Using Static Single Assignment

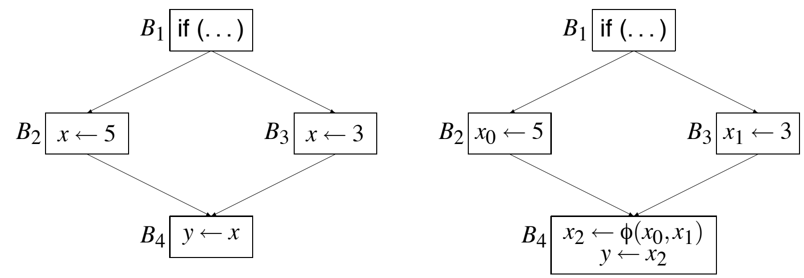
Last Time

- 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)

SSA form



Loop Optimization

Loops are important, they execute often

- typically, some regular access pattern

  regularity $\Rightarrow$ opportunity for improvement

  repetition $\Rightarrow$ savings are multiplied

- *assumption*: loop bodies execute 10^{depth} times

Classical Loop Optimizations

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

Other Loop Optimizations

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 ϕ-nodes:

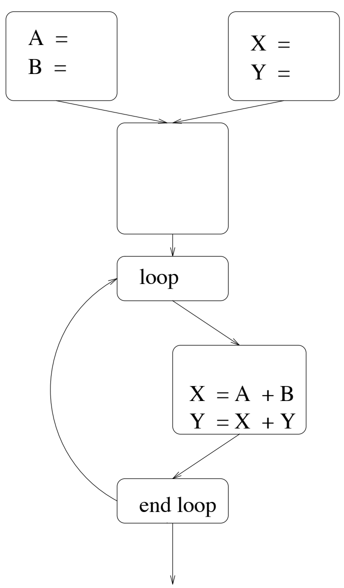
  If two non-null paths $x \rightarrow^{+} z$ and $y \rightarrow^{+} z$ converge at node z, and nodes x and y contain assignments to t (in the original program), then a ϕ-node for t must be inserted at z (in the new program)

  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 ϕ-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



Loop Invariant Code Motion

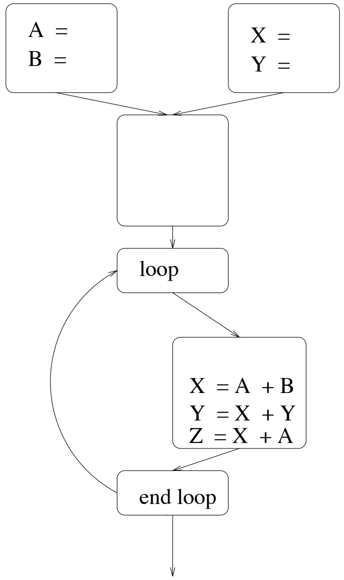
More invariants

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

Loop Invariant Code Motion: Example II



Induction Variable Recognition

- What is a loop induction variable?

- Why might we want to detect one?

$i \leftarrow 0$
while $i < 10$ **do**
    $i \leftarrow i + 1$
end

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

Does not catch all loop induction variables.

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 $i \leftarrow i \pm c$, 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

    $$i' \leftarrow i \otimes c_1 \pm c_2$$

    where $\otimes$ is one of $\times$ or $/$, 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

Optimistic Induction Variable Recognition

IV $\leftarrow \{\}$
foreach statement s in loop **do**
  if $s \equiv [i \leftarrow x \pm c] \wedge (c$ is loop invariant$)$
    IV $\leftarrow$ *IV* $\cup \{i\}$
  elsif $s \equiv [i \leftarrow x \otimes c] \wedge c$ is loop invariant
      IV $\leftarrow$ *IV* $\cup \{i\}$
  end
end
do
   changed $\leftarrow$ *false*
   foreach $s \equiv [i \leftarrow \ldots] \in$ *IV* **do**
     if $\exists u \in Uses(s) : u \notin$ *IV*
       IV $\leftarrow$ *IV* $- \{i\}$
       changed $\leftarrow$ *true*
     end
   end
while *changed*

Finds linear induction variables and catches mutual induction variables.

Optimistic Induction Variables

$i \leftarrow 0$
$k \leftarrow 0$
do
    $j \leftarrow k+1$
    $k \leftarrow j+2$
    $i \leftarrow i \times 2$
end

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 ϕ 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

$i \leftarrow 1$	$i_1 \leftarrow 1$
do	**do**
	$i_2 \leftarrow \phi(i_1, i_3)$
$\ldots(i)\ldots$	$\ldots(i_2)\ldots$
$i \leftarrow i+1$	$i_3 \leftarrow i_2+1$
$\ldots(i)\ldots$	$\ldots(i_3)\ldots$
end	**end**

