Partial Redundancy Elimination, Loop Optimization & Lazy Code Motion

Advanced Compiler Techniques
2005
Erik Stenman
Virtutech
Common-Subexpression Elimination
(Repetition)

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 if 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.

Before CSE

\[
\begin{align*}
... & \\
a &= (x+y) + z; \\
b &= a - 1; \\
c &= x + y; \\
... \\
\end{align*}
\]

After CSE

\[
\begin{align*}
... & \\
t &= x + y; \\
a &= t + z; \\
b &= a - 1; \\
c &= t; \\
... \\
\end{align*}
\]
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

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

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

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

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

Example

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

Another example:

```
x ← 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.
Loop Optimizations

♦ Loop Optimization is important because most of the execution time occurs in loops.
♦ First, we will identify loops.
♦ We will study three optimizations:
  ♦ Loop-invariant code motion.
  ♦ Strength reduction.
  ♦ Induction variable elimination.
♦ We will not talk about loop unrolling which is another important optimization technique.
What is a Loop?

- Set of nodes
- Loop header
  - Single node
  - All iterations of loop go through header
- Back edge
Anomalous Situations

♦ Two back edges, two loops, one header
♦ Compiler merges loops
♦ No loop header, no loop
Defining Loops With Dominators

Recall the concept of dominators:

- **Node** $n$ **dominates** a node $m$ if all paths from the start node to $m$ go through $n$.
- The **immediate dominator** of $m$ is the last dominator of $m$ on any path from start node.
- A **dominator tree** is a tree rooted at the start node:
  - Nodes are nodes of control flow graph.
  - Edge from $d$ to $n$ if $d$ is the immediate dominator of $n$. 
A loop has a unique entry point – the header.
At least one path back to header.
Find edges whose heads (>) dominate tails (-), these edges are back edges of loops.
Given a back edge $n \rightarrow d$:
- The node $d$ is the loop header.
- The loop consists of $n$ plus all nodes that can reach $n$ without going through $d$ (all nodes “between” $d$ and $n$)
Loop Construction Algorithm

\[
\text{loop}(d,n) \\
\text{loop} = \{d\}; \text{ stack } = \emptyset; \text{ insert(n)}; \\
\text{while stack not empty do} \\
\text{ \hspace{1cm} m = pop stack;} \\
\text{ \hspace{1cm} for all } p \in \text{ pred}(m) \text{ do insert(p);} \\
\text{insert(m)} \\
\text{ \hspace{1cm} if } m \notin \text{ loop} \text{ then} \\
\text{ \hspace{1cm} loop } = \text{ loop } \cup \{m\}; \\
\text{ \hspace{1cm} push m onto stack;} \\
\]
Nested Loops

♦ If two loops do not have same header then
  ♦ Either one loop (inner loop) is contained in the other (outer loop)
  ♦ Or the two loops are disjoint
♦ If two loops have same header, typically they are unioned and treated as one loop

Two loops:
{1,2} and {1, 3}
Unioned: {1,2,3}
Many optimizations stick code before loop.
Put a special node (loop preheader) before loop to hold this code.
Loop Optimizations

♦ Now that we have the loop, we can optimize it!
♦ Loop invariant code motion:
  ♦ Move loop invariant code to the header.
Loop Invariant Code Motion

If a computation produces the same value in every loop iteration, move it out of the loop.

```plaintext
for i = 1 to N
  x = x + 1
for j = 1 to N
  a[i,j] = 100*N + 10*i + j + x
```

