Reuse Optimization

Last time
- Dead code elimination
- Common subexpression elimination (CSE)
- Copy propagation
- Simple constants

Today
- Partial redundancy elimination (PRE)

Partial Redundancy Elimination (PRE)

Partial Redundancy
- An expression (e.g., \(x+y\)) is partially redundant at node \(n\) if some path from the entry node to \(n\) evaluates \(x+y\), and there are no definitions of \(x\) or \(y\) between the last evaluation of \(x+y\) and \(n\).

Elimination
- Discover partially redundant expressions
- Convert them to fully redundant expressions
- Remove redundancy

PRE subsumes CSE and loop invariant code motion

Loop Invariance Example

PRE removes loop invariants
- An invariant expression is partially redundant
- PRE converts this partial redundancy to full redundancy
- PRE removes the redundancy

Example

1. \(a := b + c\)
2. \(x := y * z\)
3. \(a := b + c\)
4. \(x := y * z\)

Implementing PRE [lazy code motion Knoop 92, dragon 2007]

Big picture
- Use global analysis (data-flow analysis) to discover where partial redundancy can be converted to full redundancy
- Global analysis also determines latest possible point to create redundancy
- Insert code and remove redundant expressions
- As in textbook, assuming one statement per basic block

Example

1. expr
2. insert computation
3. delete computation
**Local Properties**

An expression is locally **available** (or in \( \text{e\_gen}(b) \)) in block \( b \) if it is computed at least once and its operands are not modified after its last computation in \( b \).

An expression is locally **anticipated** if it is computed at least once and its operands are not modified before its first evaluation.

An expression is locally **used** (or in \( \text{e\_use}(b) \)) in block \( b \) if it is computed at least once. With only one statement per block, \( \text{anticipated} = \text{e\_use}(b) \).

An expression is locally **killed** (or in \( \text{e\_kill}(b) \)) in block \( b \) if any of its operands are defined in \( b \).

**Example**

\[
\mathbf{b} := \mathbf{b} + \mathbf{c}
\]

Available: \( \{ \} \)

Anticipated: \( \{ \mathbf{b} + \mathbf{c} \} \)

\( \text{e\_use} \quad \{ \mathbf{b} + \mathbf{c} \} \)

\( \text{e\_kill} \quad \{ \mathbf{b} + \mathbf{c}, \mathbf{b\_a}, \ldots \} \)

---

**Global Anticipability**

**Intuition**
- If \( e \) is globally anticipated at \( p \), then an evaluation of \( e \) at \( p \) will make the next evaluation of \( e \) redundant along all paths from \( p \).

**Flow Functions**
- \( \text{anticipated}_{\text{out}}[n] = \bigcap_{s \in \text{succ}[n]} \text{anticipated}_{\text{in}}[s] \)
- \( \text{anticipated}_{\text{in}}[n] = \text{e\_use}[n] \cup (\text{anticipated}_{\text{out}}[n] - \text{e\_kill}[n]) \)

---

**Global Availability for PRE**

**Intuition**
- Global availability for PRE is almost the same as Available Expressions, except it depends on the results of global anticipated expressions.

- If \( e \) is globally available at \( p \), then an evaluation at \( p \) will create redundancy along all paths start at \( p \).

**Flow Functions**
- \( \text{available}_{\text{in}}[n] = \bigcap_{s \in \text{pred}[n]} \text{available}_{\text{out}}[p] \)
- \( \text{available}_{\text{out}}[n] = (\text{anticipated}_{\text{in}}[n] - \text{e\_kill}[n]) \cup (\text{available}_{\text{in}}[n] - \text{e\_kill}[n]) \)

---

**Earliest**

**Intuition**
- The earliest place an expression is anticipated, but not globally available.
- Does not require iterative data-flow analysis. Just requires one pass over all statements.
- Could place an expression generation statement at the beginning of any block \( b \) where expression is in \( \text{earliest}[b] \)

