CS 526

Advanced
Compiler
Construction

http://misailo.cs.Illinois.edu/courses/cs526

STATIC SINGLE ASSIGNMENT

The slides adapted from Vikram Adve

Control Dependence

Def. Postdomination: node *p* postdominates a node *d* if all paths to the exit node of the graph starting at *d* must go through *p*

Def. In a CFG, node Y is control-dependent on node B if

- There is a non-empty path $N_0 = B, N_1, N_2, \ldots, N_k = Y$ such that Y postdominates $N_1 \ldots N_k$, and
- Y does not strictly postdominate B

Def. The Reverse Control Flow Graph (RCFG) of a CFG has the same nodes as CFG and has edge $Y \rightarrow X$ if $X \rightarrow Y$ is an edge in CFG.

Computing Control Dependence

Def. Postdomination: node *p* postdominates a node *d* if all paths to the exit node of the graph starting at *d* must go through *p*

Key observation:

Node **Y is control-dependent on Node X** *iff* $X \in DF(Y)$ in RCFG.

Algorithm:

1. Build RCFG
2. Build dominator tree for RCFG
3. Compute dominance frontiers for RCFG
4. Compute $CD(X) = \{Y \mid X \in DF(Y)\}$.

CD(X) gives the nodes that are control-dependent on X.

*The dominance frontier of node X is the **set of nodes Y** such that **X dominates a predecessor of Y**, but X does not properly dominate Y

DF(X) $= \{Y \mid \exists\, P \in Pred(Y) : X \text{ dom } P \text{ and not } (X \text{ pdom } Y)\}$

Induction Variable Substitution

Auxiliary Induction Variable

An auxiliary induction variable in a loop

```
for (int i = 0; i < n; i++) { … }
```

is any variable `j` that can be expressed as

```
c × i + m
```

at every point where it is used in the loop, where `c` and `m` are loop-invariant values, but `m` may be different at each use.

Induction Variable Substitution

Auxiliary Induction Variable

```
for (int i = 0; i < n; i++) {
    j = 2*i + 1;
    k = -i;
    l = 2*i*i + 1;
    c = c + 5;
}
```

Induction Variable Substitution

Auxiliary Induction Variable

```
for (int i = 0; i < n; i++) {
    j = 2*i + 1;        // Y
    k = -i;             // Y
    l = 2*i*i + 1;      // N
    c = c + 5;          // Y*
}
```

Optimization Goals

Identify linear expression for each auxiliary induction variable

- More effective dependence analysis, loop transformations
- Substitute linear expression in place of every use
- Eliminate expensive or loop-invariant operations from loop

Reminder: Strength Reduction

Goal: Replace expensive operations by cheaper ones

Primitive Operations: Many Examples

n * 2 → n << 1 (similarly, n/2)

n ** 2 → n * n

Recurrences

Example: …=a[i] to (base(a) + (i-1) * 4)

Such recurrences are common in array address calculations

Induction Variable Substitution

Strategy

- Identify operations of the form:

  $$x \leftarrow iv \times c, x \leftarrow iv \pm c$$

  iv: induction variable or another recurrence

  c : loop-invariant variable

- Eliminate **multiplications** from the loop body
- Eliminate induction variable if the **only remaining use** is in the loop **termination test**

Induction Variable Substitution

```
do i = 1 to 100
   sum = sum + a(i)
enddo
```

Source code

```
    sum = 0.0
    i = 1
L:  t1 = i - 1
    t2 = t1 * 4
    t3 = t2 + a
    t4 = load t3
    sum = sum + t4
    i = i + 1
    if (i <= 100) goto L
```

Intermediate code

$$
\begin{array}{ll}
 & \mathtt{sum}_0 = 0.0 \\
 & \mathtt{i}_0 = 1 \\
\mathtt{L:} & \mathtt{sum}_1 = \phi(\mathtt{sum}_0, \mathtt{sum}_2) \\
 & \mathtt{i}_1 = \phi(\mathtt{i}_0, \mathtt{i}_2) \\
 & \mathtt{t1}_0 = \mathtt{i}_1 - 1 \\
 & \mathtt{t2}_0 = \mathtt{t1}_0 * 4 \\
 & \mathtt{t3}_0 = \mathtt{t2}_0 + \mathtt{a} \\
 & \mathtt{t4}_0 = \mathtt{load}\ \mathtt{t3}_0 \\
 & \mathtt{sum}_2 = \mathtt{sum}_1 + \mathtt{t4}_0 \\
 & \mathtt{i}_2 = \mathtt{i}_1 + 1 \\
 & \mathtt{if}\ (\mathtt{i}_2 <= 100)\ \mathtt{goto}\ \mathtt{L}
\end{array}
$$