```plaintext
t1 = 100*N
for i = 1 to N
  x = x + 1
  t2 = t1 + 10*i + x
for j = 1 to N
  a[i,j] = t2 + j
```
Detecting Loop Invariant Code

- A statement is *loop-invariant* if operands are
  - Constant,
  - Have all reaching definitions outside loop, or
  - Have exactly one reaching definition, and that definition comes from an invariant statement

- Concept of exit node of loop
  - node with successors outside loop
Loop Invariant Code Detection Algorithm

for all statements in loop
    if operands are constant or have all reaching definitions outside loop, mark statement as invariant
do
    for all statements in loop not already marked invariant
        if operands are constant, have all reaching definitions outside loop, or have exactly one reaching definition from invariant statement
            then mark statement as invariant
    until there are no more invariant statements
Loop Invariant Code Motion

- Conditions for moving a statement $s$: $x = y + z$ into loop header:
  - $s$ dominates all exit nodes of loop
    - If it does not, some use after loop might get wrong value
    - Alternate condition: definition of $x$ from $s$ reaches no use outside loop (but moving $s$ may increase run time)
  - No other statement in loop assigns to $x$
    - If one does, assignments might get reordered
  - No use of $x$ in loop is reached by definition other than $s$
    - If one is, movement may change value read by use
Order of Statements in Preheader

Preserve data dependences from original program (can use order in which discovered by algorithm)

```
b = 2
i = 0
i < 80
a = b * b
c = a + a
i = i + c
```

Loop Optimizations: Loop invariant code motion
Induction Variables

Example:

```c
for j = 1 to 100
    *(&A + 4*j) = 202 - 2*j
```

Basic Induction variable:

- \( j = 1, 2, 3, 4, \ldots \)

Induction variable \( \&A + 4*j \):

- \( \&A + 4*j = \&A + 4, \ &A + 8, \ &A + 12, \ &A + 16, \ldots \)
What are induction variables?

- x is an induction variable of a loop L if
  - variable changes its value every iteration of the loop
  - the value is a function of number of iterations of the loop

- In programs, this function is normally a linear function
  Example: for loop index variable j, function d + c*j
Types of Induction Variables

♦ Base induction variable:
  ♦ Only assignments in loop are of form $i = i \pm c$

♦ Derived induction variables:
  ♦ Value is a linear function of a base induction variable.
  ♦ Within loop, $j = c*i + d$, where $i$ is a base induction variable.
  ♦ Very common in array index expressions – an access to $a[i]$ produces code like $p = a + 4*i$. 
Strength Reduction for Derived Induction Variables

i = 0

i < 10

i = i + 1
p = 4 * i

use of p

i = 0
p = 0

i < 10

i = i + 1
p = p + 4

use of p
Elimination of Superfluous Induction Variables

```
i = 0
p = 0

i < 10

i = i + 1
p = p + 4
use of p

p = 0

p < 40

p = p + 4
use of p
```
Three Algorithms

♦ Detection of induction variables:
  ♦ Find base induction variables.
  ♦ Each base induction variable has a family of derived induction variables, each of which is a linear function of base induction variable.

♦ Strength reduction for derived induction variables.

♦ Elimination of superfluous induction variables.
Output of Induction Variable Detection Algorithm

♦ Set of induction variables:
  ♦ base induction variables.
  ♦ derived induction variables.
♦ For each induction variable $j$, a triple $<i,c,d>$:
  ♦ $i$ is a base induction variable.
  ♦ the value of $j$ is $i*c+d$.
  ♦ $j$ belongs to family of $i$. 
Induction Variable Detection Algorithm

Scan loop to find all base induction variables
do

Scan loop to find all variables \( k \) with one assignment of form \( k = j \times b \) where \( j \) is an induction variable with triple \( <i,c,d> \) (\( j = i \times c + d \), \( k = (i \times c + d) \times b = i \times c \times b + d \times b \))
make \( k \) an induction variable with triple \( <i,c \times b,d \times b> \)

Scan loop to find all variables \( k \) with one assignment of form \( k = j \pm b \) where \( j \) is an induction variable with triple \( <i,c,d> \) (\( j = i \times c + d \), \( k = (i \times c + d) \pm b = i \times c \times b + d \pm b \))
make \( k \) an induction variable with triple \( <i,c,b \pm d> \)

until no more induction variables are found
Strength Reduction

t = 202
for j = 1 to 100
    t = t - 2
    *(abase + 4*j) = t

Basic Induction variable:
J    = 1, 1, 2, 1, 3, 1, 4, ... 

Induction variable 202 - 2*j
0 = 202, 200, 198, 196, ... 

Induction variable abase+4*j:
abase+4*j = abase+4, abase+8, abase+12, abase+16, ...
Strength Reduction Algorithm

for all derived induction variables j with triple <i,c,d>

Create a new variable s

Replace assignment j = i*c+d with j = s

Immediately after each assignment i = i + e,
insert statement s = s + c*e (c*e is constant)
place s in family of i with triple <i,c,d>
Insert s = c*i+d into preheader
Strength Reduction for Derived Induction Variables

