Using Static Single Assignment

Last Time

- Basic definition, and why it is useful
- How to build it

Today

- Loop Optimizations
  - Induction variables (standard vs. SSA)
  - Loop Invariant Code Motion (SSA based)

Loop Optimization

Loops are important, they execute often

- typically, some regular access pattern
  - regularity ⇒ opportunity for improvement
  - repetition ⇒ savings are multiplied
- assumption: loop bodies execute $l_0^{\text{depth}}$ times

Classical Loop Optimizations

- Loop Invariant Code Motion
- Induction Variable Recognition
- Strength Reduction
- Linear Test Replacement
- Loop Unrolling
Other Loop Optimizations

- Scalar replacement
- Loop Interchange
- Loop Fusion
- Loop Distribution
- Loop Skewing
- Loop Reversal

Loop Invariant Code Motion

- Build the SSA graph
- Need semi-pruned insertion of $\phi$-nodes:
  If two non-null paths $x \to^+ z$ and $y \to^+ z$ converge at node $z$, and nodes $x$ and $y$ contain assignments to $t$ (in the original program), then a $\phi$-node for $t$ must be inserted at $z$ (in the new program)
  and $t$ must be live across some basic block

Simple test:
If, for a statement $s \equiv [x \leftarrow y \otimes z]$, none of the operands $y, z$ refer to a $\phi$-node or definition inside the loop, then

Transform:
assign the invariant computation a new temporary name, $t \leftarrow y \otimes z$, move it to the loop pre-header, and assign $x \leftarrow t$. 

Loop Invariant Code Motion: Example I

![Diagram of loop invariant code motion](image)

More invariants

Start at loop entry point:

Test: If operands point to definitions inside loop, and those definitions are a function of loop invariants (recursive definition)

Transform: as before for each invariant
Taxonomy of Induction Variables

1. A basic induction variable is a variable $i$
   • whose only definition within the loop is an assignment of the form $i ← i \pm c$, where $c$ is loop invariant.

2. A mutual induction variable $i'$ is
   • defined once within the loop, and its value is a linear function of some other induction variable $i$ such that
     $$i' ← i \otimes c_1 \pm c_2$$
     where $\otimes$ is one of $\times$ or $\div$, and $c_1, c_2$ are loop invariant.

3. The family of a basic induction variable $i$:
   • the set of mutual induction variables on $i$

Induction Variable Recognition

- What is a loop induction variable?
- Why might we want to detect one?

\[
i ← 0
\]
\[
\text{while } i < 10 \text{ do }
\]
\[
i ← i + 1
\]
\[
\text{end }
\]

Simplest Method: Pattern match for $i ← i + c$ in loop and ensure no other definition of $i$ in loop.

Does not catch all loop induction variables.

Optimistic Induction Variable Recognition

\[
IV ← \{\}
\]
\[
\text{foreach statement } s \text{ in loop do }
\]
\[
\text{if } s ≡ [i ← x ± c] ∧ (c is loop invariant) \\nIV ← IV \cup \{i\}
\]
\[
\text{else if } s ≡ [i ← x \otimes c] ∧ (c is loop invariant) \\nIV ← IV \cup \{i\}
\]
\[
\text{end}
\]
\[
\text{end}
\]
\[
\text{changed ← false}
\]
\[
\text{foreach } s ≡ [i ← \ldots] \in IV \text{ do }
\]
\[
\text{if } \exists u ∈ \text{Uses}(s) : u ∉ IV \\nIV ← IV \setminus \{i\}
\]
\[
\text{changed ← true}
\]
\[
\text{end}
\]
\[
\text{end}
\]
\[
\text{while} \text{ changed}
\]

Finds linear induction variables and catches mutual induction variables.
Loop Induction Variables: Example I

\begin{align*}
i &\leftarrow 1 \\
k &\leftarrow 0 \\
\textbf{do} &
\begin{align*}
&j \leftarrow k + 1 \\
&k \leftarrow j + 2 \\
i &\leftarrow i \times 2 \\
\textbf{end}
\end{align*}
\end{align*}

