Reuse Optimization

Last time
- Value numbering

Today
- Common subexpression elimination (CSE)

Common Subexpression Elimination

Idea
- Find common subexpressions whose range spans the same basic blocks and eliminate unnecessary re-evaluations
- Leverage available expressions

Recall available expressions
- An expression (e.g., \(x+y\)) is available at node n if every path from the entry node to n evaluates \(x+y\), and there are no definitions of \(x\) or \(y\) after the last evaluation along that path

Strategy
- If an expression is available at a point where it is evaluated, it need not be recomputed
CS553 Lecture  
Reuse Optimization: Common SubExpr Elim

CSE Example

Will value numbering find this redundancy?
- No; value numbering operates on values
- CSE operates on expressions

Another CSE Example

**Before CSE**
- \( c := a + b \)
- \( d := m \& n \)
- \( e := b + d \)
- \( f := a + b \)
- \( g := -b \)
- \( h := b + a \)
- \( a := j + a \)
- \( k := m \& n \)
- \( j := b + d \)
- \( a := -b \)
- \( \text{if } m \& n \text{ goto L2} \)

**Summary**
- 11 instructions
- 12 variables
- 9 binary operators

**After CSE**
- \( t1 := a + b \)
- \( c := t1 \)
- \( t2 := m \& n \)
- \( d := t2 \)
- \( t3 := b + d \)
- \( e := t3 \)
- \( f := t1 \)
- \( g := -b \)
- \( h := t1 \)
- \( a := j + a \)
- \( k := t2 \)
- \( j := t3 \)
- \( a := -b \)
- \( \text{if } t2 \text{ goto L2} \)

**Summary**
- 14 instructions
- 15 variables
- 4 binary operators
CSE Approach 1

Notation
- \( \text{Avail}(b) \) is the set of expressions available at block \( b \)
- \( \text{Gen}(b) \) is the set of expressions generated and not killed at block \( b \)

If we use \( e \) and \( e \in \text{Avail}(b) \)
- Allocate a new name \( n \)
- Search backward from \( b \) (in CFG) to find statements (one for each path) that most recently generate \( e \)
- Insert copy to \( n \) after generators
- Replace \( e \) with \( n \)

Example

\[
\begin{align*}
a & := b + c \\
t_1 & := a \\
t_2 & := a \\
e & := b_1 + c \\
f & := b_2 + c
\end{align*}
\]

Problems
- Backward search for each use is expensive
- Generates unique name for each use
  - \(|\text{names}| < |\text{Uses}| > |\text{Avail}|\)
  - Each generator may have many copies

Example

\[
\begin{align*}
a & := b + c
\end{align*}
\]

CSE Approach 2

Idea
- Reduce number of copies by assigning a unique name to each unique expression

Summary
- \( \forall e \) Name\([e]\) = unassigned
- if we use \( e \) and \( e \in \text{Avail}(b) \)
  - if Name\([e]\)=unassigned, allocate new name \( n \) and Name\([e]\) = \( n \)
  - else \( n = \text{Name}[e] \)
  - Replace \( e \) with \( n \)
- In a subsequent traversal of block \( b \), if \( e \in \text{Gen}(b) \) and Name\([e]\) ≠ unassigned, then insert a copy to Name\([e]\) after the generator of \( e \)

Problem
- May still insert unnecessary copies
- Requires two passes over the code

Example

\[
\begin{align*}
a & := b + c \\
t_1 & := a
\end{align*}
\]
CSE Approach 3

Idea
– Don’t worry about temporaries
– Create one temporary for each unique expression
– Let subsequent pass eliminate unnecessary temporaries

At an evaluation of e
– Hash e to a name, n, in a table
– Insert an assignment of e to n

At a use of e in b, if e ∈ Avail(b)
– Lookup e’s name in the hash table (call this name n)
– Replace e with n

Problems
– Inserts more copies than approach 2 (but extra copies are dead)
– Still requires two passes (2nd pass is very general)

Extraneous Copies

Extraneous copies degrade performance

Let other transformations deal with them
– Dead code elimination
– Coalescing

Coalesce assignments to t1 and t2 into a single statement
  \[ t1 := b + c \]
  \[ t2 := t1 \]

– Greatly simplifies CSE
Partial Redundancy Elimination (PRE)

Partial Redundancy
- An expression (e.g., \(x+y\)) is partially redundant at node \(n\) if some path from the entry node to \(n\) evaluates \(x+y\), and there are no definitions of \(x\) or \(y\) between the last evaluation of \(x+y\) and \(n\).

Elimination
- Discover partially redundant expressions
- Convert them to fully redundant expressions
- Remove redundancy

PRE subsumes CSE and loop invariant code motion

Loop Invariance Example

PRE removes loop invariants
- An invariant expression is partially redundant
- PRE converts this partial redundancy to full redundancy
- PRE removes the redundancy

Example