Objectives

- Understand why conditional constant propagation is needed
- Apply the SCCP algorithm to propagate constants
- Understand loop induction variable optimizations
- Get an intuition as to how the Induction variable optimization works

Static Single Assignment-based Optimization

Widely Used SSA-Based Optimizations

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

Static Single Assignment-based Optimization(cont)

Dataflow optimizations for which SSA is insufficient

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

Sparse Conditional Constant Propagation: SCCP

Read this paper if you have not read it before:

Wegman and Zadeck, *Constant Propagation With Conditional Branches*, TOPLAS 1991.

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, simple ICG, ...

SCCP: Key Algorithm Strengths

Conditional Constant Propagation

Simultaneously finds constants + eliminates infeasible branches.

Optimistic

Assume every variable may be constant ($\top$), until proven otherwise.
Pessimistic $\equiv$ initially assume nothing is constant ($\perp$).

Sparse

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

SSA vs. def-use chains

Much faster: SSA graph has fewer edges than def-use graph

Paper claims SSA catches more constants (not convincing)

SCCP Examples

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.

Example 1: Needs Condition Evaluation (can be done separately)

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

Example 2: Needs "Optimistic" initial assumption

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

SCCP Examples

Example 3: Needs simultaneous condition evaluation + constant propagation

```
I = 1;
...
while (...) {
    J = I;
    I = f(...);
    ...
    if (J > 0) I = J;       // Always produces 1
}
```

Repeatedly doing constant propagation and condition evaluation separately will not prove I or J constant.

CONST Lattice and Example

Lattice L

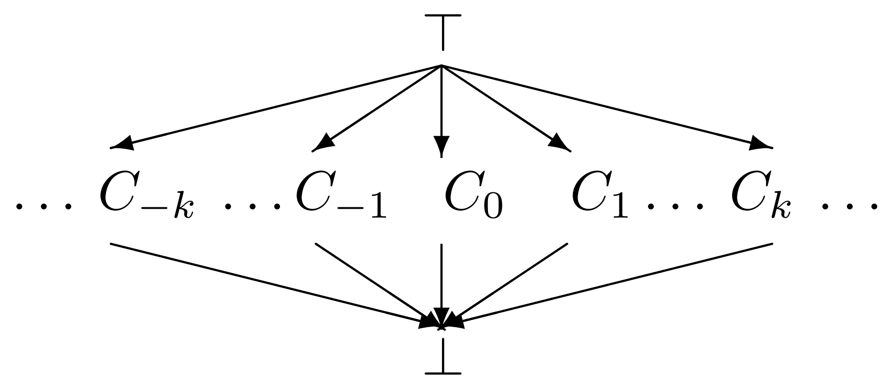
Lattice $L \equiv \{\top, C_i, \perp\}$.
$\top$ intuitively means "*May be constant.*"
$\perp$ intuitively means "*Not constant.*"

Meet Operator, $\sqcap$

$$
\begin{aligned}
\top \sqcap X &= X, \ \ \forall X \in L \\
\perp \sqcap X &= \perp, \ \ \forall X \in L \\
C_i \sqcap C_j &= \begin{cases} C_i, & \textit{iff } i = j, \\ \perp, & \textit{otherwise} \end{cases}
\end{aligned}
$$

Intuition: A Partial Order $\prec$

$\perp \ \prec \ C_i$ for any C_i.
$C_i \ \prec \ \top$ for any C_i.
$C_i \ \not\prec \ C_j$ (i.e., no ordering).



Meet of X and Y ($X \sqcap Y$) is the greatest value $\preceq$ both X and Y.

