Lazy Code Motion

COMP 512
Rice University
Houston, Texas

Spring 2009

“Lazy Code Motion,” J. Knoop, O. Ruthing, & B. Steffen, in PLDI 92
“A Variation of Knoop, Ruthing, and Steffen’s Lazy Code Motion,”
K. Drechsler & M. Stadel, SIGPLAN Notices, 28(5), May 1993
Treatment in Chapter 10 of Engineering a Compiler …

Redundant Expression

An expression is redundant at 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

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

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

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

\[
\begin{align*}
b &\leftarrow b + 1 \\
a &\leftarrow b + c
\end{align*}
\]
**Partially Redundant Expression**

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

Example

\[
\begin{align*}
  &\quad \quad b \leftarrow b + 1 \\
  &\quad \quad a \leftarrow b + c \\
  &\quad \quad a \leftarrow b + c \\
\end{align*}
\]

Inserting a copy of “\( a \leftarrow b + c \)” after the definition of \( b \) can make it redundant

**Loop Invariant Expression**

Another example

\[
\begin{align*}
  &\quad \quad x \leftarrow y \ast z \\
  &\quad \quad a \leftarrow b \ast c \\
\end{align*}
\]

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 show opportunities & limits
- Compute INSERT & DELETE sets from solutions
- Linear pass over the code to rewrite it (using INSERT & DELETE)

The history

- Partial redundancy elimination (Morel & Renvoise, CACM, 1979)
- Improvements by Drechsler & Stadel, Joshi & Dhamdhere, Chow, Knoop, Rthing & Steffen, Dhamdhere, Sorkin, …
- All versions of PRE optimize placement
  > Guarantee that no path is lengthened
- LCM was invented by Knoop et al. in PLDI, 1992
- We will look at a variation by Drechsler & Stadel
  > SIGPLAN Notices 28(5), May 1993

Lazy Code Motion

The intuitions

- Compute available expressions
- Compute anticipable expressions
- From AVAIL & Ant, we can compute an earliest placement for each expression
- Push expressions down the CFG until it changes behavior

Assumptions

- Uses a lexical notion of identity (not value identity)
- ILOC-style code with unlimited name space
- Consistent, disciplined use of names
  > Identical expressions define the same name
  > No other expression defines that name

LCM operates on expressions
It moves expression evaluations, not assignments
Avoids copies
Result serves as proxy
Lazy Code Motion

The Name Space

- \( r_i + r_j \rightarrow r_k \), always, with both \( i < k \) and \( j < k \) (hash to find \( k \))
- We can refer to \( r_i + r_j \) by \( r_k \) (bit-vector sets)
- Variables must be set by copies
  - No consistent definition for a variable
  - Break the rule for this case, but require \( r_{\text{source}} < r_{\text{destination}} \)
  - 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

Digression in Chapter 5 of EAC: “The impact of naming”

Lazy Code Motion

Local Predicates

- \( \text{DEExpr}(b) \) contains expressions defined in \( b \) that survive to the end of \( b \) (downward exposed expressions)
  \[ e \in \text{DEExpr}(b) \Rightarrow \text{evaluating } e \text{ at the end of } b \text{ produces the same value for } e \]
- \( \text{UEExpr}(b) \) contains expressions defined in \( b \) that have upward exposed arguments (both args) (upward exposed expressions)
  \[ e \in \text{UEExpr}(b) \Rightarrow \text{evaluating } e \text{ at the start of } b \text{ produces the same value for } e \]
- \( \text{ExprKill}(b) \) contains those expressions that have one or more arguments defined (killed) in \( b \) (killed expressions)
  \[ e \notin \text{ExprKill}(b) \Rightarrow \text{evaluating } e \text{ produces the same result at the start and end of } b \]

We have seen all three of these previously.
Lazy Code Motion

Availability

\[ \text{AVAILIN}(n) = \bigcap_{m \in \text{preds}(n)} \text{AVAILOUT}(m), \quad n \neq n_0 \]

\[ \text{AVAILOUT}(m) = \text{DEEXPR}(m) \cup (\text{AVAILIN}(m) \cap \overline{\text{EXPRKILL}(m)}) \]

Initialize \text{AVAILIN}(n) \text{ to the set of all names, except at } n_0

Set \text{AVAILIN}(n_0) \text{ to } \emptyset

Interpreting \text{AVAIL}

- \( e \in \text{AVAILOUT}(b) \iff \text{evaluating } e \text{ at end of } b \text{ produces the same value for } e. \text{ AVAILOUT tells the compiler how far forward } e \text{ can move }

- This differs from the way we call \text{AVAIL} in global redundancy elimination; the equations, however, are unchanged.

