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

I. Common Subexpression Elimination

Which \( b + c \) is a common subexpression?

- 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

II. Loop Invariant Code Motion

Which \( b + c \) can be moved to header?

- 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?

CS243: Partial Redundancy Elimination 4

- 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 observations:**
  - A bidirectional data flow problem ("Placement Possible") can be replaced with 4 separate unidirectional data flow problems
    - backward, forward, forward, backward
    - makes it much easier to implement
  - Attempts to minimize register lifetimes (while eliminating redundancy)

- **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
    - achieve the same amount of redundancy elimination as "earliest"
    - but hopefully reduces register lifetimes

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

- **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 without use
- **Range where a+b is anticipated** Choice
Review: Finding Partially Available Expressions using PAVIN

- Forward flow problem
  - Lattice = \{0, 1\}, meet is union (\cup), Top = 0, entry = 0, init = 0
  - \text{PAVOUT}[b] = (\text{PAVIN}[b] - \text{KILL}[b]) \cup \text{AVLOC}[b]
  - \text{PAVIN}[b] =
  - For a block,
    - Expression is \textit{locally available (AVLOC)} if computed & downwards exposed.
    - Expression is killed (\text{KILL}) if any assignments to operands.

Pass 1: Anticipated Expressions

- \text{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)
  - Add BB for every edge to BB with multiple predecessors

Example 1

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

Example 2

- \text{IN}[i] = \text{EUse}[i] \cup (\text{OUT}[i] - \text{EKill}[i])
- Add BB for every edge to BB with multiple predecessors
  - Was inserting at the \textit{frontier of anticipation} the right thing to do in this case?
    - doesn’t eliminate redundancy within loop (why not?)
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?

```
x = a + b
y = a + b
a = 10
```

Example 4
(Variation on Previous Loop)

- Is there any opportunity to eliminate redundancy here?

```
x = a + b
y = a + b
a = 0
```

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

```
Available 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>(Anticipated[b].in \ U - EKill)</td>
</tr>
<tr>
<td>Meet</td>
<td>^</td>
</tr>
<tr>
<td>Boundary</td>
<td>out[entry] = \Ø</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
  - 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 \( \in \) 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: 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 earliest placement of \( e \) but not a subsequent use

Let’s be lazy without introducing redundancy

**Postponable Expressions**

- **Domain**: Sets of expressions
- **Direction**: forward
- **Transfer Function**:
  \[ f(x) = (\text{earliest} \cup \text{postponable}) - \text{EUse}_b \]
  \[ \wedge \]
  \[ \wedge \]
- **Boundary**: \( \text{out}[\text{entry}] = \emptyset \)
- **Initialization**: \( \text{out}[b] = \{ \text{all expressions} \} \)

• Delay creating redundancy to reduce register pressure

\[ x = b + c \]
\[ y = b + c \]
\[ b = 1 \]

Anticipated.in (Ant)  Available.in (Av)  Postponable.in (P)

Entry  Exit

Ant: 1  Ant: 0  Ant: 1  Ant: 1  Ant: 1  Ant: 1
Av: 1  Av: 1  Av: 1  Av: 0  Av: 0  Av: 0
Earliest  P: 0  P: 0  P: 0  P: 0  P: 0  P: 0

OK to place

OK to place  not OK to place

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: 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:

Example Illustrating “Latest”

- \( \text{latest}[b] = (\text{earliest}[b] \cup \text{postponable.in}[b]) \cap (\text{EUse}_b \cup \neg \bigcap_{s \in \text{succ}[b]} (\text{earliest}[s] \cup \text{postponable.in}[s])) \)
- OK to place
- not OK to place
Pass 4: Cleaning Up

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

\[ x = a + b \]

not used afterwards

- 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</td>
</tr>
<tr>
<td>Boundary</td>
</tr>
<tr>
<td>Initialization</td>
</tr>
</tbody>
</table>

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 \( t \)

Remarks

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

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
Monday's Class

• Region-Based Analysis [ALSU 9.7]

• Reminder: Assignment #2 due Wednesday midnight