Loop Invariant Code Motion

**Last Time**
- Loop invariant code motion
- Value numbering

**Today**
- Finish value numbering
- More reuse optimization
  - Common subexpression elimination
  - Partial redundancy elimination

**Next Time**
- Something special

Comparing Optimistic and Pessimistic Approaches

**Differences**
- Handling of loops
- Pessimistic makes worst-case assumptions on back edges
- Optimistic requires actual contradiction to split classes
Role of SSA

**Single global result**
- Variables correspond to values

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

\[
\begin{align*}
a_1 & = b \\
a_2 & = c \\
a_3 & = d
\end{align*}
\]

- a not congruent to anything
- Congruence classes: \{a_1, b\}, \{a_2, c\}, \{a_3, d\}

**No data flow analysis**
- Optimistic: Iterate over congruence classes, not CFG nodes
- Pessimistic: Visit each assignment once

**Φ-functions**
- Make data-flow merging explicit
- Treat like normal functions

Reuse Optimization

**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 eliminates 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

Is CSE strictly better than value numbering?
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

Which is better?

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

Idea
- If block b uses expression e, and e is available
- Search backward from b (in CFG) to find the statement on each path that most recently generates e
- Insert copy to n after generators
- Replace e with n

Is this a good approach?

x and y not defined along yellow edges
CSE Approach 1 (cont)

**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 statement on each path that most recently generates $e$
- Insert copy to $n$ after generators
- Replace $e$ with $n$

**Problems?**
- Backward search for each use is expensive
- Generates unique name for each use
  - $|\text{names}| \neq |\text{Uses}| > |\text{Avail}|$
  - Each generator may have many copies

**Example**

```
   a := b + c
   t1 := a

   t2 := a
   e := b + c

   b: f := b + c
```

February 25, 2015

CSE Approach 2

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

**Summary**

uses
- $\forall e \, \text{Name}[e] = \text{unassigned}$
- 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 $n$

defs
- 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**
- Requires two passes over the code
- May still insert unnecessary copies
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 $e \in \text{Avail}(b)$
– 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)

Comparing the Three Approaches

Approach 1 and Approach 2
– Make decisions about when to insert temporaries
– Approach 1:
  – Insert temporaries as we look for redundant expressions
  – One temporary per use of redundant expression
– Approach 2
  – Use a second pass to insert temporaries

Approach 3
– Don’t worry about temporaries!
Extraneous Copies

Extraneous copies degrade performance

Let other transformations deal with them
– Dead code elimination
– Copy propagation
  Coalesce assignments to $t_1$ and $t_2$ into a single statement
  
  $t_1 := b + c$
  $t_2 := t_1$

– Greatly simplifies CSE

Loop Invariant Code Motion

Last Time
– Loop invariant code motion
– Value numbering

Today
– Finish value numbering
– More reuse optimization
  – Common subexpression elimination
  – Partial redundancy elimination

Next Time
– Something special
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 \( n \).

![Diagram of Partial Redundancy]

Question
- Can we remove partially redundant code?
- Yes. It’s a three step process

Three Steps
- Discover partially redundant expressions
- Convert them to fully redundant expressions
- Remove the redundancy

Is this beneficial?
- PRE subsumes CSE and loop invariant code motion

![Diagram of Three Steps]
Loop Invariance Example

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

**Example**

Implementing PRE

**Big picture**
- Use local properties (available and anticipated) 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**

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Transparent</th>
<th>Available</th>
<th>Anticipated</th>
</tr>
</thead>
<tbody>
<tr>
<td>b + c</td>
<td>{b + c}</td>
<td>{b + c, a + e}</td>
<td>{b + c}</td>
</tr>
</tbody>
</table>

**Questions?**

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

– In which direction does anticipation flow?
Local Properties (cont)

**Example**

- For each block in the following figure, what are the local properties with respect to $expr$?

![Diagram showing local properties example]

- Where can $e$ be computed redundantly?