Optimistic Induction Variables

\begin{align*}
i &\leftarrow 0 \\
k &\leftarrow 0 \\
\textbf{do} &
\begin{align*}
&j \leftarrow k + 1 \\
&k \leftarrow j + 2 \\
i &\leftarrow i \times 2 \\
\textbf{end}
\end{align*}
\end{align*}

Loop Induction Variables with SSA

- Build the SSA graph
- Going from the innermost to the outermost loop
- Find cycles in the SSA graph
  Each cycle may be for a basic induction variable
  if the variable in the cycle is a function of loop invariants and its value on the current iteration
  (ie, its \( \phi \) is a function of an initialized variable and an instance of \( v \) in the cycle)
- Other induction variables can depend on basic induction variables.

Loop Induction Variables: Example I

\begin{align*}
i &\leftarrow 1 \\
i_1 &\leftarrow 1 \\
\textbf{do} &
\begin{align*}
&i_2 \leftarrow \phi(i_1, i_3) \\
&\ldots(i) \\
i &\leftarrow i + 1 \\
&i_3 \leftarrow i_2 + 1 \\
&\ldots(i) \\
\textbf{end}
\end{align*}
\end{align*}

Loop Induction Variables with SSA

How to determine: If the variable(s) in the cycle is(are) a function of loop invariants and its value on the current iteration:

- The \( \phi \)-node in the cycle will take one definition from inside the loop and one from outside the loop (assuming \( \phi \)-nodes with only two inputs)
- The definition inside the loop will be part of the cycle and will get one operand from the \( \phi \)-node and any others will be loop invariant
- For linear induction variables the operator will be addition, subtraction, or unary minus
Loop Induction Variables: Example II

\[
i \leftarrow 3 \\
m \leftarrow 0 \\
\textbf{do} \\
\begin{align*}
    j & \leftarrow 3 \\
    i & \leftarrow i + 1 \\
    l & \leftarrow l + 2 \\
    j & \leftarrow j + 2 \\
    k & \leftarrow 2 \times j
\end{align*}
\textbf{end}
\]

\[
i_1 \leftarrow 3 \\
m_1 \leftarrow 0 \\
\textbf{do} \\
\begin{align*}
    i_2 & \leftarrow \phi(i_1, i_3) \\
    m_2 & \leftarrow \phi(m_1, m_3) \\
    j_1 & \leftarrow 3 \\
    i_3 & \leftarrow i_2 + 1 \\
    l_1 & \leftarrow m_2 + 1 \\
    m_3 & \leftarrow l_1 + 2 \\
    j_2 & \leftarrow i_3 + 2 \\
    k_1 & \leftarrow 2 \times j_2
\end{align*}
\textbf{end}
\]

Strength Reduction

Algorithm

Let \( i \) be an induction variable in the family of basic induction variable \( j \), such that: \( i = j \times c_1 + c_2 \)

- Create new variable, \( i' \)
- Initialize in preheader, \( i' = j \times c_1 + c_2 \)
- Track value of \( j \).
  After \( j \leftarrow j + c_3 \), add \( i' = i' + (c_1 \times c_3) \)
- Replace definition of \( i \) with \( i = i' \)

Key point

- \( c_1, c_2 \) and \( c_3 \) are constant or loop invariant, so the computation can be moved out of the loop or folded at compile time
- Reduces number of multiplies executed at run time

Strength Reduction: Example

\[
\begin{align*}
    J & = 0 \\
    I & = 0 \\
    J' & = 4 \times J + &A \\
    I' & = 0 \\
    J & = J + 1 \\
    \text{GOTO L2}
\end{align*}
\]

L2: if \( (J >= 100) \) GOTO L1
\[
\begin{align*}
    I & := 4 \times J + &A \\
    *I & := 0 \\
    J & := J + 1 \\
    \text{GOTO L2}
\end{align*}
\]

