⇔ r1
10. h: mult r1,r2 ⇔ r1
12. i: storeAl r1 ⇔ r0,@w

⇒ 14 cycles
```

Our ILOC simulator takes only one cycle to satisfy an anti dependence.
More on Scheduling

Forward list scheduling
- start with available ops
- work forward
- ready ⇒ all operands available

Backward list scheduling
- start with no successors
- work backward
- ready ⇒ latency covers operands

Different heuristics (forward) based on **Dependence Graph**
1. Longest latency weighted path to root (⇒ critical path)
2. Highest latency instructions (⇒ more overlap)
3. Most immediate successors (⇒ create more candidates)
4. Most descendents (⇒ create more candidates)
5. ...

Interactions with register allocation
- perform dynamic register renaming (⇒ may require spill code)
- move life ranges around (⇒ may remove or require spill code)
- ...
More Lexical Analysis; Syntax Analysis

Read EaC: Chapters 2.1 – 2.5; 3.1 – 3.3

Homework Problem Set 2 is posted.