SSA form

Induction Variable Substitution

$sum_0 = 0.0$
$i_0 = 1$
L: $sum_1 = \phi(sum_0, sum_2)$
$i_1 = \phi(i_0, i_2)$
$t1_0 = i_1 - 1$
$t2_0 = t1_0 * 4$
$t3_0 = t2_0 + a$
$t4_0 = \text{load } t3_0$
$sum_2 = sum_1 + t4_0$
$i_2 = i_1 + 1$
$\text{if } (i_2 <= 100) \text{ goto L}$

SSA form

$sum_0 = 0.0$
$i_0 = 1$
$t5_0 = a$
L: $sum_1 = \phi(sum_0, sum_2)$
$i_1 = \phi(i_0, i_2)$
$t5_1 = \phi(t5_0, t5_2)$
$t4_0 = \text{load } t5_0$
$sum_2 = sum_1 + t4_0$
$i_2 = i_1 + 1$
$t5_2 = t5_1 + 4$
$\text{if } (i_2 <= 100) \text{ goto L}$

After strength reduction

Induction Variable Substitution

```
     sum₀ = 0.0
     i₀ = 1
     t5₀ = a
L:   sum₁ = φ(sum₀,sum₂)
     i₁ = φ(i₀,i₂)
     t5₁ = φ(t5₀,t5₂)
     t4₀ = load t5₀
     sum₂ = sum₁ + t4₀
     i₂ = i₁ + 1
     t5₂ = t5₁ + 4
     if (i₂ <= 100) goto L
```

After strength reduction

```
     sum₀ = 0.0
     t5₀ = a
L:   sum₁ = φ(sum₀,sum₂)
     t5₁ = φ(t5₀,t5₂)
     t4₀ = load t5₀
     sum₂ = sum₁ + t4₀
     t5₂ = t5₁ + 4
     if (t5₂ <= 396 + a) goto L
```

After induction variable substitution

Induction Variable Substitution

```
      sum₀ = 0.0
      t5₀ = a
L:    sum₁ = φ(sum₀,sum₂)
      t5₁ = φ(t5₀,t5₂)
      t4₀ = load t5₀
      sum₂ = sum₁ + t4₀
      t5₂ = t5₁ + 4
      if (t5₂ <= 396 + a) goto L
```

After induction variable substitution

```
      sum₀ = 0.0
      i₀ = 1
L:    sum₁ = φ(sum₀,sum₂)
      i₁ = φ(i₀,i₂)
      t1₀ = i₁ - 1
      t2₀ = t1₀ * 4
      t3₀ = t2₀ + a
      t4₀ = load t3₀
      sum₂ = sum₁ + t4₀
      i₂ = i₁ + 1
      if (i₂ <= 100) goto L
```

SSA form

References

Cocke and Kennedy, CACM 1977 (superseded by the next one).

Allen, Cocke and Kennedy, "Reduction of Operator Strength," In Program Flow Analysis: Theory and Applications, 1981.

Classical Approach

- ACK: Classic algorithm, widely used.
- works on "loops" (Strongly Connected Regions) of flow graph
- uses def-use chains to find induction variables and recurrences

Cooper, Simpson & Vick, 2001, "Operator Strength Reduction," Trans. Prog. Lang. Sys. 23(5), Sept. 2001.

SSA-based algorithm

- Same effectiveness as ACK, but faster and simpler
- Identify induction variables from SCCs in the SSA graph

Value Numbering

- Assign an **identifying number** to each variable / expression / constant:

  x and y have same **id number**

  $\Leftrightarrow$ x = y **for all** inputs

- Use algebraic identities to simplify expressions
- Discover redundant computations and replace them
- Discover constant values, fold & propagate them

Value Numbering

- Use algebraic identities to simplify expressions
  - Commutativity (a+b = b+a), a+b+c = c+b+a, (a+b)^2 = a^2+2ab+b^2…
- Discover redundant computations and replace them
  - E.g., `y=2*x;  z=2*x+1  =>  y=2*x;  z=y+1`
- Discover constant values, fold & propagate them
  - After SCCP: e.g., `x=1;  y = x+1  =>  y = 1+1`
  - Evaluate constant expression (`y = 2`) then propagate

