Lecture 10:
Lazy Code Motion

I. Mathematical concept: a cut set

II. Lazy Code Motion Algorithm
   • Pass 1: Anticipated Expressions
   • Pass 2: (Will be) Available Expressions
   • Pass 3: Postponable Expressions
   • Pass 4: Used Expressions
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?

Can \(b + c\) be moved to header?

\[
\begin{align*}
a &= b + c \\
b &= \text{read()} \\
a &= b + c \\
\end{align*}
\]
Review: Partial Redundancy Elimination

• 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)
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 (there can be many cut sets)
  - 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** → **Choice**
Review: Where Can We Insert Computations?

- **Safety**: never introduce a new expression along any path.
  - Insertion could introduce exception, change program behavior.
  - Solution: insert expression only where it is **anticipated**, i.e., its value computed at point p will be used along **ALL** subsequent paths.

- **Performance**: never increase the # of computations on any path.
  - Under simple model, guarantees program won’t get worse.
  - Reality: might increase register lifetimes, add copies, lose.
Preparing the Flow Graph

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

- **Modify the flow graph:**
  - Add a basic block for every edge that leads to a basic block with multiple predecessors (not just on critical edges)
    - How does this help the example?
    - To keep algorithm simple: consider each statement as its own basic block and restrict placement of instructions to the beginning of a basic block

```
d = b + c
a = b + c

Safe!
```
II. Lazy Code Motion Algorithm

- Pass 1: Anticipated Expressions
- Pass 2: (Will be) Available Expressions
- Pass 3: Postponable Expressions
- Pass 4: Used Expressions

Big picture:
- First calculates the “earliest” set of blocks for insertion
  - this maximizes redundancy elimination
  - but may also result in long register lifetimes
- Then it calculates the “latest” set of blocks for insertion
  - achieving the same amount of redundancy elimination as “earliest”
  - but hopefully reducing the lifetime of the register holding the value of the expression
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></th>
<th>Anticipated 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>$f_b(x) = EUse_b \cup (x - EKill_b)$</td>
</tr>
<tr>
<td></td>
<td>EUse: used exp, EKill: exp killed</td>
</tr>
<tr>
<td>$\land$</td>
<td>$\cap$</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 frontier of anticipation
    (boundary between not anticipated and anticipated)
Example 1

See the algorithm in action

Where is \( a + b \) anticipated?

Add BB for every edge to BB with multiple predecessors

- What is the result if we insert at the frontier of anticipation?
Example 2
(Loop Invariant Code)

- Was inserting at the frontier of anticipation the right thing to do in this case?
  - doesn’t eliminate redundancy within loop (why not?)

Add BB for every edge to BB with multiple predecessors

Where is \( a + b \) anticipated?
Example 3
(More Complex Loop)

Where would we ideally like to insert “a+b” in this case?
What happens if we insert at the frontier of anticipation?

IN[i] = EUse[i] ∪ (OUT[i] - EKill[i])
Meet = ∩

• Only in added block on left
• Insert in both yellow blocks

15745: Lazy Code Motion
Example 4
(Variation on Previous Loop)

\[
\text{IN}[i] = \text{EUse}[i] \cup (\text{OUT}[i] - \text{EKill}[i])
\]

\[
\text{Meet} = \bigcap
\]

• Is there any opportunity to eliminate redundancy here?
  no: unsafe to insert in left added block (a+b not anticipated there)
  (e.g. “a+b” could be “b/a” & orange block could be “if a > 0”)

\[
x = a + b
\]

\[
y = a + b
\]

\[
a = 0
\]

\[
\text{IN}[i] = \text{EUse}[i] \cup (\text{OUT}[i] - \text{EKill}[i])
\]

\[
\text{Meet} = \bigcap
\]

\[
x = a + b
\]

\[
y = a + b
\]

\[
a = 0
\]
Pass 2: Place As Early As Possible

There is still some redundancy left!

