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 not Worklist = Ø
  remove a statement S from Worklist
  if S has the form v = ϕ(c1, … cn) and c1=…=cn=Const,
      replace S by v = c
  if S is v = 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

\[ J = 1; \]

\[ \ldots \]

\[ \text{if (} J > 0 \text{)} \]

\[ I = 1; \quad // \text{Always produces 1} \]

\[ \text{else} \]

\[ I = 2; \]
Example 2

I = 1;
...
while (...) {
    J = I;
    I = f(...);
    ...
    I = J; // Always produces 1
}
Example 3

I = 1;
...
while (...) {
    J = I;
    I = f(...);
    ...
    if (J > 0)
        I = J; // Always produces 1
}
Conditional Constant Propagation*

**Advantage:**
Simultaneously finds constants + eliminates infeasible branches.

**Optimistic**
Assume every variable may be constant, until proven otherwise.
( Pessimistic $\equiv$ 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.

---

**-sccp:** 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
Before next transformation...
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 $N_0 = B, N_1, N_2, ..., N_k = Y$ 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.

- $p$ is a postdominator of $d$ iff $p$ dominates $d$ in the RCFG.
Computing Control Dependence

**Key observation:** Node Y is control-dependent on B *iff* B ∈ DF(Y) in RCFG.

**Algorithm:**
1. Build RCFG
2. Build dominator tree for RCFG
3. Compute dominance frontiers for RCFG
4. Compute CD(B) = {Y | B ∈ DF(Y)}.

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

<table>
<thead>
<tr>
<th>Transform Passes</th>
</tr>
</thead>
<tbody>
<tr>
<td>This section describes the LLVM Transform Passes.</td>
</tr>
<tr>
<td><strong>-adce:</strong> Aggressive Dead Code Elimination</td>
</tr>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>
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
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
Induction Variable Substitution

Auxiliary Induction Variable

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

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

\[ n \times 2^2 \rightarrow n \times n \]

**Recurrences**

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

Such recurrences are common in array address calculations
Induction Variable Substitution

Strategy

• Identify operations of the form:
  \[ x \leftarrow \text{iv} \times c, \ x \leftarrow \text{iv} \pm c \]
  iv: induction variable or another recurrence
  c : loop-invariant variable

• Eliminate multiplic\textit{ations} 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)
endo

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

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
**Induction Variable Substitution**

<table>
<thead>
<tr>
<th>SSA form</th>
<th>After strength reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sum}_0 = 0.0$</td>
<td>$\text{sum}_0 = 0.0$</td>
</tr>
<tr>
<td>$i_0 = 1$</td>
<td>$i_0 = 1$</td>
</tr>
<tr>
<td>$\text{sum}_1 = \phi(\text{sum}_0, \text{sum}_2)$</td>
<td>$\text{sum}_1 = \phi(\text{sum}_0, \text{sum}_2)$</td>
</tr>
<tr>
<td>$i_1 = \phi(i_0, i_2)$</td>
<td>$i_1 = \phi(i_0, i_2)$</td>
</tr>
<tr>
<td>$t_1 = i_1 - 1$</td>
<td>$t_5 = a$</td>
</tr>
<tr>
<td>$t_2 = t_1 * 4$</td>
<td>$\text{sum}_1 = \phi(\text{sum}_0, \text{sum}_2)$</td>
</tr>
<tr>
<td>$t_3 = t_2 + a$</td>
<td>$i_1 = \phi(i_0, i_2)$</td>
</tr>
<tr>
<td>$t_4 = \text{load } t_3$</td>
<td>$t_5 = \phi(t_5_0, t_5_2)$</td>
</tr>
<tr>
<td>$\text{sum}_2 = \text{sum}_1 + t_4$</td>
<td>$t_4 = \text{load } t_5_0$</td>
</tr>
<tr>
<td>$i_2 = i_1 + 1$</td>
<td>$\text{sum}_2 = \text{sum}_1 + t_4$</td>
</tr>
<tr>
<td>$\text{if } (i_2 \leq 100) \text{ goto } L$</td>
<td>$i_2 = i_1 + 1$</td>
</tr>
<tr>
<td></td>
<td>$t_5_2 = t_5_1 + 4$</td>
</tr>
<tr>
<td></td>
<td>$\text{if } (i_2 \leq 100) \text{ goto } L$</td>
</tr>
</tbody>
</table>
**Induction Variable Substitution**

