Lecture 5

Partial Redundancy Elimination

I  Forms of redundancy
   -- global common subexpression elimination
   -- loop invariant code motion
   -- partial redundancy

II Lazy Code Motion Algorithm

Reading: Chapter 9.5
Overview

- Eliminates many forms of redundancy in one fell swoop
- Originally formulated as 1 bi-directional analysis
- Lazy code motion algorithm
  - formulated as 4 separate uni-directional passes
    (backward, forward, forward, backward)
I. Common Subexpression Elimination

- A common expression may have different values on different paths!
- On every path reaching \( p \),
  - expression \( b+c \) has been computed
  - \( b, c \) not overwritten after the expression
Loop Invariant Code Motion

- Given an expression (b+c) inside a loop, does the value of b+c change inside the loop? is the code executed at least once?
Partial Redundancy

- Can we place calculations of \( b+c \) such that no path re-executes the same expression

- Partial redundancy elimination (PRE)
  - subsumes:
    - global common subexpression (full redundancy)
    - loop invariant code motion (partial redundancy for loops)
II. Increasing the Chance of Optimization

- Critical edges
  - source basic block has multiple successors
  - destination basic block has multiple predecessors
- Assume every statement is a basic block
  - Only place statements at the beginning of a basic block
  - Add a basic block for every edge that leads to a basic block with multiple predecessors
Full Redundancy

- Full redundancy at p: expression a+b redundant on all paths
  - cutset: nodes that separate entry from p
  - cutset contains calculation of a+b
  - a, b, not redefined
Partial Redundancy

- Partial redundancy at p: redundant on some but not all paths
  - Add operations to create a cutset containing a+b
  - Note: Moving operations up can eliminate redundancy
- Constraint on placement: no wasted operation
  - a+b is “anticipated” at B if its value computed at B will be used along ALL subsequent paths
  - a, b not redefined, no branches that lead to exit without use
- Range where a+b is anticipated --> Choice
**Pass 1: Anticipated Expressions**

- **Backward pass: Anticipated expressions**
  - Anticipated\[b\].\text{in}: Set of expressions anticipated at the entry of \( b \)
  - An expression is anticipated if its value computed at point \( p \) will be used along ALL subsequent paths

<table>
<thead>
<tr>
<th>Domain</th>
<th>Anticipated Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>backward</td>
</tr>
<tr>
<td>Transfer function</td>
<td></td>
</tr>
<tr>
<td>( f_b(x) = \text{EUse}_b \cup (x - \text{EKil}_b) )</td>
<td></td>
</tr>
<tr>
<td>( \text{EUse}: \text{used exp} )</td>
<td></td>
</tr>
<tr>
<td>( \text{EKil}: \text{exp killed} )</td>
<td></td>
</tr>
<tr>
<td>Boundary</td>
<td>( \text{in[exit]} = \emptyset )</td>
</tr>
<tr>
<td>Initialization</td>
<td>( \text{in}[b] = { \text{all expressions} } )</td>
</tr>
</tbody>
</table>
Examples (1)

\[
x = a + b \\
y = a + b \\
z = a + b \\
a = 10 \\
r = a + b
\]
Examples (2)

