CS415 Compilers

Instruction Scheduling
Part 3

Lexical Analysis
Part 1

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

- First project and second homework have been posted

- First quiz will be posted over the weekend. Topic: register allocation; you will have a window of a few days to complete the quiz (30 – 45 minutes)

- Please attend recitations!
Instruction Scheduling

**EaC Chapter 12**
12.1 - 12.3

Part of the compiler’s back end

**Diagram:**
- **Instruction Selection**
  - IR
  - $m$ register
- **Register Allocation**
  - IR
  - $k$ register
- **Instruction Scheduling**
  - IR
- **Machine code**
<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>3</td>
</tr>
<tr>
<td>loadI</td>
<td>1</td>
</tr>
<tr>
<td>loadAI</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>3</td>
</tr>
<tr>
<td>storeAI</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>output</td>
<td>1</td>
</tr>
<tr>
<td>outputAI</td>
<td>1</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
Local (Forward) List Scheduling

\[
\text{Cycle} \leftarrow 0 \\
\text{Ready} \leftarrow \text{leaves of } P \\
\text{Active} \leftarrow \emptyset \\
\]

while (Ready \cup \text{Active} \neq \emptyset)
  if (\text{Ready} \neq \emptyset) then
    remove an \text{op} from \text{Ready}
    \text{S(\text{op})} \leftarrow \text{Cycle}
    \text{Active} \leftarrow \text{Active} \cup \text{op}
  \]

\text{Cycle} \leftarrow \text{Cycle} + 1

for each \text{op} \in \text{Active}
  if (\text{S(\text{op})} + \text{delay(\text{op})} \leq \text{Cycle}) then
    remove \text{op} from \text{Active}
    for each successor \text{s} of \text{op} in \text{P}
      if (\text{s} is ready) then
        \text{Ready} \leftarrow \text{Ready} \cup \text{s}

Removal in priority order

op has completed execution

If successor’s operands are ready, put it on Ready
List Scheduling Example

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

S(n) =

The Dependence Graph
(longest latency-weighted)

The Generated Code

S(n) =

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

Lecture 8
**List Scheduling Example**

**The Code**

```
<table>
<thead>
<tr>
<th>Cycle</th>
<th>Ready - Set</th>
<th>Active - Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a: loadAI. r0, @w =&gt; r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c: loadAI r0, @x =&gt; r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e: loadAI r0, @y =&gt; r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b: add r1, r1 =&gt; r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d: mult r1, r2 =&gt; r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g: loadAI r0, @z =&gt; r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f: mult r1, r3 =&gt; r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h: mult r1, r2 =&gt; r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i: storeAI r1 =&gt; r0, @w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**The Generated Code**

```
S(n) =

0: a: loadAI. r0, @w => r1
1: c: loadAI r0, @x => r2
2: e: loadAI r0, @y => r3
3: b: add r1, r1 => r1
4: d: mult r1, r2 => r1
5: g: loadAI r0, @z => r2
6: f: mult r1, r3 => r1
7: h: mult r1, r2 => r1
8: i: storeAI r1 => r0, @w
```

**The Dependence Graph**

The dependence graph's longest latency-weighted cycle is 14.
**Scheduling Example**

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

### The Code

- **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

### The Dependence Graph

Note: Here we assume that an 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

The Code

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadAI</td>
<td>r0,@w</td>
<td>r1</td>
</tr>
<tr>
<td>add</td>
<td>r1,r1</td>
<td>r1</td>
</tr>
<tr>
<td>loadAI</td>
<td>r0,@x</td>
<td>r2</td>
</tr>
<tr>
<td>mult</td>
<td>r1,r2</td>
<td>r1</td>
</tr>
<tr>
<td>loadAI</td>
<td>r0,@y</td>
<td>r3</td>
</tr>
<tr>
<td>mult</td>
<td>r1,r3</td>
<td>r1</td>
</tr>
<tr>
<td>loadAI</td>
<td>r0,@z</td>
<td>r2</td>
</tr>
<tr>
<td>mult</td>
<td>r1,r2</td>
<td>r1</td>
</tr>
<tr>
<td>storeAI</td>
<td>r1</td>
<td>r0,@w</td>
</tr>
</tbody>
</table>

The Dependence Graph

Note: Here we assume that an 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).
Scheduling Example

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

The Code

We assume full latency for anti-dependences here
1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

**The Code**

| S(n): | 0 | a: loadAl r0,@w => r1 |
| 1 | c: loadAl r0,@x => r2 |
| 2 | e: loadAl r0,@y => r3 |
| 3 | b: add r1,r1 => r1 |
| 4 | d: mult r1,r2 => r1 |
| 6 | g: loadAl r0,@z => r2 |
| 7 | f: mult r1,r3 => r1 |
| 9 | h: mult r1,r2 => r1 |
| 11 | i: storeAl r1 => r0,@w |

**The Dependence Graph**

We assume full latency for anti-dependences here

cycles 14
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 (Note: we are not doing this)
- perform dynamic register renaming (⇒ may require spill code)
- move life ranges around (⇒ may remove or require spill code)
- ...

Lecture 8
The purpose of the front end is to deal with the input language

- Perform a membership test: code $\in$ source language?
- Is the program well-formed (semantically)?
- Build an IR version of the code for the rest of the compiler

_The front end is not monolithic_
The Front End

Source code \rightarrow \textbf{Scanner} \rightarrow \textit{tokens} \rightarrow \textbf{Parser} \rightarrow \textit{IR}

Scanner

- Maps stream of characters into words/tokens
  - Basic unit of syntax
  - $x = x + y$; becomes
    \[
    <\text{id},x><\text{eq},>=<\text{id},x><\text{pl},+><\text{id},y><\text{sc},;>
    \]
- Characters that form a word/token are its \textit{lexeme}
- Its \textit{part of speech} (or \textit{syntactic category}) is called its \textit{token type}
- Scanner discards white space & (often) comments

\textbf{Speed is an issue in scanning}  
\Rightarrow \text{use a specialized recognizer}
The Front End

Parser
- Checks stream of classified words (tokens) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax (static semantics)
- Builds an IR representation of the code

We’ll get to parsing in the next lectures
More Lexical Analysis

Read EaC: Chapters 2.1 - 2.5;