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 ϕ-node in the cycle will take one definition from inside the loop and one from outside the loop (*assuming ϕ-nodes with only two inputs*)
- The definition inside the loop will be part of the cycle and will get one operand from the ϕ-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

$$
\begin{array}{lcl}
i \leftarrow 3 & & i_1 \leftarrow 3 \\
m \leftarrow 0 & & m_1 \leftarrow 0 \\
\underline{\textbf{do}} & & \underline{\textbf{do}} \\
 & & \quad i_2 \leftarrow \phi(i_1, i_3) \\
 & & \quad m_2 \leftarrow \phi(m_1, m_3) \\
\quad j \leftarrow 3 & & \quad j_1 \leftarrow 3 \\
\quad i \leftarrow i+1 & \Rightarrow & \quad i_3 \leftarrow i_2+1 \\
\quad l \leftarrow m+1 & & \quad l_1 \leftarrow m_2+1 \\
\quad m \leftarrow l+2 & & \quad m_3 \leftarrow l_1+2 \\
\quad j \leftarrow i+2 & & \quad j_2 \leftarrow i_3+2 \\
\quad k \leftarrow 2 \times j & & \quad k_1 \leftarrow 2 \times j_2 \\
\underline{\textbf{end}} & & \underline{\textbf{end}}
\end{array}
$$

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

```
    J = 0                          for (J = 0; J<100; J++)
                                       A(J) = 0
L2: if (J>=100) GOTO L1
    I := 4 * J  + &A
    *I := 0
    J := J + 1
    GOTO L2
L1:
```

Allen, Cocke, Kennedy, "Reduction in Operator Strength," in *Program Flow Analysis*, Muchnick and Jones (Eds), 1981, pp 79–101

Strength Reduction Algorithm

Algorithm

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

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

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

```
    J = 0                                J  = 0
                                         I’ = 4 * J + &A
L2: if (J >= 100) GOTO L1           L2:  if (J >= 100) GOTO L1
    I = 4 * J  + &A                      I  = I’
    *I = 0                   ⇒           *I = 0
    J = J + 1                            J  = J + 1
                                         I’ = I’ + (4 * 1)
    GOTO L2                              GOTO L2
L1:                                 L1:
```

Candidates for Strength Reduction

- *IV* multiplied by an invariant

$$
\begin{array}{lcl}
i \leftarrow 2 & & i \leftarrow 2 \\
 & & i.50 \leftarrow i \times 50 \\
\vdots & \Rightarrow & \vdots \\
i \leftarrow i+1 & & i \leftarrow i+1 \\
 & & i.50 \leftarrow i.50+50 \\
\ldots i \times 50 & & \ldots i.50
\end{array}
$$

candidates $\leftarrow \{\}$
foreach statement s in loop
  if $s \equiv [i' \leftarrow i \times c] \wedge i \in$ *IV* $\wedge\, c$ is loop invariant
    candidates $\leftarrow$ *candidates* $\cup \{s\}$
  end
end

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

Examples

$$
\begin{array}{lcl}
i \leftarrow 2 & & i \leftarrow 2 \\
 & & i.50 \leftarrow i \times 50 \\
\textbf{while } i < k \textbf{ do} & & \textbf{while } i < k \textbf{ do} \\
\quad i \leftarrow i+1 & \Rightarrow & \quad i \leftarrow i+1 \\
 & & \quad i.50 \leftarrow i.50+50 \\
\quad t \leftarrow i \times 50 & & \quad t \leftarrow i.50 \\
\textbf{end} & & \textbf{end}
\end{array}
$$

Examples

$$
\begin{array}{lcl}
j \leftarrow 2 & & j \leftarrow 2 \\
\textbf{while } j < k \textbf{ do} & & \textbf{while } j < k \textbf{ do} \\
\quad e \leftarrow j*3 & & \\
\quad i \leftarrow j+1 & \Rightarrow & \quad i \leftarrow j+1 \\
\quad t \leftarrow i*50 & & \\
\quad j \leftarrow j+1 & & \\
\textbf{end} & & \textbf{end}
\end{array}
$$

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?

- When might this happen in real code?

```
do i = 1, n
    A(i) = B(i) + B(i+1)
```

```
    i = 0

L1: ...
    i  = i + 1
    j  = i + 1
    t1 = 4*i + &A
    t2 = 4*i + &B
    t3 = 4*j + &B
    ...
```

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 $\exists i.c \in$ *IV* then replace test with "$i.c$ compare $c \times k$"
- How does the sign of c affect the test?

Example

$i \leftarrow 2$		$i \leftarrow 2$
$i.50 \leftarrow i \times 50$		$i.50 \leftarrow i \times 50$
while $i < k$ **do**	$\Rightarrow$	**while** $i.50 < 50 * k$ **do**
$\quad i \leftarrow i + 1$		
$\quad i.50 \leftarrow i.50 + 50$		$\quad i.50 \leftarrow i.50 + 50$
$\quad \ldots i.50$		$\quad \ldots i.50$
end		**end**

Reduction of operator strength

Taxonomy — Reduction of Operator Strength		
Machine Independent		
remove redundancy	no	(gets some cses)
move evaluation	no	
specialize	yes	
remove useless code	maybe	
expose opportunities	yes	
Machine Dependent		
costly op→cheap op	yes	assumes `mult` costly
hide latency	no	
use powerful op	no	