Loop Invariant Code Motion

Last Time

- Uses of SSA: reaching constants, dead-code elimination, induction variable identification

Today

- Finish up induction variable identification
- Loop invariant code motion

Next Time

- Reuse optimization
  - Global value numbering
  - Common subexpression elimination

Induction Variable Identification (cont)

Types of Induction Variables

- **Basic** induction variables (eg. loop index)
  - Variables that are defined once in a loop by a statement of the form, **`i=i+c`** (or **`i=i*c`**), where **`c`** is a constant integer
- **Derived** induction variables
  - Variables that are defined once in a loop as a linear function of another induction variable
    - $j = c_1 * i + c_2$
    - $j = i / c_1 + c_2$, where c_1 and c_2 are loop invariant

Induction Variable Identification (cont)

Informal SSA-based Algorithm

- Build the SSA representation
- Iterate from innermost CFG loop to outermost loop
  - Find SSA cycles
    - Each cycle **may** be a **basic** induction variable if a variable in a cycle is a function of loop invariants and its value on the current iteration
  - Find **derived** induction variables as functions of loop invariants, its value on the current iteration, and basic induction variables

Induction Variable Identification (cont)

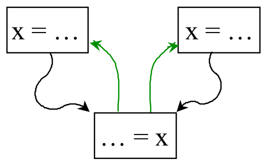
Informal SSA-based Algorithm (cont)

- Determining whether a variable is a function of loop invariants and its value on the current iteration
    - The ϕ -function in the cycle will have as one of its inputs a def from inside the loop and a def from outside the loop
    - The def inside the loop will be part of the cycle and will get one operand from the ϕ -function and all others will be loop invariant
    - The operation will be plus, minus, or unary minus

Loop Invariant Code Motion
Background: ud- and du-chains

ud-Chains

- A ud-chain connects a use of a variable to all defs of a variable that might reach it (a sparse representation of **Reaching Definitions**)



du-Chains

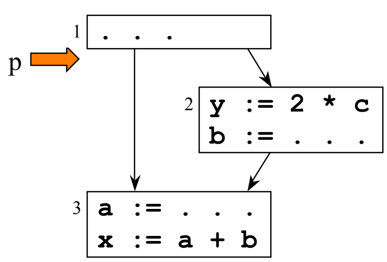
- A du-chain connects a def to all uses that it might reach (a sparse representation of **Upward Exposed Uses**)

How do ud- and du-chains differ?

Upward Exposed Uses

Definition

- An **upward exposed use** at program point p is a use that may be reached by a definition at p (i.e, no intervening definitions).



How do upward exposed uses differ from live variables?

Identifying Loop Invariant Code

Motivation

- Avoid redundant computations

Example

```
      w = . . .
      y = . . .
      z = . . .
L1:   x = y + z
      v = w + x
      . . .
      if . . . goto L1
```

Everything that x depends upon is computed outside the loop, *i.e.,* all defs of y and z are outside of the loop, so we can move x = y + z outside the loop

What happens once we move that statement outside the loop?

Algorithm for Identifying Loop Invariant Code

Input: A loop L consisting of basic blocks. Each basic block contains a sequence of 3-address instructions. We assume ud-chains have been computed.

Output: The set of instructions that compute the same value each time through the loop

Informal Algorithm:

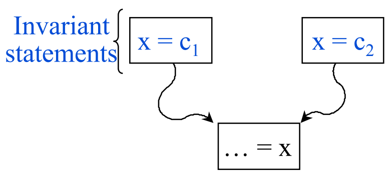
1. Mark "invariant" those statements whose operands are either
   - Constant
   - Have all reaching definitions outside of L
2. Repeat until a fixed point is reached: mark "invariant" those unmarked statements whose operands are either
   - Constant
   - Have all reaching definitions outside of L
   - Have exactly one reaching definition and that definition is in the set marked "invariant"

Is this last condition too strict?

Algorithm for Identifying Loop Invariant Code (cont)

Is the Last Condition Too Strict?

- No
- If there is more than one reaching definition for an operand, then neither one dominates the operand
- If neither one dominates the operand, then the value can vary depending on the control path taken, so the value is not loop invariant



Code Motion

What's the Next Step?

- Do we simply move the "invariant" statements outside the loop?

- No, there are three requirements that ensure that code motion does not change program semantics. For some statement

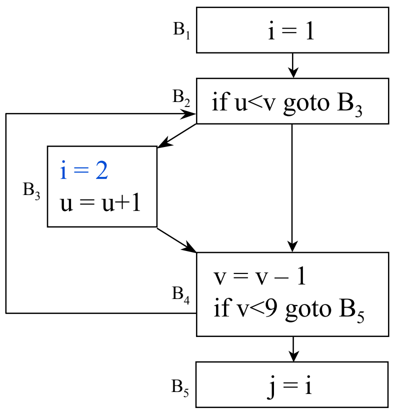
  s: `x = y + z`

  1. The block containing s dominates all loop exits
  2. No other statement in the loop assigns to x
  3. No use of x in the loop is reached by any def of x other than s

Example 1

Condition 1 is Needed

- If the block containing s does not dominate all exits, we might assign to x when we're not supposed to