L1:


Strength Reduction

Philosophy:
Replace an expensive instruction (eg, multiply) with a cheaper one (eg, addition).
- Applied to induction variable families
- Opportunity: array indexing
- Why?: slow or non-existent integer multiply

Example

\[
\begin{align*}
    J & = 0 \\
    I & := 4 \times J + &A \\
    *I & := 0 \\
    J & := J + 1 \\
    \text{GOTO L2}
\end{align*}
\]

L2: if \( (J >= 100) \) GOTO L1
\[
\begin{align*}
    I & := 4 \times J + &A \\
    *I & := 0 \\
    J & := J + 1 \\
    \text{GOTO L2}
\end{align*}
\]

L1:

Strength Reduction Algorithm

Let \( i \) be an induction variable in the family of basic induction variable \( j \), such that: \( i = j \times c_1 + c_2 \)

- Create new variable, \( i' \)
- Initialize in preheader, \( i' = j \times c_1 + c_2 \)
- Track value of \( j \).
  After \( j \leftarrow j + c_3 \), add \( i' = i' + (c_1 \times c_3) \)
- Replace definition of \( i \) with \( i = i' \)
Candidates for Strength Reduction

- IV multiplied by an invariant
  
  ```
  i ← 2
  while i < k do
    e ← j * 3
    i ← j + 1
    t ← i * 50
    j ← j + 1
  end
  ⇒
  i ← 2
  while i < k do
    i ← i + 1
    t ← i * 50
  end
  ```

```c
candidates ← {}
foreach statement s in loop
  if s ∈ {2 × i} ∧ i ∈ IV ∧ c is loop invariant
    candidates ← candidates ∪ {s}
end
```

- Polynomials: IV multiplied by different IV
- IV multiplied by itself
- IV modulo a constant
- Addition of induction variables

Examples

```
j ← 2
while j < k do
  e ← j * 3
  i ← j + 1
  t ← i * 50
  j ← j + 1
end
```

```
i ← 2
while i < k do
  i ← i + 1
  t ← i * 50
end
```

Strength Reduction Details

- What happens if two induction variables $i_1$ and $i_2$ are in the family of the same basic induction variable $j$ with the same constants $c_1$ and $c_2$?

```c
L1: ...
i = i + 1
j = i + 1
t1 = 4*i + &A
t2 = 4*i + &B
t3 = 4*j + &B
...```
Reduction of operator strength

**Taxonomy — Reduction of Operator Strength**

<table>
<thead>
<tr>
<th>Machine Independent</th>
<th>Machine Dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove redundancy</td>
<td>costly op—cheap op</td>
</tr>
<tr>
<td>move evaluation</td>
<td>yes</td>
</tr>
<tr>
<td>specialize</td>
<td>yes</td>
</tr>
<tr>
<td>remove useless code</td>
<td>maybe</td>
</tr>
<tr>
<td>expose opportunities</td>
<td>yes</td>
</tr>
</tbody>
</table>

| remove redundancy   | no     |
| move evaluation     | no     |
| specialize          | yes    |
| remove useless code | maybe  |
| expose opportunities| yes    |

Example

```
i ← 2
i.50 ← i × 50
while i < k do
  i ← i + 1
  i.50 ← i.50 + 50
  ... i.50
end
⇒
i ← 2
i.50 ← i × 50
while i.50 < 50 × k do
  i.50 ← i.50 + 50
  ... i.50
end
```

Linear Test Replacement

Eliminate the induction variable altogether
- the loop test often is the last use of a basic induction variable after strength reduction
- fewer instructions, fewer live ranges

**Algorithm**
- If the only use of a IV is the loop test and its own increment
  and if the test is always computed (ie, there is only one exit from the loop)
- Then replace the test with an equivalent one.
  Say test is “i compare k”:
    If ∃i.c ∈ IV then replace test with “i.c compare c × k”
- How does the sign of c affect the test?