Reuse Optimization

Idea
- Eliminate redundant operations in the dynamic execution of instructions

How do redundancies arise?
- Loop invariant code (e.g., index calculation for arrays)
- Sequence of similar operations (e.g., method lookup)
- Lightning frequently strikes twice

Types of reuse optimization
- Value numbering
- Common subexpression elimination
- Partial redundancy elimination

Local Value Numbering

Idea
- Each variable, expression, and constant is assigned a unique number
- When we encounter a variable, expression or constant, see if it’s already been assigned a number
  - If so, use the value for that number
  - If not, assign a new number
- Same number ⇒ same value

Example

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>#2</td>
</tr>
<tr>
<td>b</td>
<td>#1</td>
</tr>
<tr>
<td>c</td>
<td>#2</td>
</tr>
<tr>
<td>b + c</td>
<td>#1 + #2 → #3</td>
</tr>
<tr>
<td>a</td>
<td>#3</td>
</tr>
<tr>
<td>d</td>
<td>#1</td>
</tr>
<tr>
<td>d + c</td>
<td>#1 + #2 → #3</td>
</tr>
<tr>
<td>e</td>
<td>#3</td>
</tr>
</tbody>
</table>

```java
a := b + c
b := a
d := b
e := d + c
```
Global Value Numbering

How do we handle control flow?

\[
\begin{align*}
  w &= 5 \\
  x &= 5 \\
  w &\rightarrow \#1 \\
  x &\rightarrow \#1 \\
  \ldots \\
  w &= 8 \\
  x &= 8 \\
  w &\rightarrow \#2 \\
  x &\rightarrow \#2 \\
  \ldots \\
  y &= w+1 \\
  z &= x+1
\end{align*}
\]

Global Value Numbering (cont)

Idea [Alpern, Wegman, and Zadeck 1988]
- Partition program variables into congruence classes
- All variables in a particular congruence class have the same value
- SSA form is helpful

Approaches to computing congruence classes
- Pessimistic
  - Assume no variables are congruent (start with \( n \) classes)
  - Iteratively coalesce classes that are determined to be congruent
- Optimistic
  - Assume all variables are congruent (start with one class)
  - Iteratively partition variables that contradict assumption
  - Slower but better results
Role of SSA Form

SSA form is helpful
- Allows us to avoid data-flow analysis
- Variables correspond to values

<table>
<thead>
<tr>
<th>a = b</th>
<th>a = c</th>
<th>a = d</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

a not congruent to anything

<table>
<thead>
<tr>
<th>a₁ = b</th>
<th>a₂ = c</th>
<th>a₃ = d</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

Congruence classes: {a₁,b}, {a₂,c}, {a₃,d}

Basis

Idea
- If x and y are congruent then f(x) and f(y) are congruent

<table>
<thead>
<tr>
<th>x and y are congruent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ta = a</td>
</tr>
<tr>
<td>tb = b</td>
</tr>
<tr>
<td>x = f(a,b)</td>
</tr>
<tr>
<td>y = f(ta,tb)</td>
</tr>
</tbody>
</table>

- Use this fact to combine (pessimistic) or split (optimistic) classes

Problem
- This is not true for φ-functions

<table>
<thead>
<tr>
<th>a₁ = x₁</th>
<th>a₂ = y₁</th>
<th>b₁ = x₁</th>
<th>b₂ = y₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₃ = φ₁(a₁, a₂)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b₃ = φ₁(b₁, b₂)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solution: Label φ-functions with join point

<table>
<thead>
<tr>
<th>a₁ &amp; b₁ congruent?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₂ &amp; b₂ congruent?</td>
</tr>
<tr>
<td>a₃ &amp; b₃ congruent?</td>
</tr>
</tbody>
</table>
Pessimistic Global Value Numbering

Idea
- Initially each variable is in its own congruence class
- Consider each assignment statement s (reverse postorder in CFG)
  - Update LHS value number with hash of RHS
  - Identical value number $\Rightarrow$ congruence

Why reverse postorder?
- Ensures that when we consider an assignment statement, we have already considered definitions that reach the RHS operands

Algorithm

for each assignment of the form: “$x = f(a, b)$”
ValNum$x$ $\leftarrow$ UniqueValue()

for each assignment of the form: “$x = f(a, b)$” (in reverse postorder)
ValNum$x$ $\leftarrow$ Hash($f \oplus$ ValNum$a$ $\oplus$ ValNum$b$)

\[
\begin{align*}
\text{i}_1 &= 1 \\
\text{w}_1 &= \text{b}_1 \\
\text{x}_1 &= \text{b}_1 \\
\text{w}_2 &= \text{a}_1 \\
\text{x}_2 &= \text{a}_1 \\
\text{w}_3 &= \phi_n(\text{w}_1, \text{w}_2) \\
\text{x}_3 &= \phi_n(\text{x}_1, \text{x}_2) \\
\text{y}_1 &= \text{w}_3 + \text{i}_1 \\
\text{z}_1 &= \text{x}_3 + \text{i}_1
\end{align*}
\]
Snag!

