CS 526
Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526
STATIC SINGLE ASSIGNMENT

The slides adapted from Vikram Adve
SSA-Based Optimizations

- Dead Code Elimination (DCE)
- Sparse Conditional Constant Propagation (SCCP)
- Loop-Invariant Code Motion (LICM)
- Global Value Numbering (GVN)
- Strength Reduction of Induction Variables
- Live Range Identification in Register Allocation
Constant Propagation

Goals
Whenever there is a statement of the form $v = \text{Const}$, the uses of $v$ can be replaced by Const.

Safety
Analysis: Explicit propagation of constant expressions
Transformation: Most languages allow removal of computations

Profitability
Fewer computations, almost always

Opportunity
Symbolic constants, conditionally compiled code, …
Simple Constant Propagation

Worklist = All statements in the SSA program

While Worklist ≠ ∅
  
  remove a statement S from Worklist

  if S is “v = ϕ(c1,...,cn)” and c1=...=cn=c (Const),
  replace S with v = c

  if S is “v = c” (c is Const)
  Delete S from the program
  For each Statement T ∈ Uses (v)
  substitute v with c in T
  Worklist = Worklist ∪ {T}
Extensions of the Algorithm

Copy propagation:
- Assignments $x = y$ or $x = \varphi(y)$ can be replaced by a simple use of $y$.

Constant folding:
- Assignments of the form $x = a \odot b$ can be immediately evaluated if $a$ and $b$ are constants, and the statement replaced with $x = c$ ($c = a \odot b$)

Constant conditions:
- If a condition if $(x \odot y)$ always evaluate to true or false, then keep only one branch.
Conditional Constant Propagation: SCCP

Goals
Identify and replace SSA variables with constant values
Delete infeasible branches due to discovered constants

Safety
Analysis: Explicit propagation of constant expressions
Transformation: Most languages allow removal of computations

Profitability
Fewer computations, almost always (except pathological cases)

Opportunity
Symbolic constants, conditionally compiled code, …
Example 1

J = 1;
...
if (J > 0)
    I = 1; // Always produces 1
else
    I = 2;
Example 2

\[ I = 1; \]
\[ \ldots \]
\[ \textbf{while} (\ldots) \{ \]
\[ \quad J = I; \]
\[ \quad I = f(\ldots); \]
\[ \quad \ldots \]
\[ \quad I = J; \quad \text{// Always produces 1} \]
\[ \} \]

We need to proceed with the assumption that everything is constant until proved otherwise.
Example 3

\[ I = 1; \]
\[ \ldots \]
\[ \text{while (...) { } } \]
\[ \quad J = I; \]
\[ \quad I = f(...); \]
\[ \quad \ldots \]
\[ \quad \text{if (J > 0)} \]
\[ \quad \quad I = J; // Always produces 1 \]

For Ex. 1, we could do constant propagation and condition evaluation separately, and repeat until no changes. This separate approach is not sufficient for Ex. 3.
Conditional Constant Propagation*

Advantage:
Simultaneously finds constants + eliminates infeasible branches.

Optimistic
Assume every variable may be constant, until proven otherwise.
( Pessimistic ≡ initially assume nothing is constant. )

Sparse: Only propagates variable values where they are actually used or defined (using def-use chains in SSA form).

Iterative:
Build the list of constant definitions and uses using a worklist algorithm.

- **scpp**: Sparse Conditional Constant Propagation

Sparse conditional constant propagation and merging, which can be summarized as:

- Assumes values are constant unless proven otherwise
- Assumes BasicBlocks are dead unless proven otherwise
- Proves values to be constant, and replaces them with constants
- Proves conditional branches to be unconditional

Note that this pass has a habit of making definitions be dead. It is a good idea to run a DCE pass sometime after running this pass.

* Constant Propagation with Conditional Branches; M. Wegman and K. Zadeck, TOPLAS’91
Dead Code Elimination

The results of the computation are visible through return values or output statements

- We can remove the instructions that do not contribute to the visible outputs

A simple algorithm:
- Compute (or maintain) the def-use chains
- Iterate over the instructions:
  - For $v = x \text{ op } z$, if $v$ has at least one use, mark as live, otherwise mark as dead
- Remove the instructions marked as dead
Aggressive Dead Code Elimination

Ordinary DCE:
- $x_2$ is live because used in def. of $x_1$.
- $x_1$ is live because used in def. of $x_2$.

Yet...

