Partial Redundancy Elimination & Lazy Code Motion

The "Lazy Code Motion" part of this lecture draws on two papers:
1. "Lazy Code Motion", by J. Knoop, O. Rüthing, and B. Steffen, in PLDI’92;

Common-Subexpression Elimination

An occurrence of an expression in a program is a common subexpression if there is another occurrence of the expression whose evaluation always precedes this one in execution order and the operands of the expression remain unchanged between the two evaluations.

Local Common Subexpression Elimination (CSE) keeps track of the set of available expressions within a basic block and replaces instances of them by references to new temporaries that keep their value.

```
... a = (x+y)+z; b = a-1; c = x+y; ...

... i = x+y; a = i+z; b = a-1; c = i; ...
```

Before CSE

After CSE

Available Expressions

- An expression x*y is available at a program point p if
  - every path from the initial node to p evaluates x*y before reaching p,
  - and there are no assignments to x or y after the evaluation but before p.
- Available expression information can be used to do global (across basic blocks) CSE.
- If an expression is available at the point of its use, there is no need to re-evaluate it.

Computing Available Expressions

- Represent sets of expressions using bit vectors
- Each expression corresponds to a bit
- Run dataflow algorithm similar to reaching definitions
- Notice that:
  - A definition reaches a basic block if it comes from ANY predecessor in CFG.
  - An expression is available at a basic block only if it is available from ALL block’s predecessors in the CFG.

Global CSE Example

```
Expressions
1: x*y
2: i < n
3: i+c
4: x==0

u = x+y; x = 0
x = z;
b = x+y;
i = x+y;
i = i+c;
c = x+y;
i = i+c;
d = x+y
```

Before CSE
Global CSE Transformation

Expressions
1. \( x + y \)
2. \( i < n \)
3. \( i + c \)
4. \( x = 0 \)

must use same temp for CSE in all blocks

Redundant Expressions

An expression is redundant at a point \( p \) if, on every path to \( p \)
1. It is evaluated before reaching \( p \), and
2. None of its constituent values is redefined before \( p \)

Example
\[ a = b + c \]
\[ b = a + b + c \]
\[ a = b + c \]

Not all occurrences of \( b + c \) are redundant!

Some occurrences of \( b + c \) are redundant

Expressions

Partially Redundant Expressions

An expression is partially redundant at \( p \) if it is redundant along
some, but not all, paths reaching \( p \)

Example
\[ b = b + 1 \]
\[ a = b + c \]

Inserting a copy of "\( a = b + c \)" after the definition of \( b \)
can make it redundant

Fully redundant?

Loop Invariant Expressions

Another example
\[ a = y \]
\[ a = y + z \]
\[ a = b + c \]

Loop invariant expressions are partially redundant

- Partial redundancy elimination performs code motion
- Major part of the work is figuring out where to insert operations

Lazy Code Motion

The concept
- Solve data flow problems that reveal limits of code motion
- Compute INSERT & DELETE sets from solutions
- Linear pass over the code to rewrite it (using INSERT & DELETE)

The history
- Partial redundancy elimination (Karel & Reeves, CACM, 1979)
- Improvements by Drescher & Stadel, Zobi & Dhamdhere, Chow,
Knoop, Bathing & Steffen, Dhamdhere, Sarkin, ...
- All versions of PRE optimize placement
- Guarantee that no path is lengthened
- LCM was introduced by Knoop et al. in PLDI, 1992
- We will look at a variation by Drescher & Stadel

Lazy Code Motion

The intuitions
- Compute available expressions
- Compute anticipatable expressions
- These lead to an earlier placement for each expression
- Path expressions down the CFG until it changes behavior

Assumptions
- Uses a *not value* name identity (not value identity)
- Code is in an Intermediate Representation with unlimited name space
- Consistent, disciplined use of names
  - Identical expressions define the same name
  - No other expression defines that name
- Avoids copies
- Result serves as proxy
Lazy Code Motion

The Name Space

- \( r_i \in \{r_1, r_2\} \), always (hash to find it)
- We can refer to \( r_i \) by \( r_u \) (bit-vector size)
- Variables must be set by copies
  - No consistent definition for a variable
  - Break the rule for this case, but require \( r_{var} \neq r_{definition} \)
  - To achieve this, assign register names to variables first

Without this name space

- LCM must insert copies to preserve redundant values
- LCM must compute its own map of expressions to unique ids

Lazy Code Motion: Running Example

\[
B_1: r_1 \leftarrow 1 \\
B_2: r_2 \leftarrow r_1 \\
B_3: \text{expression} \\
B_4: \text{expression} \\
B_5: \text{expression} \\
B_6: \text{expression} \\
B_7: \text{expression}
\]

Variables: \( r_1, r_2, r_3 \)
Expressions: \( r_1, r_2, r_3, r_4, r_5, r_6, r_7 \)

Lazy Code Motion: Running Example

<table>
<thead>
<tr>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEEP</td>
<td>0, 0, 0</td>
<td>( \neq &lt; 0.4 )</td>
</tr>
<tr>
<td>DEEP</td>
<td>0, 1</td>
<td>( &lt; 0.4 )</td>
</tr>
<tr>
<td>KILLED</td>
<td>0, 0, 1</td>
<td>( \neq &lt; 0.4 )</td>
</tr>
</tbody>
</table>

Lazy Code Motion

Predicates (computed by Local Analysis)

- \( \text{DEEP}(b) \) contains expressions defined in \( b \) that survive to the end of \( b \).
  - \( e = \text{DEEP}(b) \Rightarrow \) evaluating \( e \) at the end of \( b \) produces the same value for \( e \) as evaluating \( e \) in its original position.
- \( \text{UEEP}(b) \) contains expressions defined in \( b \) that have upward exposed arguments (bit arg).
  - \( e = \text{UEEP}(b) \Rightarrow \) evaluating \( e \) at the start of \( b \) produces the same value for \( e \) as evaluating \( e \) in its original position.
- \( \text{KILLED}(b) \) contains those expressions whose arguments are \( \text{r} \) defined in \( b \).
  - \( e = \text{KILLED}(b) \Rightarrow \) evaluating \( e \) at the start of \( b \) does not produce the same result as evaluating \( e \) at its end.