Lazy Code Motion

Anticipability

\[ \text{ANTOUT}(n) = \bigcap_{m \in \text{succs}(n)} \text{ANTIN}(m), \quad n \text{ not an exit block} \]

\[ \text{ANTIN}(m) = \text{UEEXPR}(m) \cup (\text{ANTOUT}(m) \cap \overline{\text{EXPRKILL}(m)}) \]

Initialize \text{ANTOUT}(n) \text{ to the set of all names, except at exit blocks}

Set \text{ANTOUT}(n) \text{ to } \emptyset, \text{ for each exit block } n

Interpreting \text{ANTOUT}

- \( e \in \text{ANTIN}(b) \iff \text{evaluating } e \text{ at start of } b \text{ produces the same value for } e. \text{ ANTIN tells the compiler how far backward } e \text{ can move }

- This view shows that anticipability is, in some sense, the inverse of availability (and explains the new interpretation of \text{AVAIL})
Lazy Code Motion

The intuitions

Available expressions

- $e \in \text{AVAILOUT}(b) \Rightarrow$ evaluating $e$ at exit of $b$ gives same result
- $e \in \text{AVAILIN}(b) \Rightarrow e$ is available from every predecessor of $b$
  $\Rightarrow$ an evaluation at entry of $b$ is redundant

Anticipable expressions

- $e \in \text{ANTIN}(b) \Rightarrow$ evaluating $e$ at entry of $b$ gives same result
- $e \in \text{ANTOUT}(b) \Rightarrow e$ is anticipable from every successor of $b$
  $\Rightarrow$ evaluation at exit of $b$ would a later evaluation redundant,
  on every path, so exit of $b$ is a profitable place to insert $e$

Lazy Code Motion

Earliest placement on an edge

$\text{EARLIEST}(i,j) = \text{ANTIN}(j) \cap \text{AVAILOUT}(i) \cap (\text{EXPRKILL}(i) \cup \text{ANTOUT}(i))$

$\text{EARLIEST}(n_0,j) = \text{ANTIN}(j) \cap \text{AVAILOUT}(n_0)$

$\Rightarrow$ insert $e$ on the edge

Earliest is a predicate

- Computed for edges rather than nodes (placement)
- If $e \in \text{EARLIEST}(i,j)$
  - It can move to head of $j$, $\text{ANTIN}(j)$
  - It is not available at the end of i and $\text{EXPRKILL}(i)$
  - either it cannot move to the head of $i$ or another edge leaving $i$ prevents its placement in $i$ $\text{ANTOUT}(i)$
Lazy Code Motion

Later (than earliest) placement

\[ \text{LATERIN}(j) = \bigcap_{i \in \text{pred}(j)} \text{LATER}(i,j), \quad j \neq n_0 \]

\[ \text{LATER}(i,j) = \text{EARLIEST}(i,j) \cup (\text{LATERIN}(i) \cap \text{UEEXPR}(i)) \]

Initialize \( \text{LATERIN}(n_0) \) to \( \emptyset \)

\( x \in \text{LATERIN}(k) \iff \) every path that reaches \( k \) has \( x \in \text{EARLIEST}(i,j) \) for some edge \((i,j)\) leading to \( x \), and the path from the entry of \( j \) to \( k \) is \( x \)-clear & does not evaluate \( x \)

\( \Rightarrow \) the compiler can move \( x \) through \( k \) without losing any benefit

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

Lazy Code Motion

Rewriting the code

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

\[ \text{DELETE}(k) = \text{UEEXPR}(k) \cap \text{LATERIN}(k), \quad k \neq n_0 \]

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

Compiler uses them to guide the rewrite step

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

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

Edge placement

- \( x \in \text{INSERT}(i,j) \)

Three cases

- \(|\text{succs}(i)| = 1 \Rightarrow \text{insert at end of } i\)
- \(|\text{succs}(i)| > 1, \text{ but } |\text{preds}(j)| = 1 \Rightarrow \text{insert at start of } j\)
- \(|\text{succs}(i)| > 1, \text{ and } |\text{preds}(j)| > 1 \Rightarrow \text{create new block in } <i,j> \text{ for } x\)

Lazy Code Motion

Example

\[ B_1: \]
- \( r_1 \leftarrow 1 \)
- \( r_2 \leftarrow r_0 + @m \)
  if \( r_1 < r_2 \rightarrow B_2,B_3 \)

\[ B_2: \]
- \( r_20 \leftarrow r_{17} * r_{18} \)
- \( r_4 \leftarrow r_1 + 1 \)
- \( r_1 \leftarrow r_4 \)
  if \( r_1 < r_2 \rightarrow B_2,B_3 \)

\[ B_3: \]

| Critical edge rule will create landing pad when needed, as on edge \((B_1,B_2)\) |

Example is too small to show off Later

Insert(1,2) = \{ r_{20} \}
Delete(2) = \{ r_{20} \}