Local Value Numbering

- Each variable, expression, & constant gets a **"value number"** (hash code)

  Same value number ⇒ same value

- **Prerequisites:** low-level intermediate code and existing basic blocks
- Equivalence based solely on facts from within the **single basic block**
- If an instruction's value number is already defined, instr. can be eliminated & subsequent references subsumed
- Constant folding is simple

Local Value Numbering

```
a = x + y
```

V1 ← hash(+,VN[x],VN[y]),
Name[V1] ← a

```
b = x + y
```

hash(+,VN[x],VN[y]) == V1
So, replace x+y with a. Transformed: b = a

```
a = 1
```

Name[V1] ←∅ (can we be more precise?)

```
c = x + y
d = y + x
e = d - 1
f = e + 1
```

Can we replace?

Challenges:
tracking where each value resides
commutativity ⇒ ???
identities (e.g., Vx OR Vx × 1): ⇒
instr. gets value number of operand (Vx)

Local Value Numbering

```
a1 = x + y
```

V1 ← hash(+,VN[x],VN[y]),
Name[V1] ← a

```
b = x + y
```

hash(+,VN[x],VN[y]) == V1
So, replace x+y with a. Transformed: b = a1

```
a2 = 1
c = x + y
d = y + x
e = d - 1
f = e + 1
```

Name[V1] ←∅ (don't need anymore)

c = a1

d = a1

…

Challenges:
tracking where each value resides
commutativity ⇒ ???
identities (e.g., Vx OR Vx × 1): ⇒
instr. gets value number of operand (Vx)

Local Value Numbering

For each instruction i : x ← y op z in the block

    V1 ← VN[y]
    V2 ← VN[z]

    let v = hash(op,V1,V2)
    if (v exists in hash table)
        replace RHS with Name[v]
    else
        enter v in hash table
        VN[x] ← v
        Name[v] ← ti (new temporary)
        replace instruction with: ti ← y op z; x ← ti

Local Value Numbering (LVN)

Simplifications

- If all operands have the same value number i.e. z=x op y, and VN[x] = VN[y]
  - if *op* is MAX, MIN, AND, OR, . . replace *op* with a copy operation (z=x)
  - if *op* tests equality, replace it with z=true
  - if *op* tests inequality replace it with z=false
- if all operands are constants and we haven't already simplified the expression, then immediately evaluate the resulting constant and propagate constants down
- if one operand is constant and we haven't yet simplified the expression:
  - if a constant operand is zero, replace ADD and OR with another operand; replace MULT, AND with zero
  - if constant operand is one, replace MULT with assignment of another operand

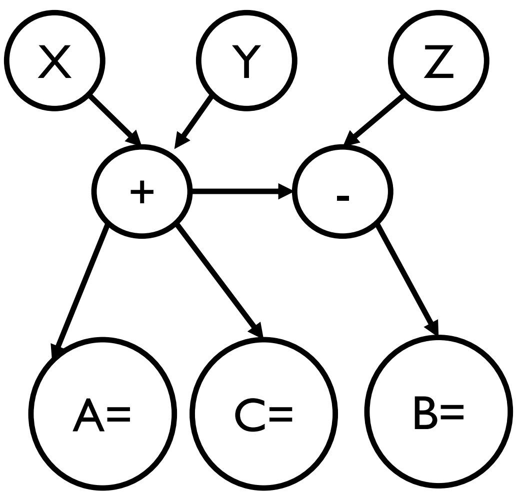
Local VN *Simplifications*

- If the operands have the same value number i.e. z=x op y, and VN[x] = VN[y]
  - if *op* is MAX, MIN, AND, OR, . . replace *op* with a copy operation (z=x)
  - if *op* tests equality, replace it with z=true
  - if *op* tests inequality replace it with z=false
- if all operands (x,y) are constants and we haven't already simplified the expression, then immediately evaluate the resulting constant and propagate constants down
- if one operand is constant and we haven't yet simplified the expression:
  - if a constant operand is zero, replace ADD and OR with another operand; replace MULT, AND with zero
  - if constant operand is one, replace MULT with assignment of another operand
- If *op* commutes, reorder its operands into **ascending order by value number** (canonical form)