a + b

Diagram:

```
  a + b
   / \
  /   \n/     /
```

```
  a + b
```

```
Examples (3)

\[
x = a + b \\
y = a + b \\
a = 10
\]
Pass 2: Place As Early As Possible

- First approximation: frontier between “not anticipated” & “anticipated”
- Complication: Anticipation may oscillate

\[
x = a + b
\]

- Assume: place expression e such that it is available where it is anticipated.
- e will be available at p if e has been anticipated but not subsequently killed on all paths reaching p

<table>
<thead>
<tr>
<th>Domain</th>
<th>Available Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Sets of expressions</td>
</tr>
<tr>
<td>Transfer function</td>
<td>( f_b(x) = (\text{Anticipated}[b].\text{in} \cup x) - \text{EKill}_b )</td>
</tr>
<tr>
<td>Boundary condition</td>
<td>out[entry] = ( \emptyset )</td>
</tr>
<tr>
<td>Initialization</td>
<td>out[b] = {all expressions}</td>
</tr>
</tbody>
</table>
Early Placement

- **earliest(b)**
  - set of expressions added to block b under early placement

- **Place expression at the earliest point anticipated and not already available**
  - earliest(b) = anticipated[b].in – available[b].in

- **Algorithm**
  - For all basic block b, if x+y \in earliest[b]
    - at beginning of b:
      - create a new variable t
      - t = x+y,
      - replace every original x+y by t
Pass 3: Lazy Code Motion

- Delay without creating redundancy to reduce register pressure

- An expression \( e \) is postponable at a program point \( p \) if
  - all paths leading to \( p \) have seen the earliest placement of \( e \) but not a subsequent use

### Postponable Expressions

<table>
<thead>
<tr>
<th>Domain</th>
<th>Sets of expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>forward</td>
</tr>
<tr>
<td>Transfer function</td>
<td>( f_b(x) = (\text{earliest}[b] \cup x) \setminus \text{EUse}_b )</td>
</tr>
<tr>
<td>( \land )</td>
<td>( \cap )</td>
</tr>
<tr>
<td>Boundary condition</td>
<td>( \text{out}[\text{entry}] = \emptyset )</td>
</tr>
<tr>
<td>Initialization</td>
<td>( \text{out}[b] = {\text{all expressions}} )</td>
</tr>
</tbody>
</table>
Latest: frontier at the end of “postponable” cut set

- latest[b] = (earliest[b] ∪ postponable.in[b]) ∩
  (EUse_b ∪ ¬(∩s ∈ succ[b](earliest[s] ∪ postponable.in[s])))

- OK to place expression: earliest or postponable

- Need to place at b if either
  - used in b, or
  - not OK to place in one of its successors

- Note because of pre-processing step:
  - if one of its successors cannot accept postponement, b has only one successor
  - The following does not exist

```
 OK to place

OK to place

OK to place

not OK to place
```
Pass 4: Cleaning Up

• Eliminate temporary variable assignments unused beyond current block

• Compute: Used.out[b]: sets of used (live) expressions at exit of b.

<table>
<thead>
<tr>
<th></th>
<th>Used Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>Sets of expressions</td>
</tr>
<tr>
<td>Direction</td>
<td>backward</td>
</tr>
<tr>
<td>Transfer function</td>
<td>$f_b(x) = (EUse[b] \cup x) -latest[b]$</td>
</tr>
<tr>
<td>$\wedge$</td>
<td>$\cup$</td>
</tr>
<tr>
<td>Boundary condition</td>
<td>$\text{in[exit]} = \emptyset$</td>
</tr>
<tr>
<td>Initialization</td>
<td>$\text{in[b]} = \emptyset$</td>
</tr>
</tbody>
</table>
• For all basic blocks $b$,
  if $(x+y) \in (\text{latest}[b] \cap \text{used.out}[b])$
  
  at beginning of $b$:
  add new $t = x+y$

  if $(x+y) \in (\text{EUse}_b \cap \neg(\text{latest}[b] \cap \neg \text{used.out}[b]))$
  replace every original $x+y$ by $t$
Summary

- Cannot execute any operations not executed originally
  - Pass 1: Anticipation: range of code motion

- Eliminate as many redundant calculations of an expression as possible, without duplicating code
  - Pass 2: Availability: move it up as early as possible

- Delay computation as much as possible to minimize register lifetimes
  - Pass 3: Postponable: move it down unless it creates redundancy (lazy code motion)

- Pass 4: Remove temporary assignment
Remarks

• Powerful algorithm
  • Finds many forms of redundancy in one unified framework

• Illustrates the power of data flow
  • Multiple data flow problems