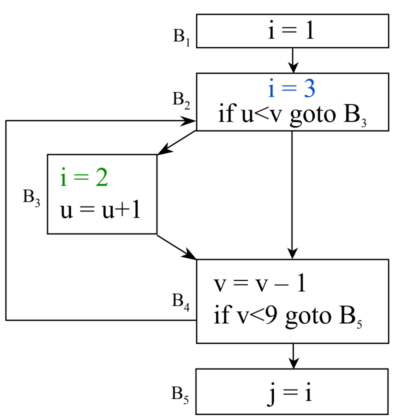
Can we move i=2 outside the loop?

i=2 is loop invariant, but B_3 does not dominate B_4, the exit node, so moving i=2 would change the meaning of the loop for those cases where B_3 is never executed

Example 2

Condition 2 is Needed

- If some other statement in the loop assigns x, the movement of the statement may cause some statement to see the wrong value



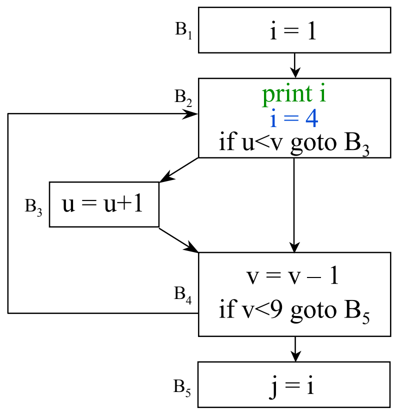
Can we move i=3 outside the loop?

B_2 dominates the exit so condition 1 is satisfied, but code motion will set the value of i to 2 if B_3 is ever executed, rather than letting it vary between 2 and 3.

Example 3

Condition 3 is Needed

- If a use in L can be reached by some other def, then we cannot move the def outside the loop



Can we move i=4 outside the loop?

Conditions 1 and 2 are met, but the use of i in block B_2, can be reached from a different def, namely i=1 from B_1.

If we were to move i=4 outside the loop, the first iteration through the loop would print 4 instead of 1

Loop Invariant Code Motion Algorithm

Input: A loop L with ud-chains and dominator information

Output: A modified loop with a preheader and 0 or more statements moved to the preheader

Algorithm:

1. Find loop-invariant statements
2. For each statement s defining x found in step 1, move s to preheader if:
   a. s is in a block that dominates all exits of L,
   b. x is not defined elsewhere in L, and
   c. all uses in L of x can only be reached by the def of x in s

Correctness

Conditions 2a and 2b ensure that the value of x computed at s is the value of x after any exit block of L. When we move s to the preheader, s will still be the def that reaches any of the exit blocks of L.

Condition 2c ensures that any use of x inside of L used (and continues to use) the value of x computed by s

Loop Invariant Code Motion Algorithm (cont)

Profitability

- Can loop invariant code motion ever increase the running time of the program?
- Can loop invariant code motion ever increase the number of instructions executed?
- Before transformation, s is executed at least once (condition 2a)
- After transformation, s is executed exactly once

Relaxing Condition 1

- If we're willing to sometimes do more work: Change the condition to
  a. The block containing s either dominates all loop exits, or x is dead after the loop

Alternate Approach to Loop Invariant Code Motion

Division of labor

- Move all invariant computations to the preheader and assign them to temporaries
- Use the temporaries inside the loop
- Insert copies where necessary
- Rely on Copy Propagation to remove unnecessary assignments

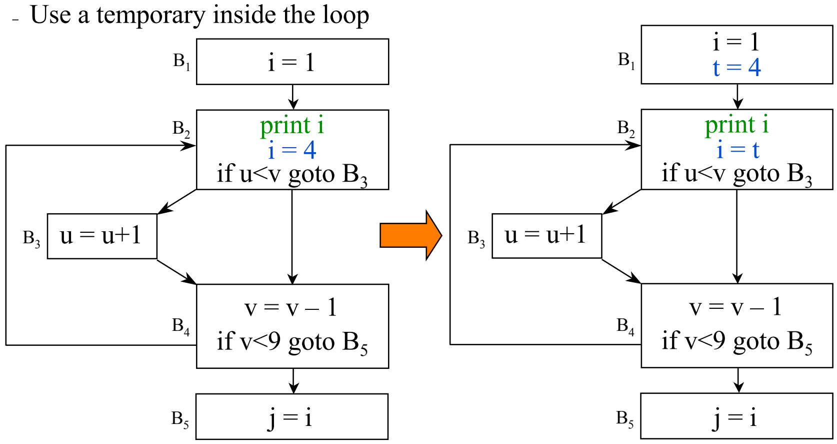
Benefits

- Much simpler: Fewer cases to handle

Example 3 Revisited

Using the alternate approach

- Move the invariant code outside the loop
- Use a temporary inside the loop



Lessons

Why did we study loop invariant code motion?

- Loop invariant code motion is an important optimization
- Because control flow, it's more complicated than you might think
- The notion of dominance is useful in reasoning about control flow
- Division of labor can greatly simplify the problem

Next Time

Lecture

- More reuse optimization