Loop Invariant Code Motion

— A Simple Classical Approach —
Background

Code in loops executes more often than code outside loops

• An expression is *loop invariant* if it computes the same value, independent of the iteration in which it is computed
• Loop invariant code can often be moved to a spot before the loop
  ♦ Execute once and reuse the value multiple times
  ♦ Reduce the execution cost of the loop

Techniques seem to fall into two categories

• Ad-hoc graph-based algorithms
• Complete data-flow solutions
• Think of loop invariant as redundant across iterations
• Always lengthens live range
Loop-invariant Code Motion

Example
• x + y is invariant in the loop
  ♦ Computed on each iteration
  ♦ Subexpressions don’t change
• Move evaluation out of loop
  ♦ Need block to hold it
  ♦ Use existing block or insert new

Relationship to redundancy
• x + y is redundant along back edge
• x + y is not redundant from loop entry
• If we add an evaluation on the entry edge, x + y in the loop is redundant

Neither x nor y change in loop
Loop-invariant Code Motion

Neither $x$ nor $y$ change in loop

Option 1
Move $x + y$ into predecessor

Option 2
Create a block for $x + y$

What’s the difference?
In practice, many loops have a **zero-trip** test

- Determines whether to run the first iteration
- Moving $x + y$ above zero-trip test is **speculative**
  - Lengthens path around the loop
  - LICM techniques often create a **landing pad**
Another difference between methods
• Some methods move expression evaluation, but not assignment
  ♦ Easier safety conditions, easier to manage transformation
  ♦ Leaves a copy operation in the loop  
    \((may\ coalesce)\)
• Other methods move the entire assignment
  ♦ Eliminates copy from loop, as well
Loop-invariant Code Motion

Control flow

• Another source of speculation
  ◦ Move it & lengthen other path
  ◦ Don’t move it
    
    while (...) 
    {
      if (y != 0)
        then z = x/y
        else z = x
    }

• Divergence may be an issue

• Don’t want to move the op if doing so introduces an exception

• If that path is hot, then y != 0 and we might move it.

Would like to use branch probabilities

Note that y is invariant in this classic example, so we could move the test, as well. See Cytron, Lowry, & Zadeck.
Loop-invariant Code Motion

Pedagogical Plan
1. A simple and direct algorithm (*today*)
2. Lazy code motion, a data-flow solution based on availability
3. Cytron, Lowry, & Zadeck’s SSA-based approach *(probably not)*

Which is best?
• The authors of 2 & 3 would each argue for their technique
• In practice, each has strengths and weaknesses
• Taught 3 last year and had someone implement it in **LLVM**
  ♦ Surprising set of complications in **SSA** implementation
    → Moving around all of those definitions proved problematic
    → Creating new names, recreating **SSA** name space ... too much work for the benefits
  ♦ Cannot recommend **CLZ** in practice
Loop Shape Is Important

Loops
• Evaluate condition before loop (if needed)
• Evaluate condition after loop
• Branch back to the top (if needed)
Merges test with last block of loop body

COMP 412 teaches this loop shape. It creates a natural place to insert a landing pad.

*while, for, do, & until* all fit this basic model

For tail recursion, unroll the recursion once ...

Paper by Mueller & Whalley in SIGPLAN ‘92 PLDI about converting arbitrary loops to this form.
Loop-invariant Code Motion

Preliminaries

• Need a list of loops, ordered inner to outer in each loop nest
  ♦ Define “natural loop” as one formed by a back edge with respect to \( \text{DOM} \)
    → An edge \((t,h)\) is a back edge if \( h \text{ DOM } t \)
    → \( h \) is the loop’s header and \( t \) is a tail of the loop
    → The loop’s body includes all predecessors back to \( h \)
  ♦ Natural loops can nest
    → \( \text{Given two headers, } h_1 \text{ and } h_2, \text{ the loops are nested if } h_1 \text{ DOM } h_2 \text{ or } h_2 \text{ DOM } h_1 \)
    → If \( h_1 \) & \( h_2 \) are unrelated by \( \text{DOM} \), loops are not nested
    → Inner to outer order is defined by \( \text{DOM} \)
• Need a landing pad on each loop
  ♦ Insert above loop header & redirect inbound edges

\[ \text{Find the loop body formed by a back edge } (t,h) \]

Given back edge \((h,t)\)

\[
\begin{align*}
\text{body} & \leftarrow \{ h \} \\
\text{stack} & \leftarrow \text{empty} \\
\text{push}(t) \\
\text{while (stack is nonempty)} \{ \\
\text{ } n & \leftarrow \text{pop}() \\
\text{if } n \notin \text{body} \text{ then } \{ \\
\text{ } \text{body} & \leftarrow \text{body } \cup \{ n \} \\
\text{ } \text{for each } x \in \text{pred}(n) \leftarrow \{ x \} \\
\text{ } \text{push}(x) \\
\} \\
\end{align*}
\]
Loop-invariant Code Motion

