Loop Invariant Code Motion

Last Time
- SSA

Today
- Loop invariant code motion
- Reuse optimization

Next Time
- Discussion (SCC paper)
- More reuse optimization
  - Common subexpression elimination
  - Partial redundancy elimination

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 RTL instructions. We assume ud-chains exist.

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

Informal Algorithm:

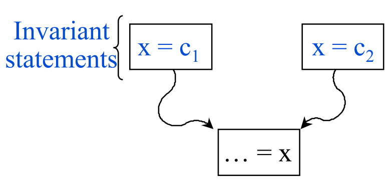
1. Mark "invariant" those statements whose operands are either
   - Constant
   - Have all reaching definitions outside of L
2. Repeat until a fixed point is reached: mark "invariant" those unmarked statements whose operands are either
   - Constant
   - Have all reaching definitions outside of L
   - 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, we need to make sure that we don't change the dominance relations involving any invariant statement

- Three conditions must be met. For some statement

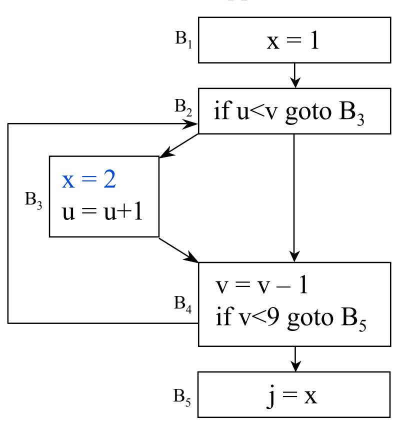
  s: `x = y + z`

  1. The block containing s dominates all loop exits
  2. No other statement in the loop assigns to x
  3. The block containing s dominates all uses of x in the loop

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



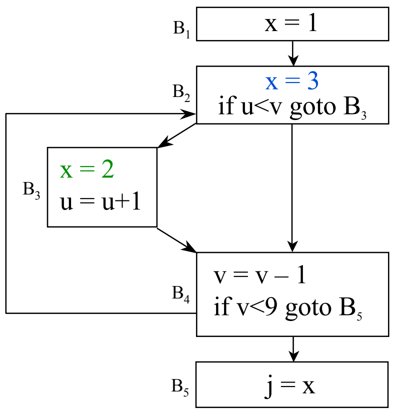
Can we move x=2 outside the loop?

x=2 is loop invariant, but B_3 does not dominate B_4, the exit node, so moving x=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



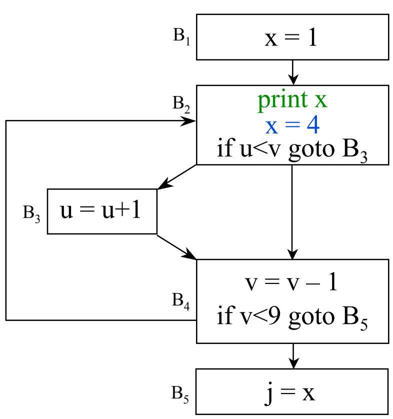
Can we move x=3 outside the loop?

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

Example 3

Condition 3 is Needed

- If the block containing s does not dominate all uses of x in the loop, we might not assign the correct value of x



Can we move x=4 outside the loop?

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

If we were to move x=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 s that def x
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. s is in a block that dominates all uses of x in L

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
- Rely on Copy Propagation to remove unnecessary assignments

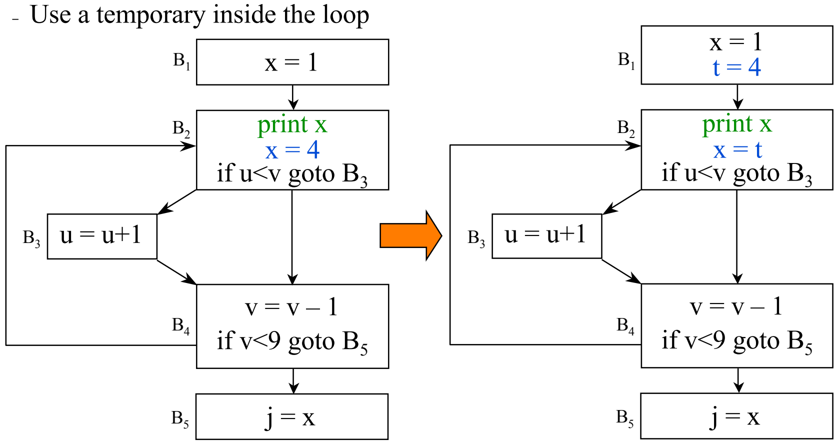
Benefits

- 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 of 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

Reuse Optimization

Idea

- Eliminate redundant operations in the dynamic execution of instructions

How do redundancies arise?