**Function**
- \( \text{earliest}[n] = \text{anticipated}_{\text{in}}[n] - \text{available}_{\text{in}}[n] \)
Example

B1: \( a := b + c \)  
B2: \( x := b + 1 \)  
B3: \( a := b + c \)  

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>used</td>
<td>([b+c])</td>
<td>([b+1])</td>
<td>([b+c])</td>
</tr>
<tr>
<td>kill</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>available_in</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>available_out</td>
<td></td>
<td>([b+c])</td>
<td></td>
</tr>
<tr>
<td>earliest</td>
<td>([b+c])</td>
<td>([b+1])</td>
<td></td>
</tr>
<tr>
<td>postponable_in</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>postponable_out</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>latest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>used_out</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>used_in</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Postponable Expressions

Intuition

– Calculates all possible program points starting from earliest in which an expression can be computed while preserving semantics and not introducing more redundancy.

– By computing an expression at the latest possible postponable point, the live range of the temporary that holds expression value is reduced.

Flow Functions

\[
\text{postponable_in}[n] = \bigcap_{s \in \text{pred}[n]} \text{postponable_out}[s]
\]

\[
\text{postponable_out}[n] = (\text{earliest}[n] - \text{e_use}[n]) \cup (\text{postponable_in}[n] - \text{e_use}[n])
\]

Latest

Intuition

– Last block the computation for which the expression computation can be postponed.

– If an expression is in latest[b] that indicates that the last point the expression can be computed is at the beginning of block b.

– Requires only one visit to each statement.

Flow Function

\[
\text{latest}[n] = (\text{earliest}[n] \cup \text{postponable_in}[n]) \cap (\text{e_use}[n] \cup \neg (\bigcap_{s \in \text{succ}[n]} (\text{earliest}[s] \cup \text{postponable_in}[s])))
\]

Example

B1: \( a := b + c \)  
B2: \( x := b + 1 \)  
B3: \( a := b + c \)  

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>use</td>
<td>([b+c])</td>
<td>([b+1])</td>
<td>([b+c])</td>
</tr>
<tr>
<td>kill</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>anticipated_out</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>anticipated_in</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>available_out</td>
<td></td>
<td>([b+c])</td>
<td></td>
</tr>
<tr>
<td>earliest</td>
<td>([b+c])</td>
<td>([b+1])</td>
<td></td>
</tr>
<tr>
<td>postponable_in</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>postponable_out</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>latest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>used_out</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>used_in</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Global Used Expressions

Intuition
- If e is globally used at p, then an evaluation of e at p will be used again along some path starting at p.
- If an expression is not in the used_out[b], then a computation of the expression should not be put at the beginning of block b, even if e is in latest[b].
- “Liveness analysis for expressions.”

Flow Functions

\[
\begin{align*}
\text{used}_{\text{out}}[n] &= \bigcup_{s \in \text{succ}[n]} \text{used}_{\text{in}}[s] \\
\text{used}_{\text{in}}[n] &= \left( \text{e}_{\text{use}}[n] - \text{latest}[n] \right) \bigcup \left( \text{used}_{\text{out}}[n] - \text{latest}[n] \right)
\end{align*}
\]

PRE Summary

Algorithm
- Insert an empty block along all edges entering a block with more than one predecessor.
- Calculate latest and used_out sets.
- For each expression e
  - create a temporary t to store e
  - for all blocks where e is in latest[b] and used_out[b], add t=e to beginning of block
- for all blocks where e is in (e_use[b] and (not latest[b] or used_out[b])), replace original e with t

What’s so great about PRE?
- A modern optimization that subsumes earlier ideas
- Composes several simple data-flow analyses to produce a powerful result
- Finds earliest and latest points in the CFG at which an expression is anticipated

Another Example

\[
\begin{align*}
\text{B1: } & a := b + c \\
\text{B2: } & b := b + 1 \\
\text{B3: } & a := b + c
\end{align*}
\]
Next Time

Assignments
- HW1 has been posted

Lecture
- Control flow
- Loops
- Dominators