Local VN *Analogy*

- Constructing a DAG from a forest (set of trees)
- Each expression is a node in a dag, edges are uses of the expression in the instructions
- Start from the leading instruction of the basic block
- Collapse nodes that are repeated into a single node and connect the edges to all uses

```
a = x + y
b = (x + y) - z
c = y + x
```



Global Value Numbering

```
W = X + Y;
if (...) {
  Z = X + Y;
  X = 1;
} else {
  Z = X + Y - 1;
}


Z = X + Y - 1;   // ??
```

Global Value Numbering

```
W1 = X1 + Y1;
if (...) {
  Z1 = X1 + Y1;
  X2 = 1;
} else {
  Z2 = X1 + Y1 - 1;
}
X3 = Phi(X1, X2)
Z3 = Phi(Z1, Z2)
Z4 = X3 + Y1 - 1;   // ??
```

Global Value Numbering

```
W1 = X + Y;
if (...) {
  Z1 = X + Y;
  W2 = 1;
} else {
  Z2 = X + Y - 1;
}
W3 = Phi(W1, W2)
Z3 = Phi(Z1, Z2)
Z4 = X + Y - 1;   // ??
```

Global Value Numbering

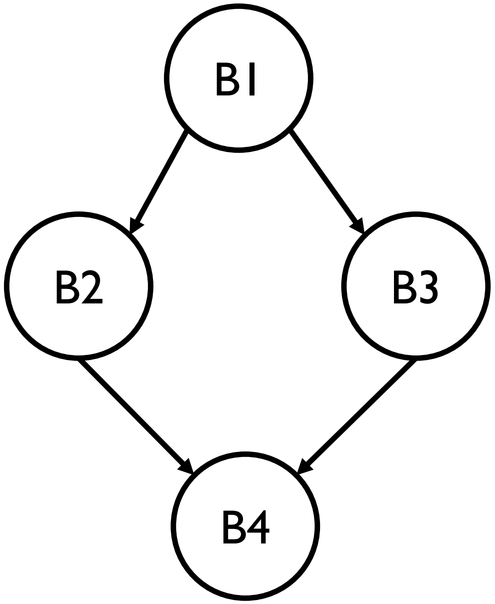
```
T1 = X + Y; W1 = T1;
if (...) {
  Z1 = X + Y;
  W2 = 1;
} else {
  Z2 = X + Y – 1;
}
W3 = Phi(W1, W2)
Z3 = Phi(Z1, Z2)
Z4 = X + Y – 1;   // ??
```

Global Value Numbering

```
T1 = X + Y; W1 = T1;
if (...) {
  Z1 = T1;
  W2 = 1;
} else {
  Z2 = T1 – 1;
}
W3 = Phi(W1, W2)
Z3 = Phi(Z1, Z2)
Z4 = T1 – 1;   // ??
```

Global Value Numbering **(DVTN)**

The Dominator-based VN Technique **(DVNT)**



- B2, B3 can be value-numbered using B1's table
- How about B4? Yes, can use the expressions from B1 (dominator node) but needs to invalidate the expressions killed in B2, B3
- Still based on hashing
- **BUT:** difficult to merge these tables
  - A variable may be redefined in B2, B3, or both

Instruction Congruence

Instructions i & j are **congruent** iff

1. They are the same instruction
2. They are assignments of constants, which are equal (e.g. $x=c_i$, $y=c_j$ and $c_i=c_j$)
3. They have one or multiple operands, e.g.,