A Simple Algorithm

for each $l$ in $\text{LOOPLIST}$ do
FactorInvariants($l$)

FactorInvariants($\text{loop}$)
MarkInvariants($\text{loop}$)
for each expr $e \in \text{loop}$ $(x \leftarrow y + z)$
if $x$ is marked $\text{invariant}$ then
begin
allocate a new name $t$
replace $o$ with $x \leftarrow t$
insert $t \leftarrow e$ in landing pad
for $\text{loop}$
end

MarkInvariants($\text{loop}$)

$\text{LOOPDEF} \leftarrow \bigcup_{\text{block } b \in \text{loop}} \text{DEF}(b)$
for each op $o \in \text{loop}$ $(x \leftarrow y + z)$
mark $x$ as $\text{invariant}$
if $y \in \text{LOOPDEF}$
then mark $x$ as $\text{variant}$
if $z \in \text{LOOPDEF}$
then mark $x$ as $\text{variant}$
Example

Consider the following simple loop nest

<table>
<thead>
<tr>
<th>do i ← 1 to 100</th>
<th>(i)</th>
<th>(j)</th>
<th>(k)</th>
<th>Dimensions addressed</th>
<th>Multiplies</th>
</tr>
</thead>
<tbody>
<tr>
<td>do j ← 1 to 100</td>
<td></td>
<td></td>
<td></td>
<td>3,000,000</td>
<td>2,000,000</td>
</tr>
<tr>
<td>do k ← 1 to 100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a(i,j,k) ← i \ast j \ast k)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Original Code*
Example

Consider the following simple loop nest

\[
\begin{align*}
\text{do } i &\leftarrow 1 \text{ to } 100 \\
\text{do } j &\leftarrow 1 \text{ to } 100 \\
& \quad t_1 \leftarrow \text{addr}(a(i,j)) \\
& \quad t_2 \leftarrow i \times j \\
& \quad \text{do } k \leftarrow 1 \text{ to } 100 \\
& \quad \quad t_1(k) \leftarrow t_2 \times k \\
& \quad \end{align*}
\]

Dimensions addressed | Multiplies
\[
\begin{array}{cc}
20,000 & 10,000 \\
1,000,000 & 1,000,000 \\
\end{array}
\]

After LICM on the Inner Loop
Example

Consider the following simple loop nest

\[
\begin{align*}
&\text{do } i \leftarrow 1 \text{ to } 100 \\
&\quad t_3 \leftarrow \text{addr}(a(i)) \\
&\text{do } j \leftarrow 1 \text{ to } 100 \\
&\quad t_1 \leftarrow \text{addr}(t_3(j)) \\
&\quad t_2 \leftarrow i \times j \\
&\text{do } k \leftarrow 1 \text{ to } 100 \\
&\quad t_1(k) \leftarrow t_2 \times k
\end{align*}
\]

<table>
<thead>
<tr>
<th>Dimensions addressed</th>
<th>Multiplies</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10,000</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
</tr>
</tbody>
</table>

After Doing Middle Loop
Safety of Loop-invariant Code Motion

What happens if we move an expression that generates an exception?

• Maybe the fault happens at a different time in execution
• Maybe the original code would not have faulted

Ideally, the compiler should delay the fault until it would occur

Options

• Mask fault & add a test to loop body
• Replace value with one that causes fault when used
• Block read access to the faulted value
• Patch the executable with a bad opcode
• Generate two copies of the loop and rerun with original code
• Never move an evaluation that can fault

Applies to all the algorithms
Profitability of Loop-invariant Code Motion

- Does the loop body always execute?
  - Placement of the landing pad is crucial

- Lengthen any paths?
  - Conservative code motion: not a problem
  - Speculative code motion: might be an issue
    → Rely on estimates of branch frequency?

- Register pressure?
  - Target of moved operation has longer live range
  - Might shorten live ranges of operands of moved op

Backus’ dilemma

Applies to all the algorithms
Shortcomings of the Simple LICM Algorithm

- Moves code out of conditionals in a speculative fashion
  - Ignores control flow inside the loop
  - Can imagine a more sophisticated approach based on control-dependence
- Moves only evaluation, not assignment
  - To move assignments would require a more complex approach to naming both arguments and results (such as the SSA name space?)
- Only finds first order invariants

```plaintext
for i ← 1 to n
  for j ← 1 to m
    ...
    x ← a * b
    y ← x * c
    ...
  end
end
```

First order invariant

Second order invariant

Need to iterate on a loop until it stabilizes, then move on