Reuse Optimization

Last time
- Discussion (SCC)
- Loop invariant code motion
- Reuse optimization: Value numbering

Today
- More reuse optimization
  - Common subexpression elimination (CSE)
  - Partial redundancy elimination (PRE)

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
CSE Example

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

Another CSE Example

Before CSE

\[
\begin{align*}
&c := a + b \\
&d := m \& n \\
&e := b + d \\
&f := a + b \\
&g := -b \\
&h := b + a \\
&a := j + a \\
&k := m \& n \\
&j := b + d \\
&a := -b \\
&\text{if } m \& n \text{ goto L2}
\end{align*}
\]

Summary

- 11 instructions
- 12 variables
- 9 binary operators

After CSE

\[
\begin{align*}
&t1 := a + b \\
&c := t1 \\
&t2 := m \& n \\
&d := t2 \\
&t3 := b + d \\
&e := t3 \\
&f := t1 \\
&g := -b \\
&h := t1 \\
&a := j + a \\
&k := t2 \\
&j := t3 \\
&a := -b \\
&\text{if } t2 \text{ goto L2}
\end{align*}
\]

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
- \( a := b + c \) (Example)
- \( t_1 := a \)
- \( t_2 := a \)
- \( e := b + c \)
- \( f := b + c \)

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

CSE Approach 2

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

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

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

Example
- \( a := b + c \)
- \( t_1 := a \)
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

\[
t_1 := b + c \\
t_2 := t_1
\]

– 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

\begin{align*}
1 & : x := y * z \\
2 & : \ldots \\
& : a := b + c \\
1 & : x := y * z \\
2 & : \ldots \\
& : a := b + c \\
1 & : x := y * z \\
2 & : \ldots \\
& : a := b + c
\end{align*}
Implementing PRE

**Big picture**
- Use local properties (availability and anticipability) to determine where redundancy can be created within a basic block
- Use global analysis (data-flow analysis) to discover where partial redundancy can be converted to full redundancy
- Insert code and remove redundant expressions

Local Properties

An expression is locally **transparent** in block b if its operands are not modified in b

An expression is locally **available** in block b if it is computed at least once and its operands are not modified after its last computation in b

An expression is locally **anticipated** if it is computed at least once and its operands are not modified before its first evaluation

Example

<table>
<thead>
<tr>
<th>Variable</th>
<th>Expression</th>
<th>Transparency</th>
<th>Availability</th>
<th>Anticipation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b + c</td>
<td>b + c</td>
<td>b + c, a + e</td>
<td>b + c</td>
</tr>
<tr>
<td>d</td>
<td>a + e</td>
<td></td>
<td></td>
<td>b + c</td>
</tr>
</tbody>
</table>
Local Properties (cont)

How are these properties useful?
- They tell us where we can introduce redundancy

**Transparent**

The expression can be redundantly evaluated anywhere in the block

**Available**

\[ a = b + c \]

The expression can be redundantly evaluated anywhere after its last evaluation in the block

**Anticipated**

\[ a = b + c \]

The expression can be redundantly evaluated anywhere before its first evaluation in the block

Global Availability

**Intuition**
- Global availability is the same as Available Expressions
- If \( e \) is globally available at \( p \), then an evaluation at \( p \) will create redundancy along all paths leading to \( p \)

**Flow Functions**

\[
\begin{align*}
\text{available}_\text{in}[n] &= \bigcap_{p \in \text{pred}(n)} \text{available}_\text{out}[p] \\
\text{available}_\text{out}[n] &= \text{locally}_\text{available}[n] \cup (\text{available}_\text{in}[n] \cap \text{transparent}[n])
\end{align*}
\]
(Global) Partial Availability

Intuition
– An expression is partially available if it is available along some path
– If e is partially available at p, then there exists a path from the entry node to p such that the evaluation of e at p would give the same result as the previous evaluation of e along the path

Flow Functions

\[
\text{partially\_available\_in}[n] = \bigcup_{p \in \text{pred}[n]} \text{partially\_available\_out}[p]
\]

\[
\text{partially\_available\_out}[n] = \text{locally\_available}[n] \cup (\text{partially\_available\_in}[n] \cap \text{transparent}[n])
\]

Global Anticipability

Intuition
– If e is globally anticipated at p, then an evaluation of e at p will make the next evaluation of e redundant along all paths from p