```
```

Loop Optimizations: Induction Variables - Strength Reduction

30
Example

double A[256], B[256][256]
j = 1

while (j < 100)
    A[j] = B[j][j]
    j = j + 2

double A[256], B[256][256]
j = 1
a = &A + 8
b = &B + 2056   // 2048+8

while (j < 100)
    *a = *b
    j = j + 2
    a = a + 16
    b = b + 4112   // 4096+16
Induction Variable Elimination

Choose a base induction variable $i$ such that only uses of $i$ are in termination condition of the form $i < n$
assignment of the form $i = i + m$

Choose a derived induction variable $k$ with $<i,c,d>$

Replace termination condition with $k < c*n+d$
Summary
Loop Optimization

♦ Important because lots of time is spent in loops.
♦ Detecting loops.
♦ Loop invariant code motion.
♦ Induction variable analyses and optimizations:
  ♦ Strength reduction.
  ♦ Induction variable elimination.
Lazy Code Motion

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

The history
♦ Partial redundancy elimination \hfill (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
♦ We will look at a variation by Drechsler \& Stadel

\textbf{SIGPLAN Notices, 28(5), May, 1993}
Lazy Code Motion

The intuitions
♦ Compute *available expressions*
♦ Compute *anticipable expressions*
♦ These lead to 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)*
♦ 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

\[\text{Avoids copies} \implies \text{Result serves as proxy}\]
Lazy Code Motion

The Name Space

- $r_i + r_j \rightarrow r_k$, always  
  $(hash\ to\ find\ k)$
- 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
Lazy Code Motion: Running Example

\[ B_1: \]
\[
\begin{align*}
r_1 & \leftarrow 1 \\
r_2 & \leftarrow r_1 \\
r_3 & \leftarrow r_0 + @m \\
r_4 & \leftarrow r_3 \\
r_5 & \leftarrow (r_1 < r_2) \\
\text{if } r_5 \text{ then } B_2 \text{ else } B_3
\end{align*}
\]

\[ B_2: \]
\[
\begin{align*}
r_{20} & \leftarrow r_{17} \times r_{18} \\
r_{21} & \leftarrow r_{19} + r_{20} \\
r_8 & \leftarrow r_{21} \\
r_6 & \leftarrow r_2 + 1 \\
r_2 & \leftarrow r_6 \\
r_7 & \leftarrow (r_2 > r_4) \\
\text{if } r_7 \text{ then } B_3 \text{ else } B_2
\end{align*}
\]

\[ B_3: \ldots \]

Variables:
\[ r_2, r_4, r_8 \]

Expressions:
\[ r_1, r_3, r_5, r_6, r_7, r_{20}, r_{21} \]
Lazy Code Motion

Predicates (computed by Local Analysis)

♦ $\text{DEExpr}(b)$ contains expressions defined in $b$ that survive to the end of $b$.
  \[ e \in \text{DEExpr}(b) \Rightarrow \text{evaluating } e \text{ at the end of } b \text{ produces the same value for } e \text{ as evaluating it in its original position.} \]

♦ $\text{UEExpr}(b)$ contains expressions defined in $b$ that have upward exposed arguments (both args).
  \[ e \in \text{UEExpr}(b) \Rightarrow \text{evaluating } e \text{ at the start of } b \text{ produces the same value for } e \text{ as evaluating it in its original position.} \]

♦ $\text{KilledExpr}(b)$ contains those expressions whose arguments are (re)defined in $b$.
  \[ e \in \text{KilledExpr}(b) \Rightarrow \text{evaluating } e \text{ at the start of } b \text{ does not produce the same result as evaluating it at its end.} \]
Lazy Code Motion: Running Example

\[ B_1:\]
\[
\begin{align*}
  r_1 & \leftarrow 1 \\
  r_2 & \leftarrow r_1 \\
  r_3 & \leftarrow r_0 + @m \\
  r_4 & \leftarrow r_3 \\
  r_5 & \leftarrow (r_1 < r_2) \\
  \text{if } r_5 \text{ then } B_2 \text{ else } B_3
\end{align*}
\]

\[ B_2:\]
\[
\begin{align*}
  r_{20} & \leftarrow r_{17} \times r_{18} \\
  r_{21} & \leftarrow r_{19} + r_{20} \\
  r_8 & \leftarrow r_{21} \\
  r_6 & \leftarrow r_2 + 1 \\
  r_2 & \leftarrow r_6 \\
  r_7 & \leftarrow (r_2 > r_4) \\
  \text{if } r_7 \text{ then } B_3 \text{ else } B_2
\end{align*}
\]