Problem
- Our algorithm assumes that we consider operands before variables that depend upon it
- Can’t deal with code containing loops!

Solution
- Ignore back edges
- Make conservative (worst case) assumption for previously unseen variable (i.e., assume its in it’s own congruence class)

Optimistic Global Value Numbering

Idea
- Initially all variables in one congruence class
- Split congruence classes when evidence of non-congruence arises
  - Variables that are computed using different functions
  - Variables that are computed using functions with non-congruent operands
### Splitting

**Initially**
- Variables computed using the same function are placed in the same class

\[
x_1 = f(a_1, b_1) \\
\vdots \\
y_1 = f(c_1, d_1) \\
\vdots \\
z_1 = f(e_1, f_1)
\]

<table>
<thead>
<tr>
<th>P</th>
<th>P'</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_1</td>
<td>y_1</td>
</tr>
<tr>
<td>z_1</td>
<td></td>
</tr>
</tbody>
</table>

**Iteratively**
- Split classes when corresponding operands are in different classes
- Example: \(a_1\) and \(c_1\) are congruent, but \(e_1\) is congruent to neither

\[
x_1 = f(a_1, b_1) \\
\vdots \\
y_1 = f(c_1, d_1) \\
\vdots \\
z_1 = f(e_1, f_1)
\]

### Splitting (cont)

**Definitions**
- Suppose \(P\) and \(Q\) are sets representing congruence classes
- \(Q\) splits \(P\) for each \(i\) into two sets
  - \(P \setminus Q\) contains variables in \(P\) whose \(i^{th}\) operand is in \(Q\)
  - \(P / Q\) contains variables in \(P\) whose \(i^{th}\) operand is not in \(Q\)
- \(Q\) properly splits \(P\) if neither resulting set is empty
Algorithm

worklist ← ∅
for each function f
    \( C_f \) ← ∅
    for each assignment of the form “\( x = f(a,b) \)”
        \( C_f \) ← \( C_f \) ∪ \{ x \}
    worklist ← worklist ∪ \{ C_f \}
CC ← CC ∪ \{ C_f \}
while worklist ≠ ∅
    Delete some D from worklist
    for each class C properly split by D (at operand i)
        CC ← CC – C
        worklist ← worklist – C
        Create new congruence classes \( C_j \) ← \{ C \_i \_D \} and \( C_k \) ← \{ C / i \_D \}
        CC ← CC ∪ \( C_j \) ∪ \( C_k \)
        worklist ← worklist ∪ \( C_j \) ∪ \( C_k \)

Note: see paper for optimization

Example

<table>
<thead>
<tr>
<th>SSA code</th>
<th>Congruence classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_0 = 1 )</td>
<td>( S_0 = { x_0 } )</td>
</tr>
<tr>
<td>( y_0 = 2 )</td>
<td>( S_1 = { y_0 } )</td>
</tr>
<tr>
<td>( x_1 = x_0 + 1 )</td>
<td>( S_2 = { x_1, y_0, z_0, z_1 } )</td>
</tr>
<tr>
<td>( y_1 = y_0 + 1 )</td>
<td>( S_3 = { x_1, z_1 } )</td>
</tr>
<tr>
<td>( z_1 = x_0 + 1 )</td>
<td>( S_4 = { y_1 } )</td>
</tr>
</tbody>
</table>

Worklist: \( S_0 = \{ x_0 \} \), \( S_1 = \{ y_0 \} \), \( S_2 = \{ x_1, y_0, z_0, z_1 \} \), \( S_3 = \{ x_1, z_1 \} \), \( S_4 = \{ y_1 \} \)

\( S_0 \) psplit \( S_0 \)? no \( S_0 \) psplit \( S_1 \)? no \( S_0 \) psplit \( S_2 \)? yes!

\( S_2 \_1 \) \( S_0 = \{ x_1, z_1 \} = S_3 \)
\( S_2 \_1 \) \( S_0 = \{ y_1 \} = S_4 \)
Comparing Optimistic and Pessimistic

Differences
- Handling of loops
- Pessimistic makes worst-case assumptions on back edges
- Optimistic requires actual contradiction to split classes

Role of SSA

Single global result
- Single def reaches each use
- No data (flow value) at each point

No data flow analysis
- Optimistic: Iterate over congruence classes, not CFG nodes
- Pessimistic: Visit each assignment once

φ-functions
- Make data-flow merging explicit
- Treat like normal functions
Next Time

Lecture
- Common-Subexpression Elimination

Study Suggestion
- Read Alpern and Zadeck 1992 Chapter about Value Numbering
- Do the examples in Muchnick Section 12.4 and examples in paper using the others algorithm