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 \( \text{Const} \).

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 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;
...
if (J > 0)
    I = 1; // Always produces 1
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 (T), 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).

SSA vs. intentionally constructed def-use graphs
Much faster: SSA graph has fewer edges than def-use graph
The original paper claims SSA catches more constants (not convincing)
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

$$c \times i + m$$

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

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

\[
\begin{align*}
n \times 2 & \rightarrow n \ll 1 \; \text{(similarly, } n/2) \\
n \times \times 2 & \rightarrow n \times n
\end{align*}
\]

**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:
  \[ x \leftarrow iv \times c, x \leftarrow iv \pm c \]
  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

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

SSA form

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
Induction Variable Substitution

Induction Variable Substitution

\[
\begin{align*}
\text{sum}_0 &= 0.0 \\
i_0 &= 1 \\
\text{sum}_1 &= \phi(\text{sum}_0, \text{sum}_2) \\
i_1 &= \phi(i_0, i_2) \\
t_{10} &= i_1 - 1 \\
t_{20} &= t_{10} * 4 \\
t_{30} &= t_{20} + a \\
t_{40} &= \text{load } t_{30} \\
\text{sum}_2 &= \text{sum}_1 + t_{40} \\
i_2 &= i_1 + 1 \\
\text{if } (i_2 \leq 100) \text{ goto } L
\end{align*}
\]

SSA form

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

After strength reduction
Induction Variable Substitution

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

After strength reduction

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

After induction variable substitution
**Induction Variable Substitution**

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

**After induction variable substitution**

\[
\begin{align*}
\text{sum}_0 &= 0.0 \\
i_0 &= 1 \\
\text{sum}_1 &= \phi(\text{sum}_0, \text{sum}_2) \\
i_1 &= \phi(i_0, i_2) \\
\text{t1}_0 &= i_1 - 1 \\
\text{t2}_0 &= \text{t1}_0 \times 4 \\
\text{t3}_0 &= \text{t2}_0 + a \\
\text{t4}_0 &= \text{load t3}_0 \\
\text{sum}_2 &= \text{sum}_1 + \text{t4}_0 \\
i_2 &= i_1 + 1 \\
\text{if (i}_2 \leq 100) \text{ goto L}
\end{align*}
\]

**SSA form**
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

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

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

• 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

\[ a = x + y \]

\[ b = x + y \]

\[ a = 1 \]

\[ c = x + y \]

\[ d = y + x \]

\[ e = d - 1 \]

\[ f = e + 1 \]

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

\[ \text{hash}(+, VN[x], VN[y]) == V_1 \]

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

\[ \text{Name}[V_1] \leftarrow \emptyset \] (can we be more precise?)

Can we replace?

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

\[ a_1 = x + y \]  \hspace{1cm} VI \leftarrow \text{hash}(+, \text{VN}[x], \text{VN}[y]), \\
\text{Name}[VI] \leftarrow a

\[ b = x + y \]  \hspace{1cm} \text{hash}(+, \text{VN}[x], \text{VN}[y]) = VI \\
\text{So, replace } x+y \text{ with } a. \text{ Transformed: } b = a_1

\[ a_2 = 1 \] \hspace{1cm} \text{Name}[VI] \leftarrow \emptyset \quad \text{(don't need anymore)}

\[ c = x + y \]  \hspace{1cm} c = a_1

\[ d = y + x \]  \hspace{1cm} d = a_1

\[ e = d - 1 \] \hspace{1cm} \ldots

\[ f = e + 1 \]

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

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

$V1 ← VN[y]$

$V2 ← VN[z]$

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

$VN[x] ← v$

$\text{Name}[v] ← \text{ti}$ (new temporary)

replace instruction with: $\text{ti} ← y \text{ op } z; x ← \text{ti}$
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 \textit{Simplifications}

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

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

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

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

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

\[ T1 = X + Y; \quad W1 = T1; \]
if (...) {
  \[ Z1 = X + Y; \]
  \[ W2 = 1; \]
} else {
  \[ Z2 = X + Y - 1; \]
}
\[ W3 = \Phi(W1, W2) \]
\[ Z3 = \Phi(Z1, Z2) \]
\[ Z4 = X + Y - 1; \quad // ?? \]
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 ( . . . . ) {
    \[ X_1 = \phi(X_0, X_2) \]
    \[ Y_1 = \phi(Y_0, Y_2) \]
    \[ X_2 = X_1 + 1 \]
    \[ Y_2 = Y_1 + 1 \]
}


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. \textit{worklist} ← 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 \(x \in c\)
      while (\textit{s} \neq \emptyset) do
         i. split \textit{s} into \textit{s} \& \textit{s'}: all users of \textit{c} in one class
         ii. put smaller of \textit{s} or \textit{s'} onto \textit{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