SCCP Overview

Assume:

- Only assignment or branch statements
- Every non-ϕ statement is in separate BB

Key Ideas

1. Constant propagation lattice = { $\top, C_i, \bot$ }
2. *Initially*: every def. has value $\top$ ("may be constant").
   Initially: every CFG edge is infeasible, except edges from s
3. Use 2 worklists: FlowWL, SSAWL
   (a) FlowWL: insert CFG edge if potentially executable
   (b) SSAWL: insert SSA edge if value of def. changes
4. *Highlights*:
   - Visit S only if some incoming edge is executable
   - Ignore ϕ argument if incoming CFG edge not executable
   - If variable changes value, add SSA out-edges to SSAWL
   - If CFG edge executable, add to FlowWL

High-Level SCCP Algorithm (1 of 2)

SCCP()

```
    Initialize(ExecFlags[], LatCell[], FlowWL, SSAWL);
    while ((Edge E = GetEdge(FlowWL ∪ SSAWL)) != 0)

        if (E is a flow edge && ExecFlag[E] == false)
            ExecFlag[E] = true
            VisitPhi(φ) ∀ φ ∈ E->sink
            if (first visit to E->sink via flow edges)
                VisitInst(E->sink)
            if (E->sink has only one outgoing flow edge E_out)
                add E_out to FlowWL
        else if (E is an SSA edge)
            if (E->sink is a φ node)
                VisitPhi(E->sink)
            else if (E->sink has 1 or more executable in-edges)
                VisitInst(E->sink)
```

High-Level SCCP Algorithm (2 of 2)

VisitPhi(ϕ) :

```
for (all operands Uk of φ)
    if (ExecFlag[InEdge(k)] == true)
        LatCell(φ) ⊓= LatCell(Uk)
        if (LatCell(φ) changed)
            add SSAOutEdges(φ) to SSAWL
```

VisitInst(S) :

Many errors in Muchnick

```
val = Evaluate(S)
LatCell(S) = val
if (LatCell(S) changed) // cannnot be Top
    if (S is Assignment)
        add SSAOutEdges(S) to SSAWL
    else // S must be a Branch
        Add one or both outgoing edges to FlowWL
```

SCCP Example

Example 3: Needs simultaneous condition evaluation + constant propagation

```
S:       // entry BB is empty
B0:      I0 = 1
B1:      if (I0 < N0)
B2:          I1 = φ(I0, I4)
             J0 = I1
B3:          I2 = f(...)
B4:          if (J0 > 0)
B5:              { I3 = J0 }
B6:          I4 = φ(I2, I3)
             if (I4 < N0)
B7:              goto B1
B8:      ...
```

SCCP Example

Some Steps of SCCP Algorithm

Edge	Call	LatVal	Edges Inserted
(1) $S \rightarrow B0$	VisitInst(I_0)	$I_0 = 1$	$I_0 \rightarrow$ if, $I_0 \rightarrow I_1$, $B0 \rightarrow B1$
$\ldots$			
$\ldots$			

SCCP Example

Some Steps of SCCP Algorithm

Edge	Call	LatVal	Edges Inserted
(1) $S \to B0$	VisitInst(I_0)	$I_0 = 1$	$I_0 \to$ if, $I_0 \to I_1$, $B0 \to B1$
...			
(2) $I_0 \to$ if	VisitInst(if)	—	$B1 \to B2$, $B1 \to B8$
...			

SCCP Example

Some Steps of SCCP Algorithm

Edge	Call	LatVal	Edges Inserted
(1) $S \rightarrow B0$	VisitInst(I_0)	$I_0 = 1$	$I_0 \rightarrow$ if, $I_0 \rightarrow I_1$, $B0 \rightarrow B1$
$\ldots$			
(2) $I_0 \rightarrow$ if	VisitInst(if)	—	$B1 \rightarrow B2$, $B1 \rightarrow B8$
(3) $I_0 \rightarrow I_1$	VisitPhi(I_1)	$I_1 = 1 \sqcap \top = 1$	$I_1 \rightarrow J_0$