Implementing PRE

**Big picture**

- Use local properties (available and anticipated) 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

![Diagram showing implementation of PRE]

Global Analysis for PRE

We’ll need three global analyses
- Globally available
- Partially available
- Globally anticipated

Global Analysis

Intuition
- **Globally available** 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 $p$

Data-flow Equations

$$\text{available}_{\text{in}}[n] = \bigcap_{p \in \text{pred}[n]} \text{available}_{\text{out}}[p]$$

$$\text{available}_{\text{out}}[n] = [\text{locally}_{\text{available}}[n] \cup (\text{available}_{\text{in}}[n] \cap \text{transparent}[n])]$$
(Globally) Partially Available

**Intuition**
- An expression is **partially available** if it is available along some path.
- If $e$ is partially available at $p$, then $\exists$ 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.

**Data-flow Equations?**

$$\text{partially\_available\_in}[n] = \bigcup_{p \in \text{pred}[n]} \text{partially\_available\_out}[p]$$
$$\text{partially\_available\_out}[n] = \text{locally\_available}[n] \bigcup (\text{partially\_available\_in}[n] \cap \text{transparent}[n])$$

Globally Anticipated

**Intuition**
- If $e$ is **globally anticipated** at $p$, then adding an evaluation of $e$ at $p$ will make $e$ redundant along all paths from $p$, i.e., you’re expecting $e$ to be computed in the future.

**Data-flow Equations?**

$$\text{anticipated\_out}[n] = \bigcap_{s \in \text{succ}[n]} \text{anticipated\_in}[s]$$
$$\text{anticipated\_in}[n] = \text{locally\_anticipated}[n] \bigcup (\text{anticipated\_out}[n] \cap \text{transparent}[n])$$
Big picture

- Use local properties (available and anticipated) 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

Global Possible Placement

Goal

- Convert partial redundancies to full redundancies
- **Possible Placement** is a backwards analysis that identifies 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

Data-flow Equations

\[
\begin{align*}
\text{ppout}[n] &= \bigcup_{s \in \text{succ}[n]} \text{ppin}[s] \\
\text{ppin}[n] &= \text{anticipated_in}[n] \bigcup \text{partially_available_in}[n] \\
&\quad \cap \left( \text{locally_anticipated}[n] \bigcup (\text{ppout}[n] \bigcap \text{transparent}[n]) \right)
\end{align*}
\]

Will turn partial redundancy into full redundancy

Middle of chain

This block is at the beginning of a chain

How do we ensure that it is redundant on every edge out of the block?

Updating Blocks

Intuition

- Perform insertion at tops of the chain
- Perform deletion at the bottoms of the chain

Data-flow Equations

\[
\begin{align*}
\text{insert}[n] &= \text{ppout}[n] \\
&\quad \cap \left( \neg \text{ppin}[n] \bigcup \neg \text{transparent}[n] \right) \\
&\quad \cap \neg \text{available_out}[n] \\
\text{delete}[n] &= \text{ppin}[n] \bigcap \text{locally_anticipated}[n]
\end{align*}
\]

Don’t insert it where it’s fully redundant

February 25, 2015
Reuse Optimization II
Updating Blocks (cont)

Intuition

- Perform insertion at tops of the chain
- Perform deletion at the bottoms of the chain

Functions

- \( \text{insert}[n] = \text{ppout}[n] \cap (\neg \text{ppin}[n] \cup \neg \text{transparent}[n]) \cap \neg \text{available}_\text{out}[n] \cap \neg \text{ppout}[n] \)?
  - Can we omit this clause?
  - No

- \( \text{delete}[n] = \text{ppin}[n] \cap \text{locally}_\text{anticipated}[n] \)

Exercise

\begin{align*}
\text{B1}: & \quad \text{a} := \text{b} + \text{c} \\
\text{B2}: & \quad \text{b} := \text{b} + 1 \\
\text{B3}: & \quad \text{a} := \text{b} + \text{c}
\end{align*}

<table>
<thead>
<tr>
<th>transparent</th>
<th>locally available</th>
<th>locally anticipated</th>
<th>available_in</th>
<th>available_out</th>
<th>partially_available_in</th>
<th>partially_available_out</th>
<th>anticipated_out</th>
<th>anticipated_in</th>
<th>ppout</th>
<th>ppin</th>
<th>insert</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

February 25, 2015

Reuse Optimization II
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
– Pointer analysis

Assignment 3
– Now available – start early!