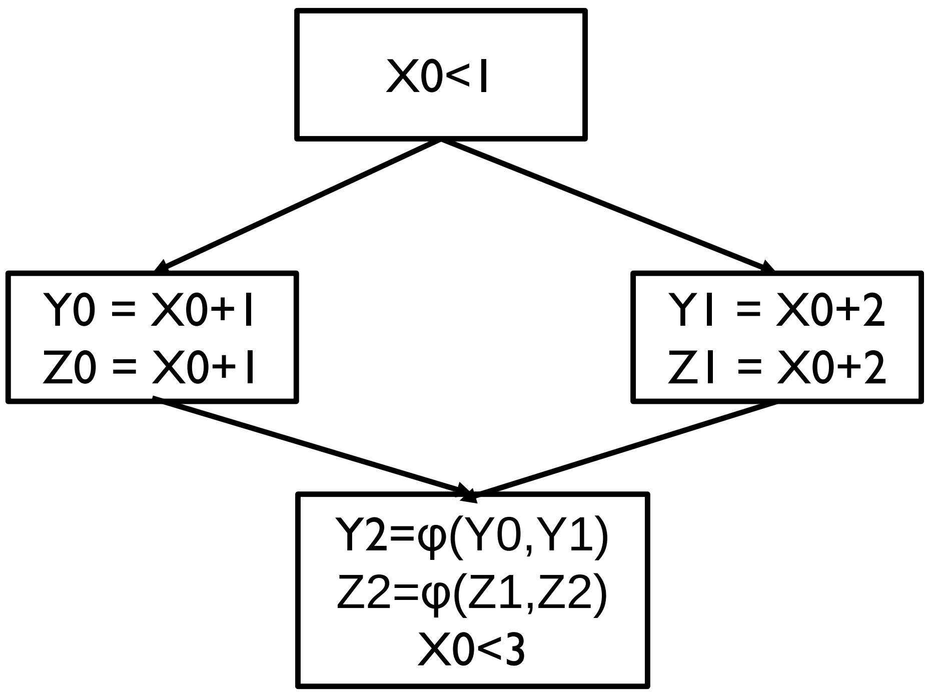
   $z_i = x_i \text{ op } y_i$

   $z_j = x_j \text{ op } y_j$

   with **same** operator and their operands are **congruent** (x_i congruent to x_j and y_i congruent to y_j), taking into consideration commutativity of the op.

   - reflexive, symmetric, transitive

Example



Congruence Classes:

1. (Y0=X0+1, Z0=X0+1)

2. (Y1=X0+2, Z1=X0+2)

3. (Y2=φ(Y0,Y1), Z2=φ(Z1,Z2))

A Global Approach (Alpern, Wegman & Zadeck)

Makes a list of sets of equivalent variables

- Uses congruence

Searches for a maximal fixed point: contains the most equal values

- Optimistic algorithm: assumes all values are equal and then splits them into finer categories

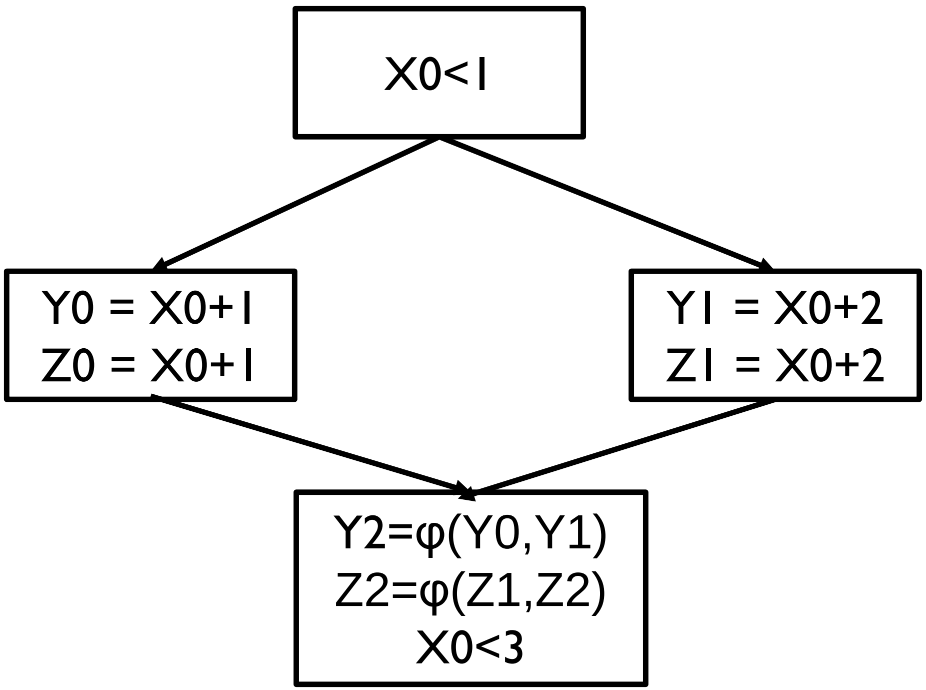
A Global Approach (Alpern, Wegman & Zadeck)

Prerequisite: Computation in SSA Form

Algorithm:

1. partition instructions into congruence classes by opcode
2. *worklist* ← all classes
3. while (*worklist* is not empty)
   a) remove a **class c** from *worklist*
   b) for each **class s** that uses some **x ∈ c**
   while (s != ∅) do
      i. split **s** into **s & s′**: all users of **c** in one class
      ii. put smaller of s or s′ onto *worklist*
4. pick a representative instruction for each partition and perform replacement

