Lazy Code Motion

Bojian Zheng
CSCD70 Spring 2018
bojian@cs.toronto.edu
Partial Redundancy Elimination

- Can we replace calculations of $b + c$ such that no path re-executes the same expression?
- subsumes
  - Global Common Subexpression
    - Full Redundancy
  - Loop Invariant Code Motion
    - Partial Redundancy for-loops
Common Subexpression Elimination

• On every path reaching $p$
  • Expression $b + c$ has been computed.
  • Neither $b$ nor $c$ is 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?
Lazy Code Motion
Lazy Code Motion

• The optimization of eliminating partial redundancy with the goal of delaying the computations as much as possible.
• How are we going to achieve this?
  • Anticipated Expressions & Will-be-Available Expressions
  • Postponable Expressions
  • Used Expressions
Our Goal

- Safety
- Maximum Redundancy Elimination
- Shortest Register Lifetime
Anticipated Expressions
Safety

• We cannot introduce operations that are not executed originally.
• Given the diagram on the right, can we insert the expression $b + c$ on the right parent?
Anticipated Expressions

• An expression $e$ is said to be **anticipated** at program point $p$ if all paths leading from $p$ eventually computes $e$ (from the values of $e$’s operands that are available at $p$).

<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>$f_b(x) = \text{EUse}_b \cup (x \cdot \text{EKil}_b)$</td>
</tr>
<tr>
<td></td>
<td>EUse: exp used, EKil: exp killed</td>
</tr>
<tr>
<td>Boundary</td>
<td>$\text{in}[\text{exit}] = \emptyset$</td>
</tr>
<tr>
<td>Initialization</td>
<td>$\text{in}[b] = {\text{all expressions}}$</td>
</tr>
</tbody>
</table>
Safety

• We cannot introduce operations that are not executed originally.

• Given the diagram on the right, can we insert the expression $b + c$ on the right parent?

• NO! The reason is because $b + c$ is not anticipated at the right parent.
Critical Edge

• If the source has multiple successors, and the destination has multiple predecessors, then the path that is connecting them is defined as **Critical Edge**.
Solution: Synthetic Block

• Add a basic block for every edge that leads to a basic block with multiple predecessors (not just the back edge).

• This simplifies the algorithm – since we can always place at the beginning of the basic block.
Example 1

What is the result after insertion at the **anticipation frontier**?

[Diagram showing a computational tree with nodes labeled as `x = a + b`, `y = a + b`, `r = a + b`, and `z = a + b` with values 0 and 1 at various points.]
Example 2: Loop Invariance

Will insertion at the **anticipation frontier** help in this case?
Example 3: More Complex Loop

Where are we expecting to place expression $a + b$? Where will it actually be placed?
Example 4: Complex Loop Variation

Can we place expression $a + b$ at the left synthetic block like what we did previously?
Questions?

• Keywords:
  • Safety
  • Anticipated Expressions
  • Synthetic Block
Will-be-Available Expressions
Complications

• Does the **anticipation frontier** approach always work?
• The reason is because we have not yet considered expression **availability**.
• Want to make the expression \( e \) available **wherever it is anticipated but unavailable**.
Will-be-Available Expressions

• An expression $e$ is said to be **will-be-available** at program point $p$ if it is anticipated and not subsequently killed along all paths reaching $p$.

• Note how it is different from Available Expressions.

<table>
<thead>
<tr>
<th></th>
<th>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>$f_0(x) = (\text{Anticipated}[b].\text{in} \cup x) - \text{EKill}_b$</td>
</tr>
<tr>
<td>$\wedge$</td>
<td>$\bigcap$</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

- \( \text{earliest}(b) \) is the set of expressions added to block \( b \) under \textbf{early placement}, and is computed from the results of \textit{anticipated} and \textit{will-be-available}.

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

- Where is the *earliest* placement?
- Is it different from the *anticipation frontier*?
Questions?

• Keywords:
  • Will-be-Available Expressions
  • Early Placement
Postponable Expressions
Shortest Register Lifetime?

• **Early Placement** goes against our goal of **shortest register lifetime**.

• We want to delay creating redundancy to reduce register pressure.
Postponable Expressions

- An expression $e$ is said to be **postponable** at program point $p$ if all paths leading to $p$ have seen earliest placement of $e$ but not a subsequent use.

<table>
<thead>
<tr>
<th></th>
<th>Postponable Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong></td>
<td>Sets of expressions</td>
</tr>
<tr>
<td><strong>Direction</strong></td>
<td>forward</td>
</tr>
<tr>
<td><strong>Transfer Function</strong></td>
<td>$f_i(x) = (\text{earliest}[b] \cup x) - \text{EUse}_b$</td>
</tr>
<tr>
<td>$\land$</td>
<td>$\cup$</td>
</tr>
<tr>
<td><strong>Boundary</strong></td>
<td>out[entry] = $\emptyset$</td>
</tr>
<tr>
<td><strong>Initialization</strong></td>
<td>out[b] = {all expressions}</td>
</tr>
</tbody>
</table>
Example

```
Example

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

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

EUse = TRUE
(causes P = 0)
```
Latest Placement

• We define the term Latest as follows:
  • It is ok to place the expression $e$: either Earliest ① or Postponable ②.
  • Need to place at $b$ if either:
    • $e$ is used in $b$ ③.
    • It is NOT ok to place in one of its successors ④.

$$\text{Latest}(b) = \left( \left( \text{earliest}(b) \cup \text{postponable}(b) \right) \cap \left( \text{EUse}(b) \cup \neg \bigcap_{s \in \text{succ}(b)} (\text{postponable}(s)) \right) \right)$$
Example

![Diagram of computational process]

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

• Keywords:
  • Postponable Expressions
  • Latest Placement
Used Expressions
Used Expressions

• An expression $e$ is said to be **used** at program point $p$ if there exists a path leading from $p$ that uses the expression before the operands are reevaluated.

<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>$\wedge$</td>
</tr>
<tr>
<td>Boundary</td>
</tr>
<tr>
<td>Initialization</td>
</tr>
</tbody>
</table>
Final Placement

• Our code transformation goes as follows:

\[ \forall b, \text{ if expression } e \in (\text{latest}(b) \cap (\text{used}(b))) \]

at the beginning of \( b \), insert \( t = e \), and replace every original \( e \) with \( t \)
Summary

- Synthetic Blocks
  - Anticipated Expressions
    - Earliest
      - Latest
        - Placement
  - Will-be-Available Expressions
    - Postponable Expressions
      - Used Expressions