Idea: Analogous to SCCP, be optimistic and assume a statement is dead unless proven otherwise (i.e., contributes to the output)

```
x_1 = \varphi(0, x_2)
x_1 > 100
```

- T
  - $x_2 = x_1 + 2$
- F
  - return 1

---

Transform Passes

This section describes the LLVM Transform Passes.

**-adce: Aggressive Dead Code Elimination**

ADCE aggressively tries to eliminate code. This pass is similar to DCE but it assumes that values are dead until proven otherwise. This is similar to SCCP, except applied to the liveness of values.
Aggressive Dead Code Elimination

In each step, mark a statement as live if:

1. It is the **output** statement (e.g., return)
2. It has **known side effects** (e.g., assignment to global variable or calling a function with side effects)
3. It defines a variable x used by an already live statement
4. It is a **conditional branch**, and some other, already live statement is control dependent on the branch (and its block)

The algorithm then converges to a set of live variables

- Caveat: the algorithm may remove “empty” infinite loops
Loop-Invariant Code Motion

Example:
\[
x = 1; y = 0
\]
\[
\text{while ( } y < 10 \text{ ) } \{
    t = \text{min}(x,2) \\
    y = y + x \\
\}
\]

Becomes:
\[
x = 1; y = 0
\]
\[
t = \text{min}(x,2) \\
\text{while ( } y < 10 \text{ ) } \{
    y = y + x \\
\}
\]

Pattern:
\[
\text{loop } \{
    v = a \text{ op } b \\
    \ldots \text{ v is used here }
\}
\]

Becomes:
\[
v = a \text{ op } b \\
\text{loop } \{
    \ldots \text{ v is used here }
\}
\]

What conditions does the code need to satisfy for this transformation to be sound?
Loop-Invariant Code Motion

Analysis

Conditions for the analysis ($v = a \text{ op } b$):

- Both $a, b$ are constants
  
  ```
  while (b) { v = 2 + 3; /* ... */ }
  ```

- Both $a, b$ are defined before the loop (SSA ensures there is only a single dominating definition for each)
  
  ```
  x = ...;
  while (b) { v = x + 1; /* ... */ }
  ```

- Both $a, b$ are referring to the variables that are in the loop but already determined to be loop invariant
  
  ```
  x = ...;
  while (b) { v = x + 1; t = v * 2; /* ... */ }
  ```

If curious about what complications arise if the program is not in the SSA form see [http://www.cs.cmu.edu/~aplatzer/course/Compilers11/17-loopinv.pdf](http://www.cs.cmu.edu/~aplatzer/course/Compilers11/17-loopinv.pdf)
Loop-Invariant Code Motion

Transformation

**Version 1:** Since the computation is in SSA form, just move it to the node before the header

- What if there is no single such node? (Make it!)
- **Loop preheader:** a single node that dominates the loop header
- How do we ensure there are no side effects?
- What if the loop-invariant computation is expensive?

**Version 2:** Like candidate 1, but add loop’s condition:

```c
if (cond) {
    t = a op b;
    while (cond) { /* ... */ }
}
```

-licm: Loop Invariant Code Motion

This pass performs loop invariant code motion, attempting to remove as much code from the body of a loop as possible. It does this by either hoisting code into the preheader block, or by sinking code to the exit blocks if it is safe. This pass also promotes must-aliased memory locations in the loop to live in registers, thus hoisting and sinking “invariant” loads and stores.
-loop-simplify: Canonicalize natural loops

This pass performs several transformations to transform natural loops into a simpler form, which makes subsequent analyses and transformations simpler and more effective. A summary of it can be found in Loop Terminology, Loop Simplify Form.

Loop pre-header insertion guarantees that there is a single, non-critical entry edge from outside of the loop to the loop header. This simplifies a number of analyses and transformations, such as LICM.

Loop exit-block insertion guarantees that all exit blocks from the loop (blocks which are outside of the loop that have predecessors inside of the loop) only have predecessors from inside of the loop (and are thus dominated by the loop header). This simplifies transformations such as store-sinking that are built into LICM.

This pass also guarantees that loops will have exactly one backedge.

Note that the simplifycfg pass will clean up blocks which are split out but end up being unnecessary, so usage of this pass should not pessimize generated code.

This pass obviously modifies the CFG, but updates loop information and dominator information.

