Lecture 11
Lazy Code Motion

I. Forms of redundancy (quick review)
• global common subexpression elimination
• loop invariant code motion
• partial redundancy

II. Lazy Code Motion Algorithm
• Mathematical concept: a cut set
• Basic technique (anticipation)
• 3 more passes to refine 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 = b + c
  d = b + c
```
• 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
```
  a = b + c
  t = b + c
  a = t
```
• 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. Lazy Code Motion

- Key observation:
  - A bi-directional (!) data flow problem can be replaced with several unidirectional data flow problems \(\rightarrow\) much easier
  - Better result as well

Preparing the Flow Graph

- Definition: Critical edges
  - source basic block has multiple successors
  - destination basic block has multiple predecessors

- Modify the flow graph: (treat every statement as a basic block)
  - To keep algorithm simple: restrict placement of instructions to the beginning of a basic block
  - Add a basic block for every edge that leads to a basic block with multiple predecessors (not just on critical edges)

Full Redundancy: A Cut Set in a Graph

- Full redundancy at \(p\): expression \(a+b\) redundant on all paths
  - a cut set: nodes that separate entry from \(p\)
  - a cut set contains calculation of \(a+b\)
  - \(a, b\), not redefined
Partial Redundancy: Completing a Cut Set

- Partial redundancy at $p$: redundant on some but not all paths
  - Add operations to create a cut set 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 $\rightarrow$ Choice

Pass 1: Anticipated Expressions

This pass does most of the heavy lifting in eliminating redundancy

- Backward pass: Anticipated expressions
  Anticipated[$b$.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
  - First approximation:
    - place operations at the frontier of anticipation (boundary between not anticipated and anticipated)

Examples (1)

- See the algorithm in action

Examples (2)

- Cannot eliminate all redundancy
Examples (3)

Do you know how the algorithm works without simulating it?

Pass 2: Place As Early As Possible

First approximation: frontier between “not anticipated” & “anticipated”

Complication: anticipation may oscillate

Pretend we calculate expression e whenever 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>Available Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain: Sets of expressions</td>
</tr>
<tr>
<td>Direction: forward</td>
</tr>
<tr>
<td>Transfer Function: $f_{b}(x) = (\text{anticipated}[b].\text{in} \cup x) - \text{EKill}[b]$</td>
</tr>
<tr>
<td>Boundary: out(entry) = $\emptyset$</td>
</tr>
<tr>
<td>Initialization: out(b) = (all expressions)</td>
</tr>
</tbody>
</table>

Pass 3: Lazy Code Motion

Let’s be lazy without introducing redundancy.

Delay 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

<table>
<thead>
<tr>
<th>Postponable Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain: Sets of expressions</td>
</tr>
<tr>
<td>Direction: forward</td>
</tr>
<tr>
<td>Transfer Function: $f_{b}(x) = (\text{earliest}[b].\text{in} \cup x) - \text{EUse}[b]$</td>
</tr>
<tr>
<td>Boundary: out(entry) = $\emptyset$</td>
</tr>
<tr>
<td>Initialization: out(b) = (all expressions)</td>
</tr>
</tbody>
</table>
Latest: frontier at the end of “postponable” cut set

- \( \text{latest}[b] = (\text{earliest}[b] \cup \text{postponable.in}[b]) \cap (\text{EUse}_b \cup \neg \bigwedge \forall s \in \text{succ}[b] (\text{earliest}[s] \cup \text{postponable.in}[s]))) \)
  - 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
- Works because of pre-processing step (an empty block was introduced to an edge if the destination has multiple predecessors)
  - if \( b \) has a successor that cannot accept postponement, \( b \) has only one successor
  - The following does not exist:

Code Transformation

- For all basic blocks \( b \),
  - if \( (x+y) \in (\text{latest}[b] \cap \text{used.out}[b]) \)
  - at beginning of \( b \):
    - add new \( t = x+y \)
    - replace every original \( x+y \) by \( t \)

Pass 4: Cleaning Up

Finally, this is easy, it is the cleanup

- Eliminate temporary variable assignments unused beyond current block
- Compute: \( \text{Used.out}[b] \): sets of used (live) expressions at exit of \( b \)

<table>
<thead>
<tr>
<th>Used Expressions</th>
<th>Domain</th>
<th>Sets of expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Direction</td>
<td>backward</td>
</tr>
<tr>
<td></td>
<td>Transfer Function</td>
<td>( f_b(x) = (\text{EUse}_b \cup x) \cap \text{latest}[b] )</td>
</tr>
<tr>
<td>Boundary</td>
<td>( \text{in[exit]} = \emptyset )</td>
<td></td>
</tr>
<tr>
<td>Initialization</td>
<td>( \text{in}[b] = \emptyset )</td>
<td></td>
</tr>
</tbody>
</table>

4 Passes for Partial Redundancy Elimination

- Heavy lifting: Cannot introduce operations not executed originally
  - Pass 1 (backward): Anticipation: range of code motion
  - Placing operations at the frontier of anticipation gets most of the redundancy
- Squeezing the last drop of redundancy:
  - An anticipation frontier may cover a subsequent frontier
    - Pass 2 (forward): Availability
      - Earliest: anticipated, but not yet available
    - Push the cut set out -- as late as possible
      To minimize register lifetimes
      - Pass 3 (forward): Postponability: move it down provided it does not create redundancy
      - Latest: where it is used or the frontier of postponability
- Cleaning up
  - Pass 4: Remove temporary assignments
Remarks

• Powerful algorithm
  – Finds many forms of redundancy in one unified framework
• Illustrates the power of data flow
  – Multiple data flow problems