Lazy Code Motion


§ 10.3.1 of EaC2e

Copyright 2011, 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.
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

Some occurrences of $b+c$ are redundant
**Partially Redundant Expression**

An expression is **partially redundant** at $p$ if it is redundant along some, but not all, paths reaching $p$.

**Example**

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

Inserting a copy of “$a \leftarrow b + c$” after the definition of $b$ can make it redundant?
Loop Invariant Expression

Another example

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
- 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, Ruthing & 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
- Drechsler & Stadel simplified the equations

PRE and its descendants are conservative
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 \( r_{source} > r_{destination} \)
  - 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

---

**LCM operates on expressions**

It moves expression evaluations, not assignments
Lazy Code Motion

Local Information  

(Computed for each block)

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

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

- **EXPRKILL(b)** contains those expressions that have one or more arguments defined (killed) in b  
  \( e \notin \text{EXPRKILL}(b) \Rightarrow \text{evaluating } e \text{ produces the same result at the start and end of } b \)
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 \( \text{AVAIL} \)

- \( e \in \text{AVAILOUT}(b) \iff \) evaluating \( e \) at end of \( b \) produces the same value for \( e \). \( \text{AVAILOUT} \) tells the compiler that an evaluation at the end of the block is covered by the evaluation earlier in the block. It also shows that evaluation of \( e \) can move to the end of the block.
- This interpretation differs from the way we talk about \( \text{AVAIL} \) in global redundancy elimination; the equations, however, are unchanged.
Lazy Code Motion

Anticipability

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

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

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 \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. If } e \text{ is also in AVAILIN}(b), \text{ the evaluation in the block is redundant.}
- \text{This view shows that anticipability is, in some sense, the inverse of availability (\& explains the new interpretation of AVAIL)}

Anticipability is identical to VeryBusy expressions
Lazy Code Motion

The intuitions

Available expressions

• \( e \in \text{AVAILOUT}(b) \) \( \Rightarrow \) evaluating \( e \) at exit of \( b \) gives same result
  \( \Rightarrow \) \( e \) could move to exit of \( b \)

• \( e \in \text{AVAILIn}(b) \) \( \Rightarrow \) \( e \) is available from every predecessor of \( b \)
  \( \Rightarrow \) an evaluation of \( e \) 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 used on every path leaving \( b \)
  \( \Rightarrow \) evaluations in \( b \)’s successors could move to the end of \( b \)
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

Can move $e$ to head of $j$ &
it is not redundant from $i$

Either killed in $i$ or would
not be busy at exit of $i$

$\Rightarrow$ insert $e$ on the edge

$\text{EARLIEST}$ 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 $i$
Lazy Code Motion

Later (than earliest) placement

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

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

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

\( x \in \text{LATER_IN}(k) \iff \) 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 \( i \) (\textit{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 \( k \)

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

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

A “critical” edge
Lazy Code Motion

Example from Knoop et al.

Original Code

Assume that bad things can happen in an empty box
Lazy Code Motion

Example from Knoop et al.

After LCM

1. 
2. a ← c
3. x ← a + b
4. 
5. 
6. 
7. 
8. h ← a + b
9. 
10. y ← h
11. 
12. 
13. 
14. 
15. h ← a + b
   y ← h
16. z ← h
17. x ← a + b
18. 

After LCM
Lazy Code Motion

Example

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

\[ B_2: \quad \ldots \]
\[ r_{20} \leftarrow r_{17} \times r_{18} \]
\[ \ldots \]
\[ r_4 \leftarrow r_1 + 1 \]
\[ r_1 \leftarrow r_4 \]
if \( r_1 < r_2 \rightarrow B_2, B_3 \]

\[ B_3: \quad \ldots \]

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

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEEXPR</td>
<td>(r_1, r_2)</td>
<td>(r_1, r_4, r_{20})</td>
</tr>
<tr>
<td>UEEXPR</td>
<td>(r_1, r_2)</td>
<td>(r_4, r_{20})</td>
</tr>
<tr>
<td>NotKilled</td>
<td>(r_{17}, r_{18}, r_{20})</td>
<td>(r_2, r_{17}, r_{18}, r_{20})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AvailIn</td>
<td>(r_{17}, r_{18})</td>
<td>(r_{1}, r_{2}, r_{17}, r_{18})</td>
</tr>
<tr>
<td>AvailOut</td>
<td>(r_1, r_2, r_{17}, r_{18})</td>
<td>(r_1, r_2, r_4, r_{17}, r_{18}, r_{20})</td>
</tr>
<tr>
<td>AntIn</td>
<td>{}</td>
<td>(r_{20})</td>
</tr>
<tr>
<td>AntOut</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1,2</th>
<th>1,3</th>
<th>2,2</th>
<th>2,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Earliest</td>
<td>(r_{20})</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

Example is too small to show off Later

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