\[ B_3:\]
\[
\ldots
\]

Variables:
- \( r_2, r_4, r_8 \)

Expressions:
- \( r_1, r_3, r_5, r_6, r_7, r_{20}, r_{21} \)

<table>
<thead>
<tr>
<th>DEExpr</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1, r3, r5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r7, r20, r21</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UEEExpr</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1, r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r6, r20</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KILLEDExpr</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>r5, r6, r7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r5, r6, r7, r21</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lazy Code Motion

Availability

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

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

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 how far forward \( e \) can move the
evaluation of \( e \), ignoring any uses of \( e \).

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

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

This differs from the way we talk about \text{AVAIL} in global redundancy
elimination.
Lazy Code Motion

Anticipability

\[ \text{AntOut}(n) = \bigcap_{m \in \text{succs}(n)} \text{AntIn}(m), \text{ } n \text{ not an exit block} \]

\[ \text{AntIn}(m) = \text{UEExpr}(m) \cup (\text{AntOut}(m) \cap \text{KilledExpr}(m)) \]

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.

\( \triangleright \) This view shows that anticipability is, in some sense, the inverse of availability (& explains the new interpretation of \text{Avail}).
Lazy Code Motion

Earliest placement

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

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

\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 \( i \) (\text{KILLEDEXPR}(i))
    - or another edge leaving \( i \) prevents its placement in \( i \) (\text{ANTOUT}(i))
Lazy Code Motion

Later (than earliest) placement

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

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

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

\( x \in \text{LATERIN}(k) \iff \) every path that reaches \( k \) has \( x \in \text{EARLIEST}(m) \) for some block \( m \), and the path from \( m \) to \( k \) is \( x \)-clear & does not evaluate \( x \).

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

\( x \in \text{LATER}(i,j) \iff \langle i,j \rangle \) 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

\[ \text{INSERT}(i,j) = \text{LATER}(i,j) \cap \text{LATERIN}(j) \]
\[ \text{DELETE}(k) = \text{UEExpr}(k) \cap \text{LATERIN}(k), \ k \neq n_0 \]

\textbf{INSERT} & \textbf{DELETE} are predicates

Compiler uses them to guide the rewrite step

\[ \diamond \ x \in \text{INSERT}(i,j) \Rightarrow \text{insert } x \text{ at start of } i, \text{ end of } j, \text{ or new block} \]

\[ \diamond \ x \in \text{DELETE}(k) \Rightarrow \text{delete first evaluation of } x \text{ in } k \]

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

B₁: \(r_1 \leftarrow 1\)
    \(r_2 \leftarrow r_1\)
    \(r_3 \leftarrow r_0 + \text{@m}\)
    \(r_4 \leftarrow r_3\)
    \(r_5 \leftarrow (r_1 < r_2)\)
    if \(r_5\) then \(B_2\) else \(B_3\)

B₂: \(r_{20} \leftarrow r_{17} \times r_{18}\)
    \(r_{21} \leftarrow r_{19} + r_{20}\)
    \(r_8 \leftarrow r_{21}\)
    \(r_6 \leftarrow r_2 + 1\)
    \(r_2 \leftarrow r_6\)
    \(r_7 \leftarrow (r_2 > r_4)\)
    if \(r_7\) then \(B_3\) else \(B_2\)

B₃: ...

---

Example is too small to show off **LATER**

**INSERT**\((1,2) = \{ r_{20}, r_{21} \}**

**DELETE**\((2) = \{ r_{20}, r_{21} \}**