Example



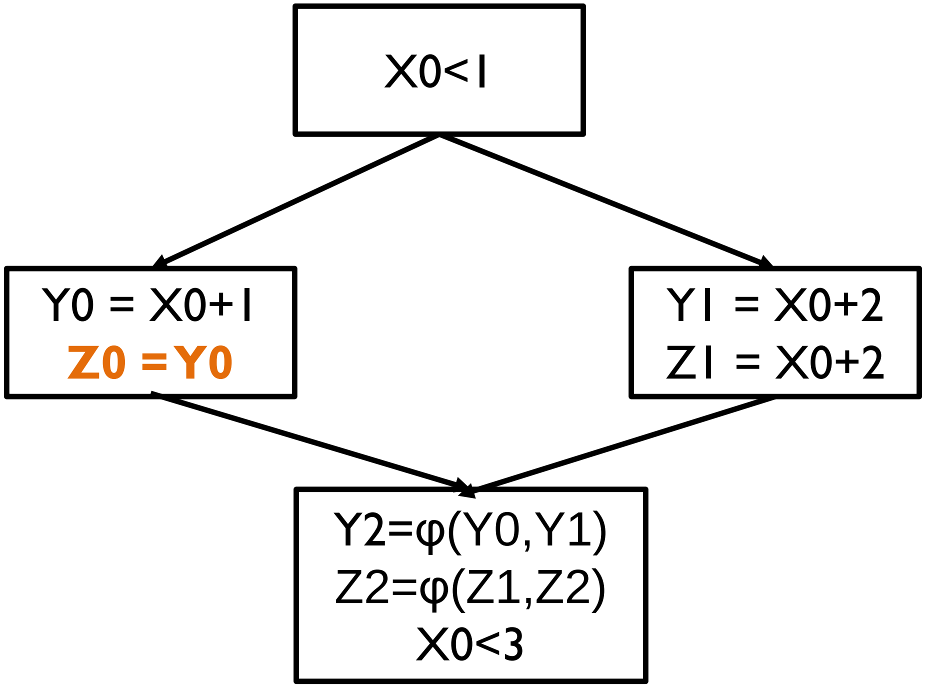
Congruence Classes:

1. (Y0=X0+1, Z0=X0+1)

2. (Y1=X0+2, Z1=X0+2)

3. (Y2=φ(Y0,Y1), Z2=φ(Z1,Z2))

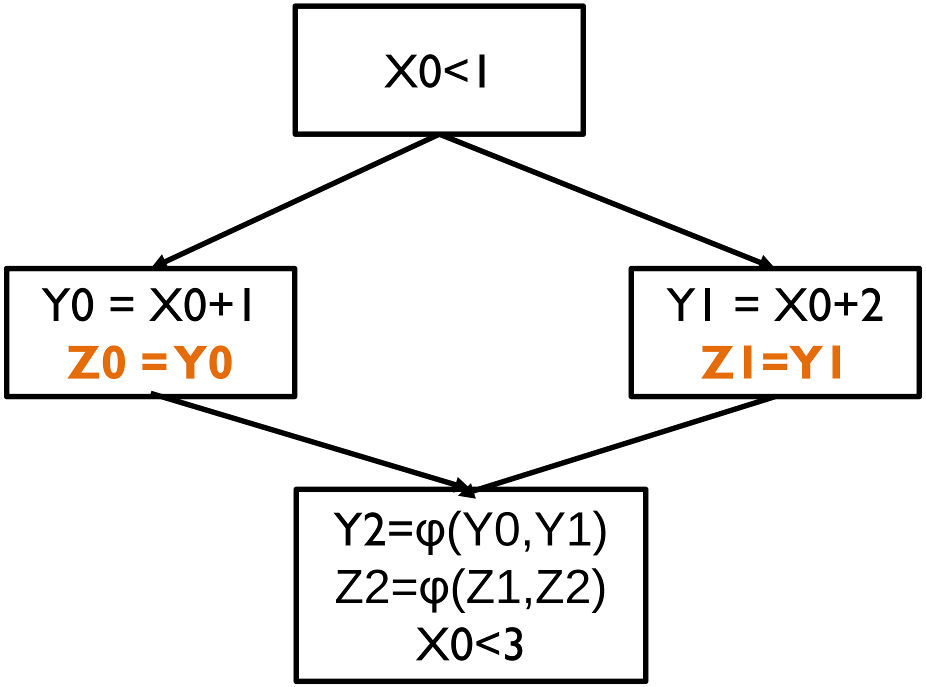
Example



Congruence Classes:

