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 $N_0 = B, N_1, N_2, ..., N_k = Y$ 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 \( Y \).*

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

Auxiliary Induction Variable
An auxiliary induction variable in a loop

```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.
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

```c
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

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

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

**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 \textbf{multiplications} from the loop body

• Eliminate induction variable if the \textbf{only remaining use} is in the loop \textbf{termination test}
Induction Variable Substitution

\[
\begin{align*}
\text{do } i &= 1 \text{ to } 100 \\
\quad \text{sum} &= \text{sum} + a(i) \\
\text{enddo}
\end{align*}
\]

Source code

\[
\begin{align*}
\text{sum} &= 0.0 \\
\text{i} &= 1 \\
\text{L:} \\
\quad \text{t1} &= \text{i} - 1 \\
\quad \text{t2} &= \text{t1} \times 4 \\
\quad \text{t3} &= \text{t2} + a \\
\quad \text{t4} &= \text{load t3} \\
\quad \text{sum} &= \text{sum} + \text{t4} \\
\quad \text{i} &= \text{i} + 1 \\
\quad \text{if (i } \leq 100) \text{ goto L}
\end{align*}
\]

Intermediate code

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

SSA form
Induction Variable Substitution

SSA form

\[
\begin{align*}
\text{sum}_0 &= 0.0 \\
\text{i}_0 &= 1 \\
\text{sum}_1 &= \phi(\text{sum}_0, \text{sum}_2) \\
\text{i}_1 &= \phi(\text{i}_0, \text{i}_2) \\
\text{t}10 &= \text{i}_1 - 1 \\
\text{t}20 &= \text{t}10 \times 4 \\
\text{t}30 &= \text{t}20 + a \\
\text{t}40 &= \text{load t}30 \\
\text{sum}_2 &= \text{sum}_1 + \text{t}40 \\
\text{i}_2 &= \text{i}_1 + 1 \\
\text{if (i}_2 \leq 100) \text{ goto L}
\end{align*}
\]

After strength reduction

\[
\begin{align*}
\text{sum}_0 &= 0.0 \\
\text{i}_0 &= 1 \\
\text{t}50 &= \alpha \\
\text{sum}_1 &= \phi(\text{sum}_0, \text{sum}_2) \\
\text{i}_1 &= \phi(\text{i}_0, \text{i}_2) \\
\text{t}51 &= \phi(\text{t}50, \text{t}52) \\
\text{t}40 &= \text{load t}50 \\
\text{sum}_2 &= \text{sum}_1 + \text{t}40 \\
\text{i}_2 &= \text{i}_1 + 1 \\
\text{t}52 &= \text{t}51 + 4 \\
\text{if (i}_2 \leq 100) \text{ goto L}
\end{align*}
\]
Induction Variable Substitution

After 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

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

L: sum1 = \phi(sum0,sum2)
t51 = \phi(t50,t52)
t40 = \text{load } t50
sum2 = sum1 + t40
t52 = t51 + 4
if (t52 <= 396 + a) goto L

After induction variable substitution

SSA form
References


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

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

• 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

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

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

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

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

Can we replace?

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

\[ a_1 = x + y \]
\[ \text{VI} \leftarrow \text{hash}(+, \text{VN}[x], \text{VN}[y]), \]
\[ \text{Name}[\text{VI}] \leftarrow a \]
\[ b = x + y \]
\[ \text{hash}(+, \text{VN}[x], \text{VN}[y]) == \text{VI} \]
\[ \text{So, replace } x+y \text{ with } a. \text{Transformed: } b = a_1 \]
\[ a_2 = 1 \]
\[ c = x + y \]
\[ c = a_1 \]
\[ d = y + x \]
\[ d = a_1 \]
\[ e = d - 1 \]
\[ f = e + 1 \]

Challenges:
- tracking where each value resides
- commutativity \(\Rightarrow \) ???
- identities (e.g., \(Vx \text{ OR } Vx \times 1\)): \(\Rightarrow\)
  - instr. gets value number of operand (Vx)
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 $v = \text{hash}(\text{op}, V1, V2)$

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. \( z=x \ op \ y \), 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 \( \text{op} \) is \( \text{MAX}, \text{MIN}, \text{AND}, \text{OR} \), . . . replace \( \text{op} \) with a copy operation \( (z=x) \)
  - if \( \text{op} \) tests equality, replace it with \( z=\text{true} \)
  - if \( \text{op} \) 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 \( \text{op} \) 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

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

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

\[ W1 = X1 + Y1; \]
\[ \text{if (\ldots)} \{ \]
\[ \quad Z1 = X1 + Y1; \]
\[ \quad X2 = 1; \]
\[ \} \text{ else } \{ \]
\[ \quad Z2 = X1 + Y1 - 1; \]
\[ \} \]
\[ X3 = \text{Phi}(X1, X2) \]
\[ Z3 = \text{Phi}(Z1, Z2) \]
\[ Z4 = X3 + Y1 - 1; \quad // \quad ?? \]
Global Value Numbering

\[
W_1 = X + Y;
\]

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

\[
Z_1 = X + Y;
\]

\[
W_2 = 1;
\]

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

\[
Z_2 = X + Y - 1;
\]

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

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

\[
Z_4 = X + Y - 1; \quad // \text{ ??}
\]
Global Value Numbering

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

\[ T_1 = X + Y; \quad W_1 = T_1; \]
\[ \text{if (...) \{ \quad Z_1 = T_1; \quad W_2 = 1; \}} \]
\[ \text{\} else \{ \quad Z_2 = T_1 - 1; \}} \]
\[ W_3 = \Phi(W_1, W_2) \]
\[ Z_3 = \Phi(Z_1, Z_2) \]
\[ Z_4 = T_1 - 1; \quad // \quad ?? \]
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
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. \((Y_0 = X_0 + 1, \ Z_0 = X_0 + 1)\)
2. \((Y_1 = X_0 + 2, \ Z_1 = X_0 + 2)\)
3. \((Y_2 = \varphi(Y_0, Y_1), \ Z_2 = \varphi(Z_1, Z_2))\)
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. \((Y_0 = X_0 + 1, Z_0 = X_0 + 1)\)

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

3. \((Y_2 = \varphi(Y_0, Y_1), Z_2 = \varphi(Z_1, Z_2))\)
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. $(Y_2 = \varphi(Y_0, Y_1), Z_2 = \varphi(Z_1, Z_2))$
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. \((Y_2 = \varphi(Y_0, Y_1), Z_2 = \varphi(Z_1, Z_2))\)
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. \((Y_2 = \phi(Y_0, Y_1), Z_2 = \phi(Z_1, Z_2))\)
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 (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


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