See also: https://llvm.org/docs/LoopTerminology.html
Induction Variable Substitution

Auxiliary Induction Variable

An auxiliary induction variable in a loop

```c
for (int i = 0; i < n; i++) { … }  
```
is any variable \( j \) that can be expressed as

\[ c \times i + m \]

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.
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
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*
}
Reminder: Strength Reduction

**Goal:** Replace expensive operations by cheaper ones

**Primitive Operations:** Many Examples

\[ n \times 2 \rightarrow n \ll 1 \text{ (similarly, } n/2) \]

\[ n^{**2} \rightarrow n \times n \]

**Recurrences**

Example: \( x = a[i] \) to \( x = (\text{base}(a) + (i-1) \times 4) \)

Such recurrences are common in array address calculations
Induction Variable Substitution

Strategy

- Identify operations of the form:
  \[ x \leftarrow iv \times c \quad \text{or} \quad x \leftarrow iv \pm c \]
  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**

-indvars: Canonicalize Induction Variables

This transformation analyzes and transforms the induction variables (and computations derived from them) into simpler forms suitable for subsequent analysis and transformation.
do i = 1 to 100
    sum = sum + a(i)
enddo

**Source code**

```plaintext
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**

```plaintext
sum0 = 0.0
i0 = 1
L:
    sum1 = \phi(sum0, sum2)
    i1 = \phi(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**
Induction Variable Substitution

SSA form

L:
\[
\begin{align*}
\text{sum}_0 &= 0.0 \\
\text{i}_0 &= 1 \\
\text{sum}_1 &= \phi(\text{sum}_0, \text{sum}_2) \\
\text{i}_1 &= \phi(\text{i}_0, \text{i}_2) \\
\text{t}_1 &= \text{i}_1 - 1 \\
\text{t}_2 &= \text{t}_1 \times 4 \\
\text{t}_3 &= \text{t}_2 + \text{a} \\
\text{t}_4 &= \text{load t}_3 \\
\text{sum}_2 &= \text{sum}_1 + \text{t}_4 \\
\text{i}_2 &= \text{i}_1 + 1 \\
\text{if (i}_2 \leq 100\text{) goto L}
\end{align*}
\]

After strength reduction

L:
\[
\begin{align*}
\text{sum}_0 &= 0.0 \\
\text{i}_0 &= 1 \\
\text{t}_5 &= \text{a} \\
\text{sum}_1 &= \phi(\text{sum}_0, \text{sum}_2) \\
\text{i}_1 &= \phi(\text{i}_0, \text{i}_2) \\
\text{t}_5 &= \phi(\text{t}_5, \text{t}_5) \\
\text{t}_4 &= \text{load t}_5 \\
\text{sum}_2 &= \text{sum}_1 + \text{t}_4 \\
\text{i}_2 &= \text{i}_1 + 1 \\
\text{if (i}_2 \leq 100\text{) goto L}
\end{align*}
\]
Induction Variable Substitution

\[
\begin{align*}
\text{sum}_0 &= 0.0 \\
i_0 &= 1 \\
t5_0 &= a \\
L: \quad \text{sum}_1 &= \phi(\text{sum}_0, \text{sum}_2) \\
i_1 &= \phi(i_0, i_2) \\
t5_1 &= \phi(t5_0, t5_2) \\
t4_0 &= \text{load } t5_0 \\
\text{sum}_2 &= \text{sum}_1 + t4_0 \\
i_2 &= i_1 + 1 \\
t5_2 &= t5_1 + 4 \\
\text{if } (i_2 \leq 100) \text{ goto } L
\end{align*}
\]

After strength reduction

\[
\begin{align*}
\text{sum}_0 &= 0.0 \\
t5_0 &= a \\
L: \quad \text{sum}_1 &= \phi(\text{sum}_0, \text{sum}_2) \\
t5_1 &= \phi(t5_0, t5_2) \\
t4_0 &= \text{load } t5_0 \\
\text{sum}_2 &= \text{sum}_1 + t4_0 \\
t5_2 &= t5_1 + 4 \\
\text{if } (t5_2 \leq 396 + a) \text{ goto } L
\end{align*}
\]