- Multiple array index calculations
- Sequence of similar operations (*e.g.,* method lookup)
- Lightning frequently strikes twice

Types of reuse optimization

- Loop invariant code motion
- Value numbering
- Common subexpression elimination
- Partial redundancy elimination

Value Numbering

Idea

- Partition program variables into **congruence classes**
- All variables in a particular congruence class have the same value

Congruence

- If x and y are congruent then f(x) and f(y) are congruent

x and **y** are congruent

```
a' = a
b' = b
x = f(a ,b)
y = f(a',b')
```


Local Value Numbering

Build congruence classes in program order

- Map each variable, expression, and constant to some unique number, which represents a congruence class
- When we encounter a variable, expression or constant, see if it's already been mapped to a number
  - If so, use the value for that number
  - If not, map to a new number

Example

```
a := b + c
d := b
b := a
e := d + c    a
```

b → ~~#1~~ #3
c → #2
b + c is #1 + # 2 → #3
a → #3
d → #1
d + c is #1 + # 2 → #3
e → #3

Local Value Numbering (cont)

Temporaries may be necessary

$\boxed{\texttt{a}} := \boxed{\texttt{b}}\boxed{+}\boxed{\texttt{c}}$

$\boxed{\texttt{a}} := \boxed{\texttt{b}}$

$\boxed{\texttt{d}} := \boxed{\texttt{b}}\boxed{+}\boxed{\texttt{c}}$

$\texttt{b} \rightarrow \#1$

$\texttt{c} \rightarrow \#2$

$\texttt{b + c}$ is $\#1 + \#2 \rightarrow \cancel{\#3}$

$\texttt{a} \rightarrow \cancel{\#3}\ \#1$

$\texttt{b + c}$ is $\#1 + \#2 \rightarrow \#4$

$\texttt{d} \rightarrow \#4$

```
t := b + c
a := b
```

$\texttt{d} := \cancel{\texttt{b + c}}\ \texttt{t}$

$\texttt{b} \rightarrow \#1$

$\texttt{c} \rightarrow \#2$

$\texttt{b + c}$ is $\#1 + \#2 \rightarrow \#3$

$\texttt{a} \rightarrow \cancel{\#3}\ \#1$

$\texttt{b + c}$ is $\#1 + \#2 \rightarrow \#3$

$\texttt{d} \rightarrow \#3$

Global Value Numbering

Issue

- Need to handle control flow
- SSA form is helpful

Approaches to computing congruence classes

- Pessimistic
  - Assume no variables are congruent (start with *n* classes)
  - Iteratively coalesce classes that are determined to be congruent
- Optimistic
  - Assume all variables are congruent (start with one class)
  - Iteratively partition variables that contradict assumption
  - Slower but better results

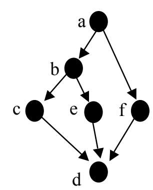
Pessimistic Global Value Numbering

Idea

- Initially each variable is in its own congruence class
- Consider each assignment statement s (reverse postorder in CFG)
  - Update LHS value number with hash of RHS
- Identical value number ⇒ congruence

Why reverse postorder?

- Ensures that when we consider an assignment statement, we have already considered definitions that reach the RHS operands



Postorder: d, c, e, b, f, a

Complication

Recall our basic assumption

- If x and y are congruent then f(x) and f(y) are congruent

x and **y** are congruent

```
a'= a
b'= b
x = f(a ,b)
y = f(a',b')
```

Problem

- This assumption is not true for ϕ-functions

$a_1 = x_1$ $a_2 = y_1$ $b_1 = x_1$ $b_2 = y_1$

n m

$a_3 = \phi_n(a_1, a_2)$ $b_3 = \phi_m(b_1, b_2)$

a_1 & b_1 congruent?

a_2 & b_2 congruent?

a_3 & b_3 congruent?

Solution: Label ϕ-functions with join point

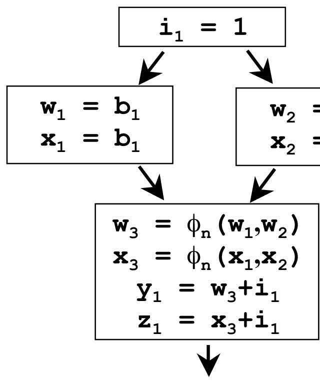
Algorithm

for each assignment of the form: "**`x = f(a,b)`**"

ValNum[x] ← UniqueValue()

for each assignment of the form: "**`x = f(a,b)`**" (in reverse postorder)

ValNum[x] ← Hash(f ⊕ ValNum[a] ⊕ ValNum[b])



a_1 *#1*
b_1 *#2*
i_1 *#3*
w_1 ~~*#4*~~ *#2*
x_1 ~~*#5*~~ *#2*
w_2 ~~*#6*~~ *#1*
x_2 ~~*#7*~~ *#1*
w_3 ~~*#8*~~ *ϕ_n(#2,#1) → #12*
x_3 ~~*#9*~~ *ϕ_n(#2,#1) → #12*
y_1 ~~*#10*~~ *+(#12,#3) → #13*
z_1 ~~*#11*~~ *+(#12,#3) → #13*

Snag!