SCCP Example

Some Steps of SCCP Algorithm

Edge	Call	LatVal	Edges Inserted
(1) $S \rightarrow B0$	VisitInst(I_0)	$I_0 = 1$	$I_0 \rightarrow$ if, $I_0 \rightarrow I_1$, $B0 \rightarrow B1$
$\ldots$			
(2) $I_0 \rightarrow$ if	VisitInst(if)	—	$B1 \rightarrow B2$, $B1 \rightarrow B8$
(3) $I_0 \rightarrow I_1$	VisitPhi(I_1)	$I_1 = 1 \sqcap \top = 1$	$I_1 \rightarrow J_0$
(4) $I_1 \rightarrow J_0$	VisitInst(J_0)	$J_0 = 1$	$J_0 \rightarrow if(\ldots)$

SCCP Example

Some Steps of SCCP Algorithm

Edge	Call	LatVal	Edges Inserted
(1) $S \rightarrow B0$	VisitInst(I_0)	$I_0 = 1$	$I_0 \rightarrow$ if, $I_0 \rightarrow I_1$, $B0 \rightarrow B1$
...			
(2) $I_0 \rightarrow$ if	VisitInst(if)	—	$B1 \rightarrow B2$, $B1 \rightarrow B8$
(3) $I_0 \rightarrow I_1$	VisitPhi(I_1)	$I_1 = 1 \sqcap \top = 1$	$I_1 \rightarrow J_0$
(4) $I_1 \rightarrow J_0$	VisitInst(J_0)	$J_0 = 1$	$J_0 \rightarrow if(\ldots)$
(5) $J_0 \rightarrow if(\ldots)$	VisitInst(if)	—	$B4 \rightarrow B5$ (*not* $B4 \rightarrow B6$)

SCCP Example

Some Steps of SCCP Algorithm

Edge	Call	LatVal	Edges Inserted
(1) $S \rightarrow B0$	VisitInst(I_0)	$I_0 = 1$	$I_0 \rightarrow$ if, $I_0 \rightarrow I_1$, $B0 \rightarrow B1$
$\ldots$			
(2) $I_0 \rightarrow$ if	VisitInst(if)	—	$B1 \rightarrow B2$, $B1 \rightarrow B8$
(3) $I_0 \rightarrow I_1$	VisitPhi(I_1)	$I_1 = 1 \sqcap \top = 1$	$I_1 \rightarrow J_0$
(4) $I_1 \rightarrow J_0$	VisitInst(J_0)	$J_0 = 1$	$J_0 \rightarrow if(\ldots)$
(5) $J_0 \rightarrow if(\ldots)$	VisitInst(if)	—	$B4 \rightarrow B5$ (*not* $B4 \rightarrow B6$)
(6) $B4 \rightarrow B5$	VisitInst(I_3)	$I_3 = 1$	$I_3 \rightarrow I_4$, $B5 \rightarrow B6$

SCCP Example

Some Steps of SCCP Algorithm

Edge	Call	LatVal	Edges Inserted
(1) $S \to B0$	VisitInst(I_0)	$I_0 = 1$	$I_0 \to$ if, $I_0 \to I_1$, $B0 \to B1$
$\ldots$			
(2) $I_0 \to$ if	VisitInst(if)	—	$B1 \to B2$, $B1 \to B8$
(3) $I_0 \to I_1$	VisitPhi(I_1)	$I_1 = 1 \sqcap \top = 1$	$I_1 \to J_0$
(4) $I_1 \to J_0$	VisitInst(J_0)	$J_0 = 1$	$J_0 \to if(\ldots)$
(5) $J_0 \to if(\ldots)$	VisitInst(if)	—	$B4 \to B5$ (*not* $B4 \to B6$)
(6) $B4 \to B5$	VisitInst(I_3)	$I_3 = 1$	$I_3 \to I_4$, $B5 \to B6$
(7) $I_3 \to I_4$	VisitInst(I_4)	$I_4 = \top \sqcap 1 = 1$	$I_4 \to I_1$
$\ldots$			

