Optimization

Last Time

Building SSA

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

Today

- Putting SSA to work
- Analysis and Transformation
  - **Analysis** - proves facts about programs
  - **Transformation** - changes the program to make it "better" while preserving its semantics
- SSA Loop Optimizations
  - Loop Invariant Code Motion
  - Induction variables
  - While we do the above, let's think about comparing SSA with dataflow def/use chains.

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 10^{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 (also known as Fision)
- Loop Skewing
- Loop Reversal

Loop Invariant Code Motion

- Build the SSA graph
- Need Briggs-minimal insertion of ϕ-nodes

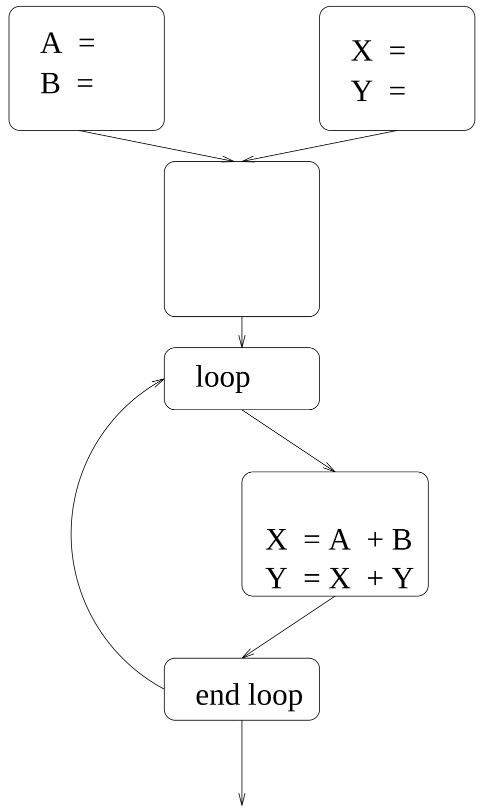
If two non-null paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ converge at node Z, and nodes X and Y contain assignments to V (in the original program), then a ϕ-node for V must be inserted at Z (in the new program).

and V must be live across some basic block

Simple test:
for a statement s, none of the operands point to a ϕ-node or a definition inside the loop.

Transformation:
Given, $l = r_1$ *op* r_2, assign the computation a new temporary name, $t_k = r_1$ *op* r_2, move it to the loop pre-header, and assign $l = t_k$.

Loop Invariant Code Motion Example I



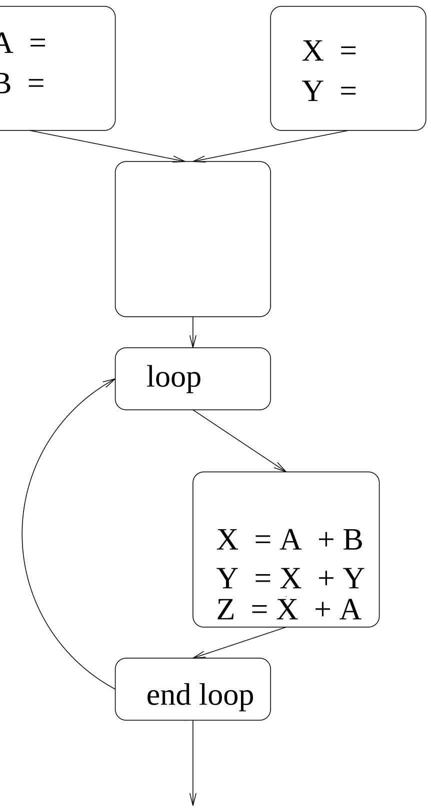
Loop Invariant Code Motion

More invariants:

- Start at roots in loop
- If the operands point to a definition inside loop, and that definition is a function of loop invariants (*recursive definition*).
- Do the same replacement as in the simple test as each invariant expression is found.

After we finish the example, let's think about using use/def chains from dataflow for loop invariant code motion.

Loop Invariant Code Motion Example II



Any more?

Induction Variable Recognition

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

$i = 0$
while $i < 10$ do

    $i = i + 1$
end while

Simplest Method:

Pattern match for "$i = i + b$" in loop and look at the DEF/USE chains to determine there are no other assignment to i in loop.

Problem: Does not catch all induction variables.

Taxonomy of Induction Variables

1. A *basic* induction variable is a variable `J`
   - whose only definition within the loop is an assignment of the form `J := J` $\pm$ `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(s) `I'` such that

     `I = c1 * I'` $\pm$ `c2`

     or

     `I = I' / c1` $\pm$ `c2`.

     where `c1`, `c2` are loop invariant.

Optimistic Induction Variable Recognition

$IV = \emptyset$
for each statement s in the loop
    if op is `ADD`,`SUB`, or `NEG`
        add s to IV
    if op is `LOAD` or `STORE` & address is loop invariant
        add s to IV
end for
repeat
    changes = false
    for each s in IV
        if either operand is not in IV
            remove s from IV
            changes = true
        endif
    end for
until $\neg$ *changes*

Finds linear induction variables.
Catches mutual induction variables.
Does not exploit SSA

Optimistic Induction Variables

$i = 0$
$k = 0$
loop

    $j = k + 1$
    $k = j + 2$

    $i = i * 2$

end loop

Loop Induction Variables with SSA

- Build the SSA graph
- Going from the innermost to the outermost loop
- Find cycles in the graph

  Each cycle *may* be a *basic* induction variable

  If the variable(s) in the cycle is a function of loop invariants and its value on the current iteration,
  i.e., 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

$i = 1$	$i_1 = 1$
loop	loop
	$i_2 = \phi(i_1, i_3)$
$\ldots (i) \ldots$	$\ldots (i_2) \ldots$
$i = i + 1$	$i_3 = i_2 + 1$
$\ldots (i) \ldots$	$\ldots (i_3) \ldots$
end loop	end loop

Loop Induction Variables with SSA

Are the variable(s) in the cycle 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*).
- Two statement cycle: 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.
- Larger cycles: Other definitions in the cycle will chain as follows:

  $i_2 = \phi(i_0, i_1)$

  $j = i_2 \pm c1$

  $k = j \pm c2$

  $\ldots$

  $r = q \pm c6$

  $i_1 = r \pm c7$

  The definition at ϕ is the basic induction variable.

  Mutual induction variables are functions of other induction variables. If they are in a cycle with a basic variable, they must follow the above form. If they are not in a cycle, they can take the form $j = c1 * i \pm c2$ where i is an induction variable.

Loop Induction Variables Example II

$i = 3$
$m = 0$
loop
    $j = 3$
    $i = i + 1$
    $l = m + 1$
    $m = l + 2$
    $j = i + 2$
    $k = 2 * j$
end loop

$i_1 = 3$
$m = 0$
loop
    $i_2 = \phi\ (i_1, i_3)$
    $m_2 = \phi\ (m_1, m_3)$
    $j_1 = 3$
    $i_3 = i_2 + 1$
    $l_1 = m_2 + 1$
    $m_3 = l_1 + 2$
    $j_2 = i_3 + 2$
    $k_1 = 2 * j_2$
end loop

Next Time

Optimizating expressions

- common subexpression elimination
- value numbering