

## Loop Invariant Code Motion

### Last Time

- Control flow analysis

### Today

- Loop invariant code motion

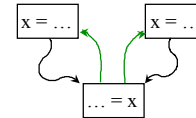
### Next Time

- Induction variable identification and elimination

## 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

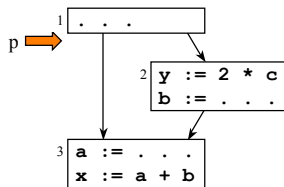
- A du-chain connects a def to all uses that it might reach (a sparse representation of **Upward Exposed Uses**)

### 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:

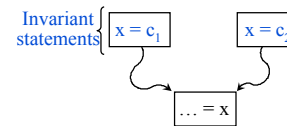
1. Mark "invariant" those statements whose operands are either
  - Constant
  - Have all reaching definitions outside of L
2. Repeat until a fixed point is reached: mark "invariant" those unmarked statements whose operands are either
  - Constant
  - Have all reaching definitions outside of L
  - Have 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

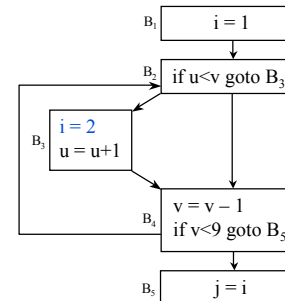
### What's the Next Step?

- 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, after the loop might get a different value



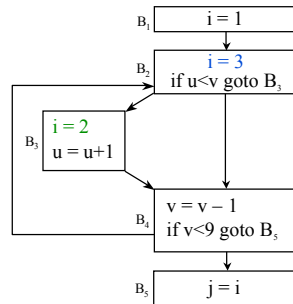
Can we move  $i=2$  outside the loop?

$i=2$  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  $i$ , 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.

CS553 Lecture

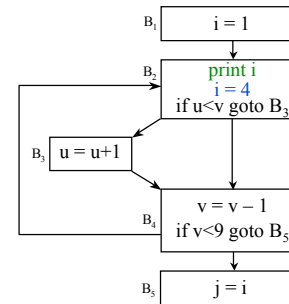
Loop Invariant Code Motion

9

## 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

CS553 Lecture

Loop Invariant Code Motion

10

## 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. Insert preheader
3. 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.  $x$  is not live-out of the loop preheader

**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  $L$  (and continues to use) the value of  $x$  computed by  $s$

CS553 Lecture

Loop Invariant Code Motion

11

## 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

CS553 Lecture

Loop Invariant Code Motion

12

## 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

CS553 Lecture

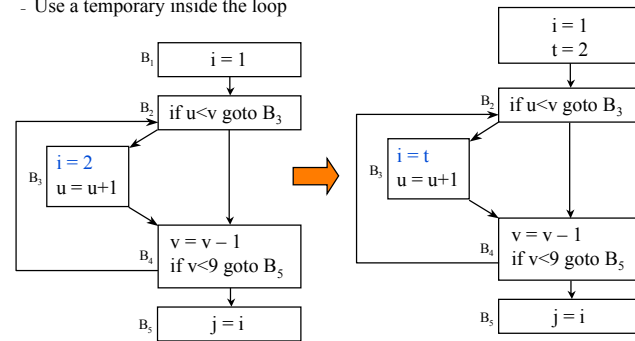
Loop Invariant Code Motion

13

## Example 1 Revisited

### Using the alternate approach

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



CS553 Lecture

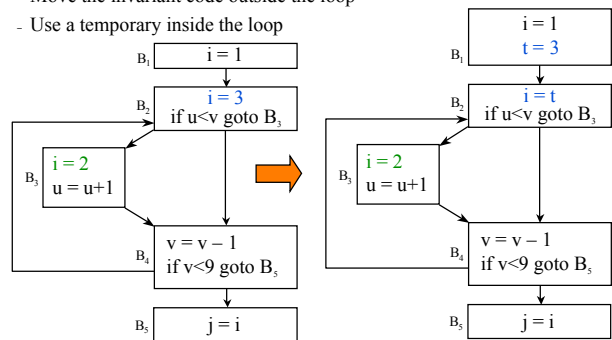
Loop Invariant Code Motion

14

## Example 2 Revisited

### Using the alternate approach

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



CS553 Lecture

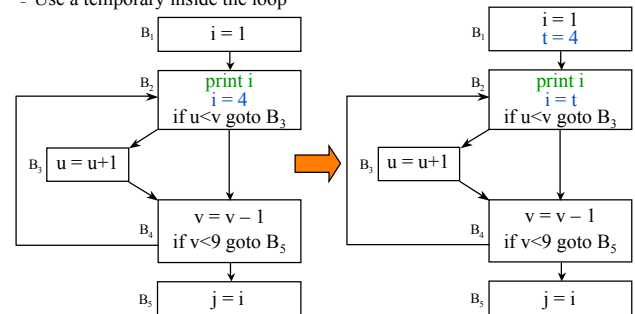
Loop Invariant Code Motion

15

## Example 3 Revisited

### Using the alternate approach

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



CS553 Lecture

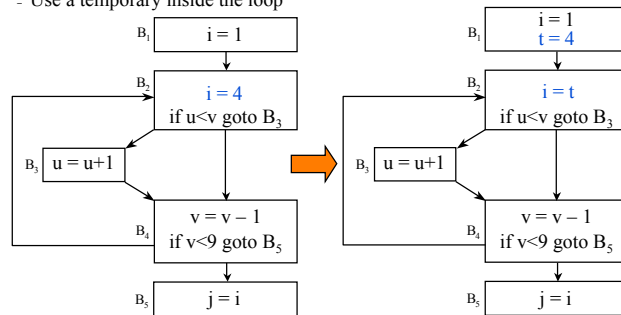
Loop Invariant Code Motion

16

## Case where copy prop and dead code do code motion

### Using the alternate approach

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



CS553 Lecture

Loop Invariant Code Motion

17

## Converting While Loops to Do-While Loops

**Input:** AssemFlowGraph

**Output:** Modified AssemFlowGraph with loop checks at bottom of loop.

### Algorithm:

1. Find loops. Each loop has a set of nodes, a header, an exit node, and a tail node.
2. Insert preheader. A new node with a newly generated label instruction.
3. Find LoopCheck subgraph. Set of nodes and edges not including the header that are reachable from the header and end at the exit node. Does include the exit node.
4. Move LoopCheck subgraph between preheader and header. Make exit node fall through be the header.
5. Replace the loop tail with a copy of the LoopCheck subgraph.
  1. Reverse logic in exit node. NE should become EQ.
  2. Cjump should jump to loop header.
  3. Previous jump should become the fall through.

CS553 Lecture

Loop Invariant Code Motion

18

## 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

CS553 Lecture

Loop Invariant Code Motion

19

## Next Time

### Reading

- Ch 18.3

### Lecture

- Induction variable elimination

### Project 3

- Will post sometime next week

CS553 Lecture

Loop Invariant Code Motion

20