SCCP Example

Some Steps of SCCP Algorithm

Edge	Call	LatVal	Edges Inserted
(1) $S \rightarrow B0$	VisitInst(I_0)	$I_0 = 1$	$I_0 \rightarrow$ if, $I_0 \rightarrow I_1$, $B0 \rightarrow B1$
$\ldots$			
(2) $I_0 \rightarrow$ if	VisitInst(if)	—	$B1 \rightarrow B2$, $B1 \rightarrow B8$
(3) $I_0 \rightarrow I_1$	VisitPhi(I_1)	$I_1 = 1 \sqcap \top = 1$	$I_1 \rightarrow J_0$
(4) $I_1 \rightarrow J_0$	VisitInst(J_0)	$J_0 = 1$	$J_0 \rightarrow if(\ldots)$
(5) $J_0 \rightarrow if(\ldots)$	VisitInst(if)	—	$B4 \rightarrow B5$ (*not* $B4 \rightarrow B6$)
(6) $B4 \rightarrow B5$	VisitInst(I_3)	$I_3 = 1$	$I_3 \rightarrow I_4$, $B5 \rightarrow B6$
(7) $I_3 \rightarrow I_4$	VisitInst(I_4)	$I_4 = \top \sqcap 1 = 1$	$I_4 \rightarrow I_1$
$\ldots$			
(8) $I_4 \rightarrow I_1$	VisitInst(I_1)	$I_1 = 1 \sqcap 1 = 1$	— (I_1 *unchanged*)
$\ldots$			

Induction Variable Substitution

Auxiliary Induction Variable

An *auxiliary induction variable* in a loop (`DO i = L, U, s`) is any variable 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, but m may be different at each use.

Optimization Goals

- Identify linear expression for each auxiliary induction variable
  $\Longrightarrow$ More effective dependence analysis, loop transformations
- *Optional*: Substitute linear expression in place of every use
- Eliminate expensive or loop-invariant operations from loop

Operator Strength Reduction

General Goal

Replace expensive operations by cheaper ones

Primitive Operations: Many Examples

- `n * 2` $\rightarrow$ `n << 1` (similarly, n/2)
- `n ** 2` $\rightarrow$ `n * n`

For Recurrences

- Example: `(base + (i-1) * 4)`
  Such recurrences are commonplace in array address calculations
  Note: Aux. induction variables are just a special case
- *Loop termination test*: $i < 100$:
  Can replace and eliminate an induction variable.
  This is called *Linear Function Test Replacement*.

Strength Reduction for Recurrences

Strategy

- Identify operations of the form:

  $$x \leftarrow \mathit{iv} \times c,\ x \leftarrow \mathit{iv} \pm c$$

  iv: induction variable or another recurrence
  c : loop-invariant variable
  Note: Fundamentally a dataflow problem
- eliminate multiplications from inside the loop
- eliminate induction variable if only remaining use is in loop termination test