\[
\begin{align*}
\text{sum}_0 &= 0.0 \\
i_0 &= 1 \\
t_{50} &= a \\
L: & \quad \text{sum}_1 = \phi(\text{sum}_0, \text{sum}_2) \\
i_1 &= \phi(i_0, i_2) \\
t_{51} &= \phi(t_{50}, t_{52}) \\
t_{40} &= \text{load } t_{50} \\
\text{sum}_2 &= \text{sum}_1 + t_{40} \\
i_2 &= i_1 + 1 \\
t_{52} &= t_{51} + 4 \\
\text{if } (i_2 \leq 100) \text{ goto } L
\end{align*}
\]

**After strength reduction**

\[
\begin{align*}
\text{sum}_0 &= 0.0 \\
t_{50} &= a \\
L: & \quad \text{sum}_1 = \phi(\text{sum}_0, \text{sum}_2) \\
t_{51} &= \phi(t_{50}, t_{52}) \\
t_{40} &= \text{load } t_{50} \\
\text{sum}_2 &= \text{sum}_1 + t_{40} \\
t_{52} &= t_{51} + 4 \\
\text{if } (t_{52} \leq 396 + a) \text{ goto } L
\end{align*}
\]

**After induction variable substitution**
Induction Variable Substitution (recap)

After induction variable substitution

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

Before induction variable substitution

\[
\begin{align*}
\text{sum}_0 &= 0.0 \\
\text{i}_0 &= 1 \\
\text{L:} & \quad \text{sum}_1 = \phi(\text{sum}_0, \text{sum}_2) \\
& \quad \text{i}_1 = \phi(\text{i}_0, \text{i}_2) \\
& \quad \text{t1}_0 = \text{i}_1 - 1 \\
& \quad \text{t2}_0 = \text{t1}_0 \times 4 \\
& \quad \text{t3}_0 = \text{t2}_0 + a \\
& \quad \text{t4}_0 = \text{load} \ \text{t3}_0 \\
& \quad \text{sum}_2 = \text{sum}_1 + \text{t4}_0 \\
& \quad \text{i}_2 = \text{i}_1 + 1 \\
& \quad \text{if} \ (\text{i}_2 \leq 100) \ \text{goto} \ \text{L}
\end{align*}
\]
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 \((a+b = b+a), \ a+b+c = c+b+a, \ (a+b)^2 = a^2+2ab+b^2\ldots\)

• Discover redundant computations and replace them
  • E.g., \(y=2\times x; \ z=2\times x+1 \Rightarrow y=2\times x; \ z=y+1\)

• 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

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

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

\[
\text{hash}(+, VN[x], VN[y]) == V1
\]

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

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

Can we replace?

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

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

\[
V1 \leftarrow \text{hash}(+, VN[x], VN[y]), \\
\text{Name}[V1] \leftarrow a \\
\text{hash}(+, VN[x], VN[y]) == V1 \\
\text{So, replace } x+y \text{ with } a. \text{ Transformed: } b = a_1 \\
\text{Name}[V1] \leftarrow \emptyset \quad \text{(don’t need anymore)} \\
\]

Challenges:
tracking where each value resides
commutativity ⇒ ???
identities (e.g., Vx OR Vx × 1): ⇒
\text{instr. gets value number of operand } (Vx)
Local Value Numbering

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

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

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

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

\[ b = a_1 \]
\[ c = a_1 \]
\[ d = a_1 \]

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

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

\[
\begin{align*}
V1 & \leftarrow VN[y] \\
V2 & \leftarrow VN[z] \\
\end{align*}
\]

