Lazy Code Motion

— The Data-Flow Approach to Code Motion —

§ 10.3.1 of EaC2e

Copyright 2015, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved

Citation numbers refer to entries in the EaC2e bibliography.
Announcements

• Next week is break

• Midterm exam
  ♦ Available today, in class
  ♦ Due back on Tuesday 3/10/2015 at 5 PM
  ♦ Three questions:
    → Matching question through today’s lecture
    → Question on data-flow analysis
    → Question on the construction of SSA form
  ♦ Two-hours, closed-notes, closed-literature, take-home exam

• You should be working on your labs
  ♦ Three benchmarks available on CLEAR
  ♦ More to come
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

```
a ← b + c
b ← b + 1
a ← b + c
a ← b + c
a ← b + c
```
Partially Redundant Expression

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

**Example**

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

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

**fully redundant?**
Loop Invariant Expression

Another Example

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

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
  ♦ Availability & anticipability, then placement
• Compute INSERT & DELETE sets from solutions
• Linear pass over the code to rewrite it (using INSERT & DELETE)

The History

• Partial redundancy elimination (PRE) [267] (Morel & Renvoie, CACM, 1979)
• Improvements by Drechsler & Stadel [133], Joshi & Dhamdhere [209], Chow [81], Knoop, Ruthing & Steffen [225], Dhamdhere [130], Sorkin [321], Hailperin [178], Kennedy, Lo, et al. [220] ...
• All versions of PRE optimize placement
  ♦ Guarantee that no path is lengthened
• LCM was published by Knoop et al. in PLDI 92
• Drechsler & Stadel simplified the equations

Dhamdhere applied these same ideas to strength reduction [127, 131] and hoisting [129]. Others have followed this path, as well [209, 220, 226].
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 code that is *not* in SSA form. Lexical identity conflicts with SSA’s notion of unique names.
Lazy Code Motion

The Name Space

• \( r_i + r_j \rightarrow r_k \), always, with both \( i < k \) and \( j < k \)
  
  ♦ \( r_k \) is always set by \( r_i + r_j \) or \( r_j + r_i \), and by no other expression
  
• We can refer to \( r_i + r_j \) by \( r_k \)

• Variables must be set by copies
  
  ♦ No consistent definition for a variable
  
  ♦ Break the rule for this case, but require \( r_{source} < r_{destination} \)
  
  ♦ To achieve this, assign register names to variables first

Without this name space

• \textbf{LCM} must insert copies to preserve redundant values

• \textbf{LCM} must compute its own map of expressions to unique ids

The restrictions on the name space in \textbf{LCM} goes all the way back to Morel & Renvoise [267]. It is mentioned as an assumption in the original paper.
Lazy Code Motion

Local Information

(Computed for each block)

- **DEEXPR(b)** contains expressions defined in b that survive to the end of b
  
  \[ e \in \text{DEEXPR}(b) \implies \text{evaluating } e \text{ at the end of } b \text{ produces the same value for } e \]

- **UEEXPR(b)** contains expressions defined in b that have upward exposed arguments (both args)
  
  \[ e \in \text{UEEXPR}(b) \implies \text{evaluating } e \text{ at the start of } b \text{ produces the same value for } e \]

- **EXPRKILL(b)** contains those expressions that have one or more arguments defined (killed) in b
  
  \[ e \not\in \text{EXPRKILL}(b) \implies \text{evaluating } e \text{ produces the same result at the start and end of } b \]
Lazy Code Motion

Availability

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

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

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

Interpreting \text{AVAILOUT}

- \( e \in \text{AVAILOUT}(b) \iff \) evaluating \( e \) at end of \( b \) produces the same value for \( e \). \text{AVAILOUT} tells the compiler how far forward \( e \) can move
- This interpretation differs from the way we 
  talk about \text{AVAILOUT} in global redundancy elimination; the equations, however, are unchanged.
Lazy Code Motion

Anticipability

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

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

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 \(\text{ANTOUT}\)

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

• This view shows that anticipability is, in some sense, the inverse of availability (& 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

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

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

\( \text{EARLIEST} \) is a predicate

• Computed for edges rather than nodes

• \( e \in \text{EARLIEST}(i,j) \) 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 \( i \) or another edge leaving \( i \) prevents its placement in \( i \)

Can move \( e \) to head of \( j \) & it is not redundant from \( i \) and

Either killed in \( i \) or would not be busy at exit of \( i \)

\( \Rightarrow \) insert \( e \) on the edge
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 \(i\)

\(\text{(moving it across the edge exposes that use)}\)
Lazy Code Motion

Rewriting the code

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

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

\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 \( k \)

1 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

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

Example

$B_1$: $r_1 \leftarrow 1$
$r_2 \leftarrow r_0 + @m$
if $r_1 < r_2 \rightarrow B_2, B_3$

$B_2$: ...
$r_{20} \leftarrow r_{17} \times r_{18}$
...
$r_4 \leftarrow r_1 + 1$
$r_1 \leftarrow r_4$
if $r_1 < r_2 \rightarrow B_2, B_3$

$B_3$: ...

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

<table>
<thead>
<tr>
<th>B1</th>
<th>B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEEXPR</td>
<td>r1,r2</td>
</tr>
<tr>
<td>UEEXPR</td>
<td>r1,r2</td>
</tr>
<tr>
<td>NotKilled</td>
<td>r17,r18,r20</td>
</tr>
<tr>
<td></td>
<td>r1,r4,r20</td>
</tr>
<tr>
<td></td>
<td>r4,r20</td>
</tr>
<tr>
<td></td>
<td>r2,r17,r18,r20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B1</th>
<th>B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVAILIN</td>
<td>r17,r18</td>
</tr>
<tr>
<td>AVAILOUT</td>
<td>r1,r2,r17,r18</td>
</tr>
<tr>
<td>ANTIN</td>
<td>{}</td>
</tr>
<tr>
<td>ANTOUT</td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td>r20</td>
</tr>
<tr>
<td></td>
<td>r1,r2,r4,r17,r18,r20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B1</th>
<th>B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EARLIEST</td>
<td>r20</td>
</tr>
<tr>
<td></td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td>{}</td>
</tr>
</tbody>
</table>

Example is too small to show off LATER

$\text{INSERT}(1,2) = \{ r_{20} \}$
$\text{DELETE}(2) = \{ r_{20} \}$

See the papers for more detailed examples.
Lazy Code Motion

Improving the Results

Simpson attacked the problem of LCM’s reliance on lexical identity

• Performed global value numbering, then rewrote the name space to encode value identity into lexical identity

• In essence, his technique joined the code placement aspects of LCM with the value-based equivalence detection of global value numbering

Briggs rearranged expressions to expose more lexical identities

• Used algebraic reassociation to rewrite expressions into a canonical form
  ♦ Associativity & commutativity, + distribution in some limited forms

• Preconditioning the code with reassociation exposed more opportunities