After induction variable substitution
Induction Variable Substitution (recap)

\[
\begin{align*}
\text{sum}_0 &= 0.0 \\
\text{t5}_0 &= a \\
L: \quad \text{sum}_1 &= \phi(\text{sum}_0, \text{sum}_2) \\
\text{t5}_1 &= \phi(\text{t5}_0, \text{t5}_2) \\
\text{t4}_0 &= \text{load} \ \text{t5}_0 \\
\text{sum}_2 &= \text{sum}_1 + \text{t4}_0 \\
\text{t5}_2 &= \text{t5}_1 + 4 \\
\text{if} \ (\text{t5}_2 \leq 396 + a) \ \text{goto} \ L
\end{align*}
\]

\[
\begin{align*}
\text{Before induction variable substitution} \\
\text{sum}_0 &= 0.0 \\
i_0 &= 1 \\
L: \quad \text{sum}_1 &= \phi(\text{sum}_0, \text{sum}_2) \\
i_1 &= \phi(i_0, i_2) \\
t_1_0 &= i_1 - 1 \\
t_2_0 &= t_1_0 \ast 4 \\
t_3_0 &= t_2_0 + a \\
t_4_0 &= \text{load} \ \text{t3}_0 \\
\text{sum}_2 &= \text{sum}_1 + \text{t4}_0 \\
i_2 &= i_1 + 1 \\
\text{if} \ (i_2 \leq 100) \ \text{goto} \ L
\end{align*}
\]

\[
\begin{align*}
\text{After induction variable substitution} \\
\text{sum}_0 &= 0.0 \\
i_0 &= 1 \\
L: \quad \text{sum}_1 &= \phi(\text{sum}_0, \text{sum}_2) \\
i_1 &= \phi(i_0, i_2) \\
t_1_0 &= i_1 - 1 \\
t_2_0 &= t_1_0 \ast 4 \\
t_3_0 &= t_2_0 + a \\
t_4_0 &= \text{load} \ \text{t3}_0 \\
\text{sum}_2 &= \text{sum}_1 + \text{t4}_0 \\
i_2 &= i_1 + 1 \\
\text{if} \ (i_2 \leq 100) \ \text{goto} \ L
\end{align*}
\]
References

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

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


SSA-based algorithm

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

Code:

\[
\begin{align*}
a &= x + y \\
b &= x + y \\
a &= 1 \\
c &= x + y \\
d &= y + x \\
e &= d - 1 \\
f &= e + 1
\end{align*}
\]

• **Analysis:** Determining equivalent computations (variables, expressions, consts)

• **Transformation:** Eliminates duplicates with a semantics-preserving optimization

• Form of redundancy elimination
Value Numbering

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

  \[ x \text{ and } y \text{ have same id number} \iff x = y \text{ for all inputs} \]

• Use algebraic identities to simplify expressions

• Discover redundant computations & 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., \(y=2*x; \ z=2*x+1 \Rightarrow y=2*x; \ z=y+1\)

• Discover constant values, fold & propagate them
  • After SCCP: e.g., \(x=1; \ y = x+1 \Rightarrow y = 1+1\)
  • 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

\[ a = x + y \]

\[ V1 \leftarrow \text{hash}(+, \text{VN}[x], \text{VN}[y]), \]
\[ \text{Name}[V1] \leftarrow a \]

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

\[ a = 1 \]
\[ \text{Name}[V1] \leftarrow \emptyset \quad \text{(can we be more precise?)} \]

\[ c = x + y \]
\[ d = y + x \]
\[ e = d - 1 \]
\[ f = e + 1 \]

Challenges:
- tracking where each value resides
- commutativity \( \Rightarrow ??? \)
- identities (e.g., \( Vx \) OR \( Vx \times 1 \)): \( \Rightarrow \)
  - instr. gets value number of operand (Vx)
Local Value Numbering

\[ a_1 = x + y \]
\[ b = x + y \]
\[ a_2 = 1 \]
\[ c = x + y \]
\[ d = y + x \]
\[ e = d - 1 \]
\[ f = e + 1 \]

\[ V1 \leftarrow \text{hash}(+, \text{VN}[x], \text{VN}[y]), \]
\[ \text{Name}[V1] \leftarrow a \]
\[ \text{hash}(+, \text{VN}[x], \text{VN}[y]) == V1 \]
So, replace \( x+y \) with \( a \). Transformed: \( b = a \)

\[ \text{Name}[V1] \leftarrow \emptyset \quad \text{(don't need anymore)} \]

Challenges:
- tracking where each value resides
- commutativity \( \Rightarrow \) ???
- identities (e.g., \( Vx \) OR \( Vx \times 1 \)): \( \Rightarrow \)
  - instr. gets value number of operand (\( Vx \))
Local Value Numbering

\[ a_1 = x + y \]
\[ b_1 = a_1 \]
\[ a_2 = 1 \]
\[ c_1 = a_1 \]
\[ d_1 = a_1 \]
\[ e = d_1 - 1 \]
\[ f = e + 1 \]

\[ V_1 \leftarrow \text{hash}(+, \text{VN}[x], \text{VN}[y]), \]
\[ \text{Name}[V_1] \leftarrow \text{a} \]

hash(+, VN[x], VN[y]) == V_1

So, replace \( x+y \) with \( \text{a} \). Transformed: \( b = a_1 \)

\[ \text{Name}[V_1] \leftarrow \emptyset \quad \text{(don’t need anymore)} \]

