Lecture 5

Partial Redundancy Elimination

I. Redundancy Optimizations
   • Global common subexpression elimination
   • Loop invariant code motion
   • Partial redundancy elimination

II. Lazy Code Motion Algorithm
   • Mathematical concept: a cut set
   • Basic technique (anticipation)
   • 3 more passes to refine algorithm

Reading: Chapter 9.5

Jens Knoop, Oliver Rüthing, Berhard Steffen, Lazy Code Motion, PLDI 1992. (Most Influential PLDI Paper Award, 2002)
Overview

- **Redundancy optimizations**
  - Global common subexpression elimination
  - Loop invariant code motion
  - Partial redundancy elimination (subsumes the above)
    - Originally formulated as 1 bi-directional analysis!

- **Partial redundancy: Lazy code motion algorithm**
  - Formulated as 4 separate uni-directional passes
    - backward, forward, forward, backward
  - Easier to analyze and better performance than 1 bi-directional analysis

- **Shows off the power and elegance of data flow**
  - Sees program as a graph
  - Uses the math concept of a cut set, used later in instruction scheduling
Outline of This Lecture

• **Overview**
  – Simple examples to build up your intuition
  – Introduce cut sets
  – Key: understand what the algorithm does without simulation
  – Details of the algorithm

• **Simple but hard**: Please work out examples after class immediately
I. Common Subexpression Elimination

Build up intuition about redundancy elimination with examples of familiar concepts

- 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

\[
a = b + c
\]

\[
d = b + c
\]

\[
a = b + c
\]

\[
\]

\[
b = 7
\]

\[
da = b + c
\]

\[
d = b + c
\]

\[
b = 7
\]

\[
a = b + c
\]

\[
d = b + c
\]

\[
\]

\[
\]
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?

Observations:
- Important optimization. Why?
- Unlike common subexpression elimination, need to place an instruction in a new program point! Where?
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

- Original PRE: a bi-direction data flow problem! Hard to understand
- **Problem: Critical edges**
  - Source basic block has multiple successors
  - Destination basic block has multiple predecessors

- **Lazy Code Motion (Knoop 92)** – most influential paper award
  - Suggest changing the graph: getting rid of critical edges
  - Increase opportunities for optimization
  - Replace the bi-directional data flow problem into 4 unidirectional data flows – simpler to understand.
II. 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

A cut set of node $p$: a set of nodes that separate the start node from $p$

Full redundancy at $p$: expression $a+b$ redundant on all paths
  - there exists a cut set containing calculations of $a+b$
  - and $a$, $b$, are 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 \( a+b \)
- **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 before use, no branches that lead to exit without using \( a+b \)
- **Range where \( a+b \) is anticipated → Choice**
- **Greedy**: Place operations at the earliest frontier to minimize redundancy
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>Domain</th>
<th>Anticipated Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets of expressions</td>
<td>backward</td>
</tr>
<tr>
<td>Transfer Function</td>
<td>$f_b(x) = \text{EUse}_b \cup (x \cdot \text{EKill}_b)$</td>
</tr>
<tr>
<td></td>
<td>EUse: used exp, EKill: exp killed</td>
</tr>
<tr>
<td>Boundary</td>
<td>in[exit] = $\emptyset$</td>
</tr>
<tr>
<td>Initialization</td>
<td>in[b] = {all expressions}</td>
</tr>
</tbody>
</table>

- **First approximation:**
  - place operations at the earliest frontier of anticipation (boundary from “not anticipated” to “anticipated”)

  e.g. $c = a + b$
  - EUse: $a + b$
  - EKill: all exp using $c$
Examples (1)

See the algorithm in action

(Added basic blocks assumed; not shown for simplicity)
Examples (2)

Can Partial Redundancy Elimination eliminate all partial redundancy?

\[
x = a + b
\]
\[
z = a + b
\]
Code Duplication

• Code duplication can reduce redundancy

• Cost: larger code size
  → can be costly if it causes more misses in the instruction cache

• Code duplication not used for partial redundancy elimination
  – Used in instruction scheduling for parallelism
Examples (3)

Do you know how the algorithm works without simulating it?
Pass 2: Place As Early As Possible

There is still some redundancy left!

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

\[ \text{earliest}(b) = \text{anticipated}[b].in \text{- available}[b].in \]

- An anticipation frontier may cover a subsequent frontier.
  - Once an expression has been anticipated (and assumed evaluated) 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 \( p \)
- **Place expression at the earliest point anticipated and not already available**
  - \( \text{earliest}(b) = \text{anticipated}[b].in - \text{available}[b].in \)
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>Domain</th>
<th>Sets of expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>forward</td>
</tr>
<tr>
<td>Transfer Function</td>
<td>$f_b(x) = (\text{Anticipated}[b].\text{in} \cup x) - \text{EKill}_b$</td>
</tr>
<tr>
<td>Boundary</td>
<td>$\text{out}[\text{entry}] = \emptyset$</td>
</tr>
<tr>
<td>Initialization</td>
<td>$\text{out}[b] = {\text{all expressions}}$</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 ∈ 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</td>
<td>$f_b(x) = (\text{earliest}[b] \cup x) - \text{EUse}_b$</td>
</tr>
<tr>
<td>$\wedge$</td>
<td>$\cap$</td>
</tr>
<tr>
<td>Boundary</td>
<td>out[entry] = $\emptyset$</td>
</tr>
<tr>
<td>Initialization</td>
<td>out[b] = {all expressions}</td>
</tr>
</tbody>
</table>
Latest: Latest Frontier 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.in
- Need to place at b if either
  - used in b, or
  - not OK to place in one of its successors
- Note: If postponable.out[b] and ¬ postponable.in[s], b is empty and there is only one successor s
Pass 4: Cleaning Up

Finally... this is easy, it is like liveness

• Eliminate temporary variable assignments unused beyond the current block
• Compute: Used.out[b]: sets of used (live) expressions at exit of b.

Used Expressions

<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>$f_b(x) = (EUse[b] \cup x) - \text{latest}[b]$</td>
</tr>
<tr>
<td>$\wedge$</td>
<td>$\cup$</td>
</tr>
<tr>
<td>Boundary</td>
<td>$\text{in[exit]} = \emptyset$</td>
</tr>
<tr>
<td>Initialization</td>
<td>$\text{in}[b] = \emptyset$</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

- **Squeeze 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

- **Clean up**: Remove unused assignment
  - Pass 4: **Used**: if not used, don’t do anything
Remarks

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

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