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

<table>
<thead>
<tr>
<th></th>
<th>(will be) Available Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>Sets of expressions</td>
</tr>
<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>$\land$</td>
<td>$\cap$</td>
</tr>
<tr>
<td>Boundary</td>
<td>$\text{out[entry]} = \emptyset$</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
  - calculated from results of first 2 passes
- **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:
      create a new variable t
      t = x+y,
      replace every original x+y by t

**Result:**
- Maximized redundancy elimination
- Placed as early as possible
- But: register lifetimes?
Pass 3: Postponable Expressions

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 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 $f_b(x) = (\text{earliest}[b] \cup x) - \text{EUse}_b$</td>
<td></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>
Example Illustrating “Postponable”

Ant: 1 Av: 1 P: 0
Av: 1
Ant: 1 Av: 1 P: 0
Ant: 1 Av: 0 P: 0
Ant: 0 Av: 0 P: 0
Entry

Earliest (Ant=1, Av=0)

Ant: 1 Av: 1 P: 1

Ant: 1 Av: 1 P: 1

Ant: 1 Av: 1 P: 1

Ant: 1 Av: 1 P: 1

Ant: 1 Av: 1 P: 0

Ant: 0

Exit

\[ \text{Earliest} = \text{Ant} \cup \text{Av} - \text{EKill} \]

\[ \text{Ant.IN}[i] = \text{EUse}[i] \cup (\text{Ant.OUT}[i] - \text{EKill}[i]) \]

\[ \text{Avail.OUT}[i] = (\text{Ant.IN}[i] \cup \text{Avail.IN}[i]) - \text{EKill}[i] \]

\[ \text{Post.OUT}[i] = (\text{Earliest}[i] \cup \text{Post.IN}[i]) - \text{EUse}[i] \]

\[ \text{EUse} = \text{TRUE} \]

(cause: \( P = 0 \))
Latest: frontier at the end of “postponable” cut set

- \( \text{latest}[b] = (\text{earliest}[b] \cup \text{postponable.in}[b]) \cap \\
  (\text{EUse}_b \cup \neg(\bigcap_{s \in \text{succ}[b]} (\text{earliest}[s] \cup \text{postponable.in}[s]))) \)

  - OK to place expression: \text{earliest} or \text{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:

\[ \begin{align*}
\text{OK to place} & \quad \text{OK to place} \\
\text{OK to place} & \quad \text{not OK to place}
\end{align*} \]
Example Illustrating “Latest”

\[ \text{latest}[b] = (\text{earliest}[b] \cup \text{postponable.in}[b]) \cap \\
(E\text{Use}_b \cup \neg(\bigcap_{s \in \text{succ}[b]}(\text{earliest}[s] \cup \text{postponable.in}[s]))) \]
Pass 4: Used Expressions

Finally… this is easy, it is like liveness (for expressions)

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

\[
x = a + b
\]

not used afterwards

<table>
<thead>
<tr>
<th>Domain</th>
<th>Sets of expressions</th>
</tr>
</thead>
<tbody>
<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}[\text{exit}] = \emptyset)</td>
</tr>
<tr>
<td>Initialization</td>
<td>(\text{in}[b] = \emptyset)</td>
</tr>
</tbody>
</table>
Code Transformation

- For all basic blocks $b$,
  
  $$(x+y) \in (\text{latest}[b] \cap \text{used.out}[b])$$

  at beginning of $b$: 
  
  add $new \ t = x+y$

  replace every original $x+y$ by $t$
4 Passes for Partial Redundancy Elimination

1. **Safety**: 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

2. **Squeezing the last drop of redundancy**: An anticipation frontier may cover a subsequent frontier
   - Pass 2 (forward): **Availability**
   - **Earliest**: anticipated, but not yet available

3. **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

4. **Cleaning up**
   - Pass 4 (backward): **Remove unneeded temporary assignments**
Remarks

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

• **Illustrates the power of data flow**
  – Multiple data flow problems
Today’s Class

I. Mathematical concept: a cut set

II. Lazy Code Motion Algorithm
   • Pass 1: Anticipated Expressions
   • Pass 2: (Will be) Available Expressions
   • Pass 3: Postponable Expressions
   • Pass 4: Used Expressions

Wednesday’s Class

• Static Single Assignment
  – ALSU 6.2.4