Challenges:
What happens with \( e \) and \( f \)?
Local Value Numbering

For each instruction $i : x \leftarrow y \text{ op } z$ in the block

$V1 \leftarrow VN[y]$  
$V2 \leftarrow VN[z]$ 

let $v = \text{hash}(\text{op}, V1, V2)$
if ($v$ exists in hash table)
    replace RHS with $\text{Name}[v]$
else
    enter $v$ in hash table
$VN[x] \leftarrow v$
$\text{Name}[v] \leftarrow \text{ti}$ (new temporary)
replace instruction with: “$\text{ti} \leftarrow y \text{ op } z; x \leftarrow \text{ti}$”
Local VN *Simplifications*

- If the operands have the same value number i.e. $z=x \text{ op } y$, and $\text{VN}[x] = \text{VN}[y]$
  - if $\text{op}$ is MAX, MIN, AND, OR, . . . replace $\text{op}$ with a copy operation ($z=x$)
  - if $\text{op}$ tests equality, replace it with $z=true$
  - if $\text{op}$ 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 $\text{op}$ 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

\[
\begin{align*}
a &= x + y \\
b &= (x + y) - z \\
c &= y + x
\end{align*}
\]
Global Value Numbering

\[ W = X + Y; \]
\[ \text{if (...) { } } \]
\[ \quad Z = X + Y; \]
\[ \quad X = 1; \]
\[ } \text{ else { } } \]
\[ \quad Z = X + Y - 1; \]
\[ } \]

\[ U = X + Y - 1; \quad // ?? \]
Global Value Numbering

\[ W_1 = X_1 + Y_1; \]
\[ \text{if} (\ldots) \{ \]
\[ \quad Z_1 = X_1 + Y_1; \]
\[ \quad X_2 = 1; \]
\[ \} \text{ else } \{ \]
\[ \quad Z_2 = X_1 + Y_1 - 1; \]
\[ \} \]
\[ X_3 = \Phi(X_1, X_2) \]
\[ Z_3 = \Phi(Z_1, Z_2) \]
\[ U_1 = X_3 + Y_1 - 1; \quad // \quad ?? \]
Global Value Numbering

\[ T_1 = X_1 + Y_1; \quad W_1 = T_1; \]

\[ \text{if (...) \{} \]
\[ \quad Z_1 = T_1; \]
\[ \quad X_2 = 1; \]
\[ \} \text{ else \{} \]
\[ \quad Z_2 = T_1 - 1; \]
\[ \} \]

\[ X_3 = \Phi(X_1, X_2) \]
\[ Z_3 = \Phi(Z_1, Z_2) \]
\[ U_1 = X_3 + Y_1 - 1; \quad // \quad ?? \]
Global Value Numbering

\[ W = X + Y; \]
\[ \text{if} (...) \{ \]
\[ \quad Z = X + Y; \]
\[ \quad W = 1 + Z; \]
\[ \} \text{ else } \{ \]
\[ \quad Z = X + Y - 1; \]
\[ \} \]

\[ U = X + Y - 1; \quad // ?? \]
Global Value Numbering

\[
W_1 = X_1 + Y_1;
\]

\[
\text{if} \ (\ldots) \ {\}
\]

\[
\begin{align*}
Z_1 &= X + Y; \\
W_2 &= 1 + Z_1;
\end{align*}
\]

\[
\} \ \text{else} \ {\}
\]

\[
\begin{align*}
Z_2 &= X + Y - 1; \\
\end{align*}
\]

\[
\}
\]

\[
W_3 = \Phi(W_1, W_2)
\]

\[
Z_3 = \Phi(Z_1, Z_2)
\]

\[
U_1 = X_1 + Y_1 - 1; \quad // \quad ??
\]
Global Value Numbering

\[ T_1 = X_1 + Y_1; \quad W_1 = T_1; \]
\[ \text{if (\ldots) } \{ \]
\[ \quad Z_1 = X_1 + Y_1; \]
\[ \quad W_2 = 1 + Z_1; \]
\[ \} \quad \text{else } \{ \]
\[ \quad Z_2 = X_1 + Y_1 - 1; \]
\[ \} \]
\[ W_3 = \Phi(W_1, W_2) \]
\[ Z_3 = \Phi(Z_1, Z_2) \]
\[ U_1 = X_1 + Y_1 - 1; \quad // \quad ?? \]
Global Value Numbering

