Lecture 5

Partial Redundancy Elimination

I. Forms of redundancy
   • 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
• Shows off the power and elegance of data flow
• Plan
  – Simple examples to build up intuition
  – Introduce mathematical concept: cut sets
  – Key: understand what the algorithm does without simulation
  – Details of the algorithm
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)

  Unifying theory: More powerful, elegant \( \Rightarrow \) but less direct.

II. Preparing the Flow Graph

- Key observation
  - Can replace a bi-directional (!) data flow with several unidirectional data flows \( \Rightarrow \) much easier
  - Better result as well!

- 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

Key mathematical concept

- 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 with out use

- Range where a+b is anticipated → 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

<table>
<thead>
<tr>
<th>Anticipated Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
</tr>
<tr>
<td>Direction</td>
</tr>
<tr>
<td>Transfer Function</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Boundary</td>
</tr>
<tr>
<td>Initialization</td>
</tr>
</tbody>
</table>

- **First approximation**:
  - place operations at the frontier of anticipation (boundary between not anticipated and anticipated)

---

Examples (1)

See the algorithm in action

```
x = a + b
z = a + b
y = a + b
x = a + b
r = a + b
a = 10
```
Examples (2)

\[ x = a + b \]

\[ z = a + b \]

- Cannot eliminate all redundancy

Examples (3)

Do you know how the algorithm works without simulating it?

\[ x = a + b \]

\[ y = a + b \]

\[ a = 10 \]

\[ x = a + b \]

\[ y = a + b \]

\[ a = 10 \]
Pass 2: Place As Early As Possible

- First approximation: frontier between "not anticipated" & "anticipated"
- Complication: Anticipation may oscillate
- An anticipation frontier may cover a subsequent frontier.
- Once an expression has been anticipated, it is "available" to subsequent frontiers → no need to re-evaluate.
- e will be available at p if e has been "anticipated but not subsequently killed" on all paths reaching p

Available Expressions

- 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</td>
</tr>
<tr>
<td>Direction</td>
</tr>
<tr>
<td>Transfer Function</td>
</tr>
<tr>
<td>^</td>
</tr>
<tr>
<td>Boundary</td>
</tr>
<tr>
<td>Initialization</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 \( x+y \in \text{earliest}[b] \) at beginning of b:
      - let \( t \) be the unique variable representing \( x+y \)
      - add \( t = x+y \),
      - replace every original \( x+y \) in the program by \( t \)

Pass 3: Lazy Code Motion

Let’s be lazy without introducing redundancy.

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

<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 ( f_b(x) = (\text{earliest}[\text{b}] \cup x) \cdot \text{EUse}_b )</td>
<td></td>
</tr>
<tr>
<td>Boundary</td>
<td>( \text{out}[\text{entry}] = \emptyset )</td>
</tr>
<tr>
<td>Initialization</td>
<td>( \text{out}[\text{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
- 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:

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>Used Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
</tr>
<tr>
<td>Direction</td>
</tr>
<tr>
<td>Transfer Function f_b(x)</td>
</tr>
<tr>
<td>Boundary in[exit]</td>
</tr>
<tr>
<td>Initialization in[b]</td>
</tr>
</tbody>
</table>
Code Transformation

Original version: For each basic block \( b \),
if \( x+y \in \text{earliest}[b] \)
at beginning of \( b \):
    let \( t \) be the unique variable representing \( x+y \)
    add \( t = x+y \),
replace every original \( x+y \) in the program by \( t \)

New version: For each basic block \( b \),
if \( (x+y) \in (\text{latest}[b] \cap \neg \text{used.out}[b]) \) { }
else
    if \( x+y \in \text{latest}[b] \)
at beginning of \( b \):
    let \( t \) be the unique variable representing \( x+y \)
    add \( t = x+y \),
replace every original \( x+y \) in the program by \( t \)

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 assignment**
Remarks

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

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