Problem

- Our algorithm assumes that we consider operands before variables that depend upon it
- Can't deal with code containing loops!

Solution

- Ignore back edges
- Make conservative (worst case) assumption for previously unseen variable (*i.e.,* assume its in its own congruence class)

Optimistic Global Value Numbering

Idea

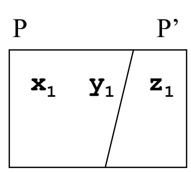
- Initially all variables in one congruence class
- Split congruence classes when evidence of non-congruence arises
  - Variables that are computed using different functions
  - Variables that are computed using functions with non-congruent operands

Splitting

Initially

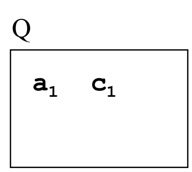
- Variables computed using the same function are placed in the same class

$$x_1 = f(a_1, b_1)$$
. . .
$$y_1 = f(c_1, d_1)$$
. . .
$$z_1 = f(e_1, f_1)$$



Iteratively

- *Split* classes when corresponding operands are in different classes
- Example: a_1 and c_1 are congruent, but e_1 is congruent to neither

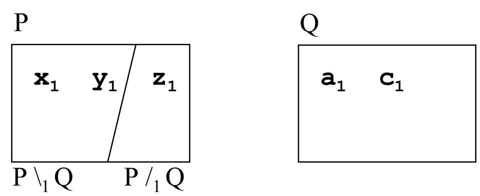


Splitting (cont)

Definitions

- Suppose P and Q are sets representing congruence classes
- Q **splits** P for each i into two sets
  - $P \setminus_i Q$ contains variables in P whose i^{th} operand is in Q
  - $P /_i Q$ contains variables in P whose i^{th} operand is not in Q
- Q **properly splits** P if neither resulting set is empty

```
x1 = f(a1,b1)
. . .
y1 = f(c1,d1)
. . .
z1 = f(e1,f1)
```



Algorithm

worklist $\leftarrow \varnothing$
for each function f
    $C_f \leftarrow \varnothing$
    for each assignment of the form "x = f(a,b)"
        $C_f \leftarrow C_f \cup \{ x \}$
    worklist $\leftarrow$ worklist $\cup \{C_f\}$
    $CC \leftarrow CC \cup \{C_f\}$
while worklist $\neq \varnothing$
    Delete some D from worklist
    for each class C properly split by D (at operand i)
        $CC \leftarrow CC - C$
        worklist $\leftarrow$ worklist $- C$
        Create new congruence classes $C_j \leftarrow \{C \setminus_i D\}$ and $C_k \leftarrow \{C /_i D\}$
        $CC \leftarrow CC \cup C_j \cup C_k$
        worklist $\leftarrow$ worklist $\cup C_j \cup C_k$

Example

SSA code

```
x0 = 1
y0 = 2
x1 = x0+1
y1 = y0+1
z1 = x0+1
```

Congruence classes

S_0	$\{x_0\}$
S_1	$\{y_0\}$
$\cancel{S_2}$	$\cancel{\{x_1, y_1, z_1\}}$
S_3	$\{x_1, z_1\}$
S_4	$\{y_1\}$

Worklist: $\cancel{S_0=\{x_0\}}$, $\cancel{S_1=\{y_0\}}$, $\cancel{S_2=\{x_1, y_1, z_1\}}$ $\cancel{S_3=\{x_1, z_1\}}$, $\cancel{S_4=\{y_1\}}$

S_0 psplit S_0? no S_0 psplit S_1? no S_0 psplit S_2? yes!

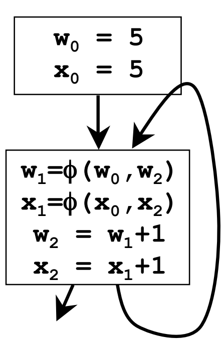
$S_2 \backslash_1 S_0 = \{x_1, z_1\} = S_3$

$S_2 /_1 S_0 = \{y_1\} = S_4$

Comparing Optimistic and Pessimistic

Differences

- Handling of loops
- Pessimistic makes worst-case assumptions on back edges
- Optimistic requires actual contradiction to split classes



Role of SSA

Single global result

- Variables correspond to values

a not congruent to anything

```
a = b
. . .
a = c
. . .
a = d
```

Congruence classes: $\{a_1,b\}$, $\{a_2,c\}$, $\{a_3,d\}$

```
a1 = b
. . .
a2 = c
. . .
a3 = d
```

No data flow analysis

- Optimistic: Iterate over congruence classes, not CFG nodes
- Pessimistic: Visit each assignment once

ϕ-functions

- Make data-flow merging explicit
- Treat like normal functions

Next Time

Discussion

- Sparse conditional constant propagation paper

Lecture

- More reuse optimizations