Reuse Optimization

Idea

- Eliminate redundant operations in the dynamic execution of instructions

How do redundancies arise?

- Loop invariant code (*e.g.,* index calculation for arrays)
- Sequence of similar operations (*e.g.,* method lookup)
- Lightning frequently strikes twice

Types of reuse optimization

- Value numbering
- Common subexpression elimination
- Partial redundancy elimination

Local Value Numbering

Idea

- Each variable, expression, and constant is assigned a unique number
- When we encounter a variable, expression or constant, see if it's already been assigned a number
  - If so, use the value for that number
  - If not, assign a new number
- Same number ⇒ same value

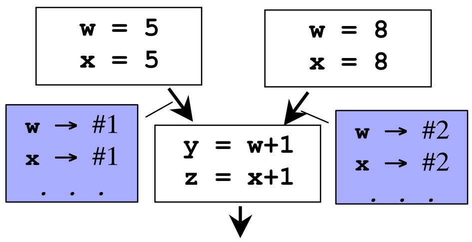
Example

```
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

Global Value Numbering

How do we handle control flow?



Global Value Numbering (cont)

Idea [Alpern, Wegman, and Zadeck 1988]

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

Role of SSA Form

SSA form is helpful

- Allows us to avoid data-flow analysis
- Variables correspond to values

a not congruent to anything

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

a_1 = b
. . .
a_2 = c
. . .
a_3 = d

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

Basis

Idea

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

x and **y** are congruent

```
ta = a
tb = b
x = f(a,b)
y = f(ta,tb)
```

- Use this fact to combine (pessimistic) or split (optimistic) classes

Problem

- This 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

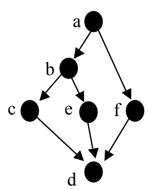
Pessimistic Global Value Numbering

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?

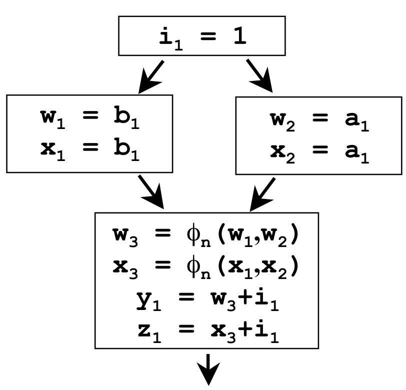
- Ensures that when we consider an assignment statement, we have already considered definitions that reach the RHS operands



Postorder: d, c, e, b, f, a ←

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 it's own congruence class)

Optimistic Global Value Numbering

Idea

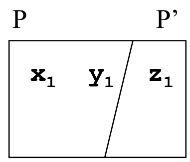
- Initially all variables in one congruence class
- Split congruence classes when evidence of non-congruence arises
  - Variables that are computed using different functions
  - Variables that are computed using functions with non-congruent operands

Splitting

Initially

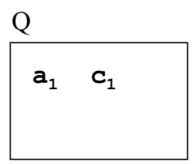
- Variables computed using the same function are placed in the same class

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



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

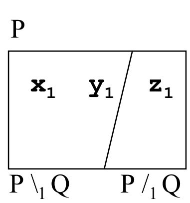


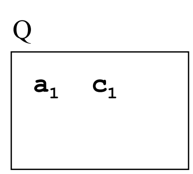
Splitting (cont)

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

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





Algorithm

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

Note: see paper for optimization

Example

SSA code

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

Congruence classes

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

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

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

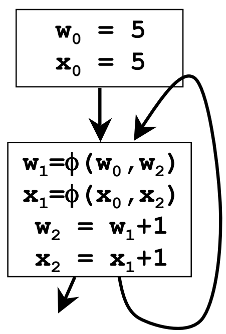
$S_2 \setminus_1 S_0 = \{x_1, z_1\} = S_3$

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

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

- Single def reaches each use
- No data (flow value) at each point

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

Lecture

- Common-Subexpression Elimination

Study Suggestion

- Read Alpern and Zadeck 1992 Chapter about Value Numbering
- Do the examples in Muchnick Section 12.4 and examples in paper using the others algorithm