Code generation

Topics

- register allocation
  - place operands in registers
  - reduce load/store operations

- instruction scheduling
  - calculate when each instruction executes
  - enables pipelining to hide latency
  - required on VLIW architectures

- high-level languages
  - object oriented
  - functional

- code generation generators
  - automate backend construction
  - IR to code using specification
Code generation for trees

Sethi-Ullman algorithm

- generates 3-address code for expression trees
- uses minimal number of registers
- combines allocation and scheduling

Overview of algorithm

Phase 1

- compute number of registers required to evaluate a subtree without storing values to memory
- label each interior node with that number

Phase 2

- walk the tree and generate code
- evaluation order guided by labels
Sethi-Ullman Phase 1

if n is a leaf then
    label(n) ← 1
else begin /* n is an interior node */
    let $n_1, n_2, \ldots, n_k$ be the children of n, ordered so that
    \[
    \text{label}(n_1) \geq \text{label}(n_2) \geq \cdots \geq \text{label}(n_k)
    \]
    label(n) ← max$_{1 \leq i \leq k}$$(\text{label}(n_i) + i - 1)$

Can compute labels in postorder

For $n \leq 2$, label is defined recursively as:

\[
\text{label}(n) = \begin{cases} 
    l_1 + 1 & \text{if } l_1 = l_2 \\
    \max(l_1, l_2) & \text{if } l_1 \neq l_2
\end{cases}
\]

label = minReg (minimum # of registers)
Sethi-Ullman Phase 2

REG = current register number (initialized to 1)

procedure gencode(n)
  if n is leaf “name”
    /* case 0 — just load it */
    emit(load, REG, name);
  else if n is interior node “op n1 n2” then
    if label(n1) ≥ label(n2) then
      /* case 1 — generate left child first */
      gencode(n1); REG = REG+1;
      gencode(n2); REG = REG-1;
      emit(op, REG, REG, REG+1);
    else label(n1) < label(n2) then
      /* case 2 — generate right child first */
      gencode(n2); REG = REG+1;
      gencode(n1); REG = REG-1;
      emit(op, REG, REG+1, REG);
    endif
  endif
endif
Sethi-Ullman Example

\[
\begin{align*}
\text{gencode}(t_4) & \quad \text{case 1} \\
\text{gencode}(t_1) & \quad \text{case 1} \\
\text{gencode}(a) & \quad \text{case 0} \\
\text{load } r_1, a & \\
\text{gencode}(b) & \quad \text{case 0} \\
\text{load } r_2, b & \\
\text{add } r_1, r_1, r_2 & \\
\text{gencode}(t_3) & \quad \text{case 2} \\
\text{gencode}(t_2) & \quad \text{case 1} \\
\text{gencode}(c) & \quad \text{case 0} \\
\text{load } r_2, c & \\
\text{gencode}(d) & \quad \text{case 0} \\
\text{load } r_3, d & \\
\text{add } r_2, r_2, r_3 & \\
\text{gencode}(e) & \quad \text{case 0} \\
\text{load } r_3, e & \\
\text{sub } r_2, r_3, r_2 & \\
\text{sub } r_1, r_1, r_2 & \\
\end{align*}
\]
Improved code generation for trees

Delayed-load architectures

- issue \texttt{load}, result appears $k$ cycles later
- attempt to access target of load early causes hardware to stall (\textit{interlock})
- $k$ increases for modern microprocessors

Apply instruction scheduling

- move load back at least $k$ slots from \textit{op}
- to maintain legality, may need more registers

Naive approach

- issue all loads, then execute all operators
- will use too many registers

Phase ordering problem

- allocate registers first $\Rightarrow$ many stalls
- schedule instructions first $\Rightarrow$ many registers
Delayed load scheduling (DLS)

Approach

1. schedule the operations (à la Sethi-Ullman)
2. schedule the loads

Legal ordering

- children of an operator appear before it
- each load appears before operator that uses it

The final schedule

- preserves relative order of operations
  \( \text{ops} \leftrightarrow \text{ops} \)
- preserves relative order of loads
  \( \text{loads} \leftrightarrow \text{loads} \)
- changes relative order of loads to operations

The DLS algorithm

The canonical order
Given $R$ registers

1. schedule $R$ loads
2. schedule a series of $(op, load)$ pairs
3. schedule the remaining $R - 1$ ops

This keeps extra register pressure down

The algorithm

1. run Sethi-Ullman algorithm
   - calculate $minReg$ for each subtree
   - create an ordering of the operators

2. put loads into canonical order
   - uses $minReg + 1$ regs
   - requires some renaming
DLS example

Canonical ordering

<table>
<thead>
<tr>
<th>Operators</th>
<th>Loads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. sub</td>
<td>1. load m3</td>
</tr>
<tr>
<td>2. shift</td>
<td>2. load m4</td>
</tr>
<tr>
<td>3. add</td>
<td>3. load m5</td>
</tr>
<tr>
<td>4. div</td>
<td>4. load m6</td>
</tr>
<tr>
<td>5. mult</td>
<td>5. load m1</td>
</tr>
<tr>
<td></td>
<td>6. load m2</td>
</tr>
</tbody>
</table>
DLS example

<table>
<thead>
<tr>
<th></th>
<th>Sethi-Ullman</th>
<th>DLS(3 registers)</th>
<th>DLS(4 registers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load r1, m3</td>
<td>load r1, m3</td>
<td>load r1, m3</td>
</tr>
<tr>
<td>2</td>
<td>load r2, m4</td>
<td>load r2, m4</td>
<td>load r2, m4</td>
</tr>
<tr>
<td>3</td>
<td>-stall-</td>
<td>load r3, m5</td>
<td>load r3, m5</td>
</tr>
<tr>
<td>4</td>
<td>sub r1, r1, r2</td>
<td>sub r1, r1, r2</td>
<td>sub r1, r1, r2</td>
</tr>
<tr>
<td>5</td>
<td>load r2, m5</td>
<td>load r2, m6</td>
<td>load r4, m6</td>
</tr>
<tr>
<td>6</td>
<td>load r3, m6</td>
<td>-stall-</td>
<td>load r4, m6</td>
</tr>
<tr>
<td>7</td>
<td>-stall-</td>
<td>shift r2, r3, r2</td>
<td>load r4, m6</td>
</tr>
<tr>
<td>8</td>
<td>shift r2, r2, r3</td>
<td>load r3, m1</td>
<td>shift r3, r3, r4</td>
</tr>
<tr>
<td>9</td>
<td>add r1, r1, r2</td>
<td>add r1, r1, r2</td>
<td>add r1, r1, r3</td>
</tr>
<tr>
<td>10</td>
<td>load r2, m1</td>
<td>load r2, m2</td>
<td>div r2, r2, r4</td>
</tr>
<tr>
<td>11</td>
<td>load r3, m2</td>
<td>-stall-</td>
<td>mult r1, r2, r1</td>
</tr>
<tr>
<td>12</td>
<td>-stall-</td>
<td>div r2, r3, r2</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>div r2, r2, r3</td>
<td>mult r1, r2, r1</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>mult r1, r2, r1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Limitations

Input

- handles trees, not dags
- limited to a single basic block
- values not kept in registers

Output

- $delay > 1 \Rightarrow$ optimality not guaranteed
- non-constant $delay$ causes deeper problems

Strengths

- fast, simple algorithm
- clever metric for spilling
- no excuse to do worse

This work raises the bar for non-optimizing compilers