1. (Y0=X0+1, Z0=X0+1)
2. (Y1=X0+2, Z1=X0+2)
3. (Y2=φ(Y0,Y1), Z2=φ(Z1,Z2))

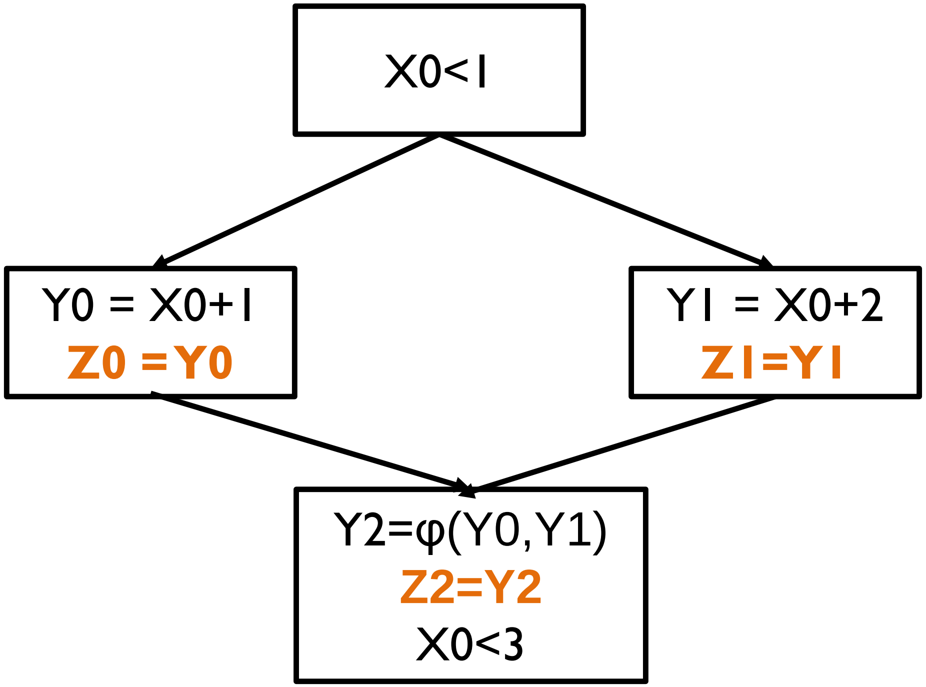
Example



Congruence Classes:

1. (Y0=X0+1, Z0=X0+1)
2. (Y1=X0+2, Z1=X0+2)
3. (Y2=φ(Y0,Y1), Z2=φ(Z1,Z2))

Example



Congruence Classes:

1. $(Y0=X0+1, Z0=X0+1)$
2. $(Y1=X0+2, Z1=X0+2)$
3. $(Y2=\varphi(Y0,Y1), Z2=\varphi(Z1,Z2))$

Properties of the Algorithm

- Cannot prove congruences that involve different operators:
  - $5 \times 2 \sim= 7 + 3$ or
  - $3 + 1 \sim= 2 + 2$ or
  - $x \times 1 \sim= x$
- Need separate pass to transform code (partitioning must complete first)
- Powerful technique but ignores compile-time costs

- Alternative: SCC Based Algorithm (Cooper&Simpson)
  - SCC often beats AWZ in practice

References

Long history in literature

- form of redundancy elimination (compare CSE)
- local version using hashing: late 60's Cocke & Schwartz, 1969
- algorithms for blocks, extended blocks, dominator regions, entire procedures, and (maybe) whole programs
- easy to understand algorithm for single block
- larger scopes cause more complex algorithms

1. Alpern, Wegman & Zadeck, "Detecting Equality of Variables in Programs," *Proceedings POPL* 1988
2. Cooper & Simpson, "SCC-Based Value Numbering," *Rice University TR CRPC-TR95636-S*, 1995.
3. Briggs, Cooper, Simpson, "Value Numbering," *Software–Practice and Experience*, June 1997 (supplementary only).

Optimizations where we will need more information

- Copy Propagation
- Global Common Subexpression Elimination (GCSE)
- Partial Redundancy Elimination (PRE)
- Redundant Load Elimination
- Dead or Redundant Store Elimination
- Code Placement Optimizations