\[ T_1 = X_1 + Y_1; \quad W_1 = T_1; \]
\[ \text{if (\ldots) } \{ \]
\[ \quad Z_1 = T_1; \]
\[ \quad W_2 = 1 + Z_1; \]
\[ \} \quad \text{else } \{ \]
\[ \quad Z_2 = T_1 - 1; \quad // \; X_1 + Y_1 - 1 \]
\[ \} \]
\[ W_3 = \Phi(W_1, W_2) \]
\[ Z_3 = \Phi(Z_1, Z_2) \]
\[ U_1 = X_1 + Y_1 - 1; \quad // \; ?? \]
Global Value Numbering

\[ T_1 = X_1 + Y_1; \quad W_1 = T_1; \]
if (...) {
    \[ Z_1 = T_1; \]
    \[ W_2 = 1 + Z_1; \]
} else {
    \[ Z_2 = T_1 - 1; \quad // \quad X_1 + Y_1 - 1 \]
}
\[ W_3 = \Phi(W_1, W_2) \]
\[ Z_3 = \Phi(Z_1, Z_2) \]
\[ U_1 = T_1 - 1; \quad // \quad ?? (Done?) \]
Yet another example

\[
X_0 = 1 \\
Y_0 = 1 \\
\text{while} (. . . .) \{ \\
\quad X_1 = \phi(X_0, X_2) \\
\quad Y_1 = \phi(Y_0, Y_2) \\
\quad X_2 = X_1 + 1 \\
\quad Y_2 = Y_1 + 1 \\
\}\]
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 $i$ and $j$ are congruent iff:

1. They are the same instruction, or

2. They are assignments of constants, which are equal (e.g. $x:=c_i$, $y:=c_j$ and $c_i==c_j$), or

3. They have one or multiple operands, e.g.,
   
   $z_i = x_i \text{ op } y_i$
   $z_j = x_j \text{ op } y_j$

   *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 op.
A Global Approach (Alpern, Wegman & Zadeck)

Prerequisite: Computation must be in SSA Form

Algorithm:

1. partition instructions into congruence classes by opcode
2. \textit{worklist} $\leftarrow$ all classes
3. while (\textit{worklist} is not empty)
   a) remove a class \textit{c} from \textit{worklist}
   b) for each class \textit{s} that uses some $x \in c$
      i. split \textit{s} into $s^{c} \& s^{-c}$: all users of \textit{c} are in $s^{c}$ and those that are not are in $s^{-c}$
      ii. if \textit{s} was on the \textit{worklist}, remove \textit{s}, $s^{c}$ and $s^{-c}$
      iii. Else put smaller of $s^{c}$ or $s^{-c}$ onto the \textit{worklist}
4. pick a representative instruction for each partition and perform replacement
Example

Congruence Classes:

1. $(Y_0=X_0+1, Z_0=X_0+1)$
2. $(Y_1=X_0+2, Z_1=X_0+2)$
3. $(Y_2=\varphi(Y_0,Y_1), Z_2=\varphi(Z_0,Z_1))$
Example

Congruence Classes:

1. \((Y_0 = X_0 + 1, Z_0 = X_0 + 1)\)

2. \((Y_1 = X_0 + 2, Z_1 = X_0 + 2)\)

3. \((Y_2 = \varphi(Y_0, Y_1), Z_2 = \varphi(Z_0, Z_1))\)
Example

Congruence Classes:

1. \((Y_0 = X_0 + 1, Z_0 = X_0 + 1)\)
2. \((Y_1 = X_0 + 2, Z_1 = X_0 + 2)\)
3. \((Y_2 = \varphi(Y_0, Y_1), Z_2 = \varphi(Z_0, Z_1))\)
Example

Congruence Classes:

1. \((Y_0 = X_0 + 1, Z_0 = X_0 + 1)\)

2. \((Y_1 = X_0 + 2, Z_1 = X_0 + 2)\)

3. \((Y_2 = \varphi(Y_0, Y_1), Z_2 = \varphi(Z_1, Z_2))\)
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 (see references)
  • 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


-gvn: Global Value Numbering

This pass performs global value numbering to eliminate fully and partially redundant instructions. It also performs redundant load elimination.
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