Flow Functions

\[
\text{anticipated\_out}[n] = \bigcap_{s \in \text{succ}[n]} \text{anticipated\_in}[s]
\]

\[
\text{anticipated\_in}[n] = \text{locally\_anticipated}[n] \cup (\text{anticipated\_out}[n] \cap \text{transparent}[n])
\]
Global Possible Placement

Goal
– Convert partial redundancies to full redundancies
– Possible Placement uses a backwards analysis to identify locations where such conversions can take place
  – $e \in \text{ppin}[n]$ can be placed at entry of $n$
  – $e \in \text{ppout}[n]$ can be placed at exit of $n$

Start with locally anticipated expressions

Push Possible Placement backwards as far as possible

Global Possible Placement (cont)

The placement will create a redundancy on every edge out of the block

Flow Functions

\[
\text{ppout}[n] = \bigcap_{s \in \text{succ}[n]} \text{ppin}[s]
\]

\[
\text{ppin}[n] = \text{anticipated_in}[n] \cap \text{partially_available_in}[n] \cap (\text{locally_anticipated}[n] \cup (\text{ppout}[n] \cap \text{transparent}[n]))
\]

Middle of chain
This block is at the beginning of a chain
Will turn partial redundancy into full redundancy
Updating Blocks

Intuition
– Perform insertions at top of the chain
– Perform deletion at the bottom of the chain

Functions
– delete[n] = ppin[n] ∩ locally_anticipated[n]
– insert[n] = ppout[n] ∩ (¬ppin[n] ∪ ¬transparent[n]) ∩ ¬available_out[n] Don’t insert it where it’s fully redundant

Updating Blocks (cont)

Intuition
– Perform insertions at top of the chain
– Perform deletion at the bottom of the chain

Functions
– delete[n] = ppin[n] ∩ locally_anticipated[n]
– insert[n] = ppout[n] ∩ (¬ppin[n] ∪ ¬transparent[n]) ∩ ¬available_out[n]

¬ppout[n]? No
Can we omit this clause?
### Example

B1: \[ a := b + c \]

B2: \[ b := b + 1 \]

B3: \[ a := b + c \]

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>transparent</td>
<td>( b+c )</td>
<td>( b+c )</td>
<td>( b+c )</td>
</tr>
<tr>
<td>locally_available</td>
<td>( b+c )</td>
<td>( b+c )</td>
<td>( b+c )</td>
</tr>
<tr>
<td>locally_anticipated</td>
<td>( b+c )</td>
<td>( b+1 )</td>
<td>( b+c )</td>
</tr>
<tr>
<td>available_in</td>
<td>( b+c )</td>
<td>( b+c )</td>
<td>( b+c )</td>
</tr>
<tr>
<td>partially_available_in</td>
<td>( b+c )</td>
<td>( b+c )</td>
<td>( b+c )</td>
</tr>
<tr>
<td>partially_available_out</td>
<td>( b+c )</td>
<td>( b+c )</td>
<td>( b+c )</td>
</tr>
<tr>
<td>anticipated_out</td>
<td>( b+c )</td>
<td>( b+c )</td>
<td>( b+c )</td>
</tr>
<tr>
<td>anticipated_in</td>
<td>( b+c )</td>
<td>( b+1 )</td>
<td>( b+c )</td>
</tr>
<tr>
<td>ppout</td>
<td>( b+c )</td>
<td>( b+1 )</td>
<td>( b+c )</td>
</tr>
<tr>
<td>ppin</td>
<td>( b+c )</td>
<td>( b+c )</td>
<td>( b+c )</td>
</tr>
<tr>
<td>insert</td>
<td>( b+c )</td>
<td>( b+c )</td>
<td>( b+c )</td>
</tr>
<tr>
<td>delete</td>
<td>( b+c )</td>
<td>( b+c )</td>
<td>( b+c )</td>
</tr>
</tbody>
</table>

### Comparing Redundancy Elimination

**Value numbering**
- Examines values not expressions
- Symbolic
- Knows nothing about algebraic properties \((1+x = x+1)\)

**CSE**
- Examines expressions

**PRE**
- Examines expressions
- Subsumes CSE and loop invariant code motion
- Simpler implementations are now available

**Constant propagation**
- Requires that values be statically known
PRE Summary

What’s so great about PRE?
- A modern optimization that subsumes earlier ideas
- Composes several simple data-flow analyses to produce a powerful result
  - Finds earliest and latest points in the CFG at which an expression is anticipated

Next Time

Lecture
- Alias analysis