Global Dataflow Optimizations

Overview of some fundamental machine-independent optimizations

Sparse Conditional Constant Propagation: SCCP

Simultaneously find constant-valued expressions and eliminate infeasible branches

Loop Invariant Code Motion: LICM

Hoist loop-invariant computations out of one or more loops.

Global Common Subexpression Elimination: GCSE

Identify redundant evaluations of expressions across an entire procedure (i.e., in the presence of control-flow).

Sparse Conditional Constant Propagation: SCCP

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

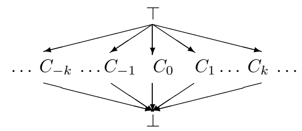
Lattice $L \equiv \{\top, C_i, \perp\}$.
$\top$ intuitively means "*May be constant.*"
$\perp$ intuitively means "*Not constant.*"

Meet Operator, $\sqcap$

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

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 = { $\top, C_i, \perp$ }
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
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-edge
            VisitInst(E->sink)
```

High-Level SCCP Algorithm (2 of 2)

VisitPhi(ϕ) :

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

VisitInst(S) : *Many errors in Muchnic*

```
val = Evaluate(S)
if (S is Assignment)
    LatCell(S) = val
    if (LatCell(S) changed)
        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) $S \to B0$	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)	—	$B1 \to B2,\ B1 \to B8$
(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)	$I_4 = \top \sqcap 1 = 1$	$I_4 \to I_1$
...			
(8) $I_4 \to I_1$	VisitInst(I_1)	$I_1 = 1 \sqcap 1 = 1$	— (I_1 *unchanged*)
...			

Loop-invariant Code Motion: LICM (1 of 2)

`S:  X = A + B;` *// enclosed in natural loop L*

Goals

For as many such statements S as possible:

- Move evaluation of *rvalue* $(A + B)$ out of L, if legal
- Move def of *lvalue* (X) out of L, if legal

Safety

Analysis: Find reaching defs of each variable in RHS and check if they are all outside the loop, or only one def reaches the variable and it is loop-invariant

Transformation: *Next slid*

Loop-invariant Code Motion: LICM (2 of 2)

Profitability

- Fewer computations (often, *much* fewer)
- Adds some `copy` instructions ⇒ cheaper than any operation
- May stretch some live ranges

Opportunity

- Array indexing expressions
- Structure indexing expressions
- Effect of previous tranformations (e.g., SCCP, DCE)
- Reordering program subexpressions by *loop-level*

Examples Illustrating Code Motion Rules

Example 1: Invariant def overwritten by later def

```
for (i=0; i < N; ++i) {
      X = a * b;    // hoist a*b but not def of X
      Y = X * i;
      X = Y + 1; }
```

Example 2: Def does not dominate exit

```
for (i=0; i < N; ++i) {
      if (...)
            X = a * b; // hoist a*b but not def of X }
```

Example 3: Multiple defs reach a use

```
for (i=0; i < N; ++i) {
      X = a * b;       // hoist a*b but not def of X
      if (...)
            X = X * i;
      Y = X; }
```


Checking Legality of Code Motion

Moving expression evaluation out of L:

($E1$) *Strict*: S must dominate all exit nodes from loop L

($E1'$) *Relaxed*: S must dominate all exit nodes from loop L
or $X + Y$ must not cause any exceptions

Moving def of X out of L:

($D1$) S must dominate all exit nodes from L
except exit nodes where X is dead

($D2$) No other statement in the loop must store to X

($D3$) No use of X in L must be reached by any other def of X.

Note: SSA simplifies these conditions!

($D1$) S must dominate all exit nodes from L
except exit nodes where X is dead

Algorithm for Loop-Invariant Code Motion (1 of 2)

Inputs

Procedure in 3-address form
Natural loop L, with preheader block P
Def-use and Use-def chains for the procedure

LICM()

```
repeat (until no new statements are marked)
   for (each statement S: X = expr in L)
      IsInvariant = true;
      for (all operands u ∈ S)
         if (any defs reaching u are within L)
            if (more than one def reaches u
                || (the single def d reaching u is
                      not constant and not invariant))
               { IsInvariant = false; break }
      if (IsInvariant)   // expr is loop-invariant
         Mark s invariant
```

Algorithm for Loop-Invariant Code Motion (2 of 2)

```
for (each statement S: X = expr in L) do
   if (S is marked invariant)
      if (BB containing S dominates all loop exits
              || expr causes no exceptions)
         insert tmp = expr just before loop L
      if (conditions (D1)...(D3) are satisfied) {
         insert X = tmp just before loop L;
         delete S
      } else
         replace S with X = tmp
```


Global Common Subexpression Elimination (1 of 2)

Goal

Eliminate redundant evaluation of an expression if it is available on all incoming paths

Safety

- *Analysis*: AVAIL proves that the value is current
- *Transformation*:
  - Introduce new temporary for each CSE discovered
  - don't add evaluations to any path

Global Common Subexpression Elimination (1 of 2)

Profitability

- same or fewer evaluations on every path
- add some `copy` instructions
  - ⇒ *many copies coalesce away during allocation*
- *major cost:* can stretch live ranges
  - ⇒ *may need forward substitution to undo some CSE results*

Opportunity

1. Array indexing expressions
2. Structure indexing expressions
3. *Clean* user-written code

Algorithm for GCSE

Inputs

(1) 3-address code + CFG for a procedure

(2) Numbered set of expressions $\mathcal{U} = \{e_1, \ldots e_N\}$
Use lexically identical expressions; apply reassociation first

(3) Available expressions, AVAIL$_{in}(B)$, for each block B

GCSE()

```
EverRedundant[i] = false,   ∀1 ≤ i ≤ N;
for each block B
    for each statement S : X = Y op Z in B
        if (e_j = "Y op Z" ∈ AVAIL_in(B)
              and e_j is not killed before S in B)
        {
            EverRedundant[j] = true
            Create new temporary tmp_j
            Replace S with X = tmp_j }
        }
```

Algorithm for GCSE

```
for each block B
    for each original statement T : X = Y op Z in B
        if (EverRedundant[k])      // where e_k = "Y op Z"
        {
            replace T with the pair:
                tmp_j  = Y op Z
                W    =  tmp_j
        }
```