Lazy Code Motion: Running Example

\[
B_1: r_1 \leftarrow 1 \\
B_2: r_2 \leftarrow r_1 \\
B_3: \text{expression} \\
B_4: \text{expression} \\
B_5: \text{expression} \\
B_6: \text{expression} \\
B_7: \text{expression}
\]

Variables: \( r_1, r_2, r_3 \)
Expressions: \( r_1, r_2, r_3, r_4, r_5, r_6, r_7 \)

Lazy Code Motion

Availability

\[
\text{Aval}(n) = \bigcap_{n \in \text{dom(m)}} \text{AvalOut}(m), \quad n \neq n_0 \\
\text{AvalOut}(m) = \text{DEEP}(m) \cup (\text{Aval}(m) \cap \text{KILLED}(m))
\]

Initialize \( \text{AvalOut}(n) \) to the set of all names, except at \( n_0 \).
Set \( \text{AvalOut}(n) \) to \( \emptyset \) for each exit block \( n \).

Interpreting \( \text{AvalOut} \)

- \( e = \text{AvalOut}(b) \Rightarrow \) evaluating \( e \) at start of \( b \) produces the same value for \( e \).
- \( \text{AvalOut} \) tells the compiler how far forward \( e \) can move the evolution of \( e \), ignoring any uses of \( e \).
- This differs from the way we talk about \( \text{Aval} \) in global redundancy elimination.

Lazy Code Motion: Running Example

\[
\text{AntOut}(n) = \bigcap_{n \in \text{dom(m)}} \text{AntIn}(m), \quad n \neq n_0 \text{not an exit block} \\
\text{AntIn}(m) = \text{UEEP}(m) \cup (\text{AntOut}(m) \cap \text{KILLED}(m))
\]

Initialize \( \text{AntOut}(n) \) to the set of all names, except at exit blocks.
Set \( \text{AntOut}(n) \) to \( \emptyset \) for each exit block \( n \).

Interpreting \( \text{AntOut} \)

- \( e = \text{AntOut}(b) \Rightarrow \) evaluating \( e \) at start of \( b \) produces the same value for \( e \).
- \( \text{AntOut} \) tells the compiler how far backward \( e \) can move.
- This view shows that anticipability is, in some sense, the inverse of availability (it explains the new interpretation of \( \text{Aval} \)).
Lazy Code Motion

**Earliest placement**

\[
\text{EARLIEST}(i, j) = \text{Antin}(i) \land \text{AVAILOUT}(j) \land (\text{KilledEXP}(i) \lor \text{AntOut}(j))
\]

\[
\text{EARLIEST}(i, 0) = \text{Antin}(i) \land \text{AVAILOUT}(0)
\]

**EARLIEST** is a predicate

- Computed for edges rather than nodes (placement)
- \( e = \text{EARLIEST}(j) \) if
  - It can move to head of \( j \).
  - It is not available at the end of \( i \), and
  - either it cannot move to the head of \( i \) (KilledEXP(i))
  - or another edge leaving \( i \) prevents its placement in \( k \) (AVAILOUT(i))

KT2, 2007

---

**Later (than earliest) placement**

\[
\text{LATER}(i, j) = \text{NF}(i) \land \text{LATER}(i, j) \land \text{EARLIEST}(i, j)
\]

\[
\text{LATER}(i, j) = \text{NF}(i) \land \text{LATER}(i, j)
\]

**Initialization** \( \text{LATER}(i, 0) = \text{NF}(i) \land \text{LATER}(i, 0) \land \text{EARLIEST}(i, 0) \land \text{UEXP}(i) \)

\( x = \text{LATER}(i, j) \) if \( i \) is its earliest placement, or it can be moved forward from \( i \) (LATER(i,j)) and placement at entry to \( i \) does not anticipate a use in \( j \) (moving it across the edge exposure that used it)

KT2, 2007

---

Lazy Code Motion

**Rewriting the code**

\[
\text{INSERT}(i, j) = \text{LATER}(i, j) \land \text{LATER}(i, j)
\]

\[
\text{DELETE}(k) = \text{UEXPI}(k) \land \text{LATER}(i, j), k = n_i
\]

\( \text{INSERT} \) and \( \text{DELETE} \) are predicates

Compiler uses them to guide the rewrite step

- \( x = \text{INSERT}(i, j) \) \( \Rightarrow \) insert \( x \) at the end of \( i \), beginning of \( j \), or new block
- \( x = \text{DELETE}(k) \) \( \Rightarrow \) delete first evaluation of \( x \) in \( k \)

**Example**

If local redundancy elimination has already been performed, only one copy of \( x \) exists. Otherwise, remove all upward exposed copies of \( x \).

KT2, 2007

---

Lazy Code Motion Example

**Example**

If \( i \) then \( B_2 \) else \( B_3 \)

\[
\text{INSERT}(i, 1) = \{ f_a, f_{x_1} \}
\]

**Example after Rewrite**

 KT2, 2007