Strength Reduction Example

```
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

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

SSA form

Strength Reduction Example (Continued)

$$
\begin{array}{ll}
 & \texttt{sum}_0 = \texttt{0.0} \\
 & \texttt{i}_0 = \texttt{1} \\
 & \texttt{t5}_0 = \texttt{a} \\
\texttt{L:} & \texttt{sum}_1 = \phi(\texttt{sum}_0, \texttt{sum}_2) \\
 & \texttt{i}_1 = \phi(\texttt{i}_0, \texttt{i}_2) \\
 & \texttt{t5}_1 = \phi(\texttt{t5}_0, \texttt{t5}_2) \\
 & \texttt{t4}_0 = \texttt{load t5}_0 \\
 & \texttt{sum}_2 = \texttt{sum}_1 + \texttt{t4}_0 \\
 & \texttt{i}_2 = \texttt{i}_1 + \texttt{1} \\
 & \texttt{t5}_2 = \texttt{t5}_1 + \texttt{4} \\
 & \texttt{if (i}_2 \texttt{ <= 100) goto L}
\end{array}
$$

After strength reduction

$$
\begin{array}{ll}
 & \texttt{sum}_0 = \texttt{0.0} \\
 & \texttt{t5}_0 = \texttt{a} \\
\texttt{L:} & \texttt{sum}_1 = \phi(\texttt{sum}_0, \texttt{sum}_2) \\
 & \texttt{t5}_1 = \phi(\texttt{t5}_0, \texttt{t5}_2) \\
 & \texttt{t4}_0 = \texttt{load t5}_0 \\
 & \texttt{sum}_2 = \texttt{sum}_1 + \texttt{t4}_0 \\
 & \texttt{t5}_2 = \texttt{t5}_1 + \texttt{4} \\
 & \texttt{if (t5}_2 \texttt{ <= 396 + a) goto L}
\end{array}
$$

After LFTR

Approaches to Strength Reduction

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

Allen, Cocke and Kennedy, "Reduction of Operator Strength," In S. Muchnick and N. Jones, editors, Program Flow Analysis: Theory and Applications*, pages 79–101. Prentice-Hall, 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
- worklist to find recurrences defined from other recurrences

$$y \leftarrow x \times a + b, \; z \leftarrow y \pm c$$

Approaches to Strength Reduction

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
- Mark each recurrence as an induction variable to find recurrences defined from other recurrences
  - $\Rightarrow$ order of SCCs is crucial
  - $\Rightarrow$ process an operation after all its operands

Tarjan's SCC algorithm Drives the Process

DFS(*node*)

  node.*DFSnum* $\leftarrow$ *nextDFSnum* $++$

  node.*visited* $\leftarrow$ TRUE

  node.*low* $\leftarrow$ *node*.*DFSnum*

  PUSH(*node*)

  for each $o \in$ {operands of *node*}

    if not o.*visited*

      DFS(o)

      node.*low* $\leftarrow$ MIN(*node*.*low*, o.*low*)

    if o.*DFSnum* $<$ *node*.*DFSnum* and $o \in$ *stack*

      node.*low* $\leftarrow$ MIN(o.*DFSnum*, *node*.*low*)

if *node*.*low* $=$ *node*.*DFSnum*

  SCC $\leftarrow \emptyset$

  do

    $x \leftarrow$ POP()

    SCC $\leftarrow$ *SCC* $\cup \{x\}$

  while $x \neq$ *node*

$\rightarrow$ ProcessSCC(*SCC*) $\leftarrow$

with one small addition . . .

Process Each SCC

```
ProcessSCC(scc )
   if (scc is a single node n : X = iv × rc or X = iv ± rc)
      Replace(n, iv, rc)
   else
      if ( all nodes in scc are ∈ {φ, +, −,COPY }
           and all external operands are loop-invariant)
                 // E.g., i2 = φ(i0, i2) + c
                 // May have multiple +, −,, COPY nodes
         Mark all nodes of scc as induction variables
         n.header = Header(scc ) ∀n ∈ scc
      else
         // scc is not an induction variable
         for each node n ∈ scc
           if (n : X = iv × rc or X = iv ± rc)
              Replace(n, iv, rc)
           else
              n.header = null
           endif
      endif
   endif
end // ProcessSCC()
```

Operator Strength Reduction Algorithm (contd.)

Replace an IV with a copy from a reduced version

```
Replace(n, iv, rc)
    Result = Reduce(n.op, iv, rc)
    Replace code for n with copy:  n.LHS = Result
    n.header ⟵ iv.header
end // Replace()
```

Operator Strength Reduction Algorithm (contd.)

Copy and simplify code for an induction variable:

```
Reduce(opcode, iv, rc)
    // Reuse computation of iv if it exists already
    IVcopy = HashLookup(opcode, iv, rc)
    if (IVcopy found)
        return IVcopy

    IVcopy = Copy of iv and its SCC
    HashInsert(opcode, iv, rc, IVcopy)

    For each operand o of IVcopy
        if (o ∈ scc )
            Reduce(opcode, o, rc)
        else
            // Insert SSA code to compute Result directly:
            //   (a) recursively strength-reduce operands w.r.t. outer loops
            //   (b) initialize Result outside loop
            //   (c) increment Result inside loop

    HashInsert(opcode, iv, rc, IVcopy)
    return IVcopy
end // Reduce()
```