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

Simplifications

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

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

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

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

\[
W_1 = X_1 + Y_1;
\]
\[
\text{if} \ (\ldots) \ {\}
\]
\[
\quad Z_1 = X_1 + Y_1;
\]
\[
\quad X_2 = 1;
\]
\[
\} \ \text{else} \ {\}
\]
\[
\quad Z_2 = X_1 + Y_1 - 1;
\]
\[
\}
\]
\[
X_3 = \Phi(X_1, X_2)
\]
\[
Z_3 = \Phi(Z_1, Z_2)
\]
\[
Z_4 = X_3 + Y_1 - 1; \quad // \ ??
\]
Global Value Numbering

\[ W_1 = X_1 + Y_1; \]
\[ \text{if (...) \{ } \]
\[ \quad Z_1 = X + Y; \]
\[ \quad W_2 = 1; \]
\[ \} \text{ else \{ } \]
\[ \quad Z_2 = X + Y - 1; \]
\[ \} \]
\[ W_3 = \Phi(W_1, W_2) \]
\[ Z_3 = \Phi(Z_1, Z_2) \]
\[ Z_4 = X_1 + Y_1 - 1; \quad // \quad ?? \]
Global Value Numbering

\[ T_1 = X_1 + Y_1; \ W_1 = T_1; \]
\[ \text{if (\ldots)} \{ \]
\[ \quad Z_1 = X_1 + Y_1; \]
\[ \quad W_2 = 1; \]
\[ \} \ \text{else} \{ \]
\[ \quad Z_2 = X_1 + Y_1 - 1; \]
\[ \} \]
\[ W_3 = \Phi(W_1, W_2) \]
\[ Z_3 = \Phi(Z_1, Z_2) \]
\[ Z_4 = X_1 + Y_1 - 1; \quad \text{// ??} \]
Global Value Numbering

\[ T1 = X1 + Y1; \quad W1 = T1; \]
if (...) {
  \[ Z1 = T1; \]
  \[ W2 = 1; \]
} else {
  \[ Z2 = T1 - 1; \quad // \quad X1 + Y1 - 1 \]
}
\[ W3 = \text{Phi}(W1, W2) \]
\[ Z3 = \text{Phi}(Z1, Z2) \]
\[ Z4 = X1 + Y1 - 1; \quad // \quad ?? \]
Global Value Numbering

\[ T_1 = X_1 + Y_1; \quad W_1 = T_1; \]
if (...) {
  \[ Z_1 = T_1; \]
  \[ W_2 = 1; \]
} else {
  \[ Z_2 = T_1 - 1; \quad // \quad X_1 + Y_1 - 1 \]
}
\[ W_3 = \Phi(W_1, W_2) \]
\[ Z_3 = \Phi(Z_1, Z_2) \]
\[ Z_4 = T_1 - 1; \quad // \quad ?? \]
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
Example – Need for Global VN

\[ X_0 = 1 \]
\[ Y_0 = 1 \]

while (\ldots) {
    \[ X_1 = \phi(X_0, X_2) \]
    \[ Y_1 = \phi(Y_0, Y_2) \]
    \[ X_2 = X_1 + 1 \]
    \[ Y_2 = Y_1 + 1 \]
}
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.,
   \[
   z_i = x_i \text{ op } y_i \\
   z_j = x_j \text{ op } y_j
   \]
   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 in SSA Form

Algorithm:

1. partition instructions into congruence classes by opcode
2. \( \text{worklist} \leftarrow \text{all classes} \)
3. while (\( \text{worklist} \) is not empty)
   a) remove a class \( c \) from \( \text{worklist} \)
   b) for each class \( s \) that uses some \( x \in c \)
      while (\( s \neq \emptyset \)) do
         i. split \( s \) into \( s \) & \( s' \): all users of \( c \) in one class
         ii. put smaller of \( s \) or \( s' \) onto \( \text{worklist} \)
4. pick a representative instruction for each partition and perform replacement
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


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