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).

**SCCP: Key Algorithm Strengths**

**Conditional Constant Propagation**
Simultaneously finds constants + eliminates infeasible branches.

**Optimistic**
Assume every variable may be constant (⊥), 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. 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)

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

**Example 2:** Needs “Optimistic” initial assumption

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

**Example 3: Needs simultaneous condition evaluation + constant propagation**

```plaintext
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 \( L \)**:

\[
L = \{\top, C_i, \bot\}
\]

\( \top \) intuitively means "May be constant.

\( \bot \) intuitively means "Not constant."

**Meet Operator, \( \sqcap \)**

\[
\begin{align*}
\top \sqcap X &= X, \quad \forall X \in L \\
\bot \sqcap X &= \bot, \quad \forall X \in L \\
C_i \sqcap C_j &= \{ C_i, \text{ if } i = j, \bot, \text{ otherwise} \}
\end{align*}
\]

**Intuition: A Partial Order**

\( \bot \prec \top \) 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-\( \phi \) 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 *
3. Use 2 worklists: FlowWL, SSAWL
4. **Highlights:**
   - Visit \( S \) only if some incoming edge is executable
   - Ignore \( \phi \) 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()**

```plaintext
Initialize(ExecFlags[], LatCell[], FlowWL, SSAWL);
while ((Edge E = GetEdge(FlowWL \cup SSAWL)) != 0)
    if (E is a flow edge && ExecFlag[E] == false)
        ExecFlag[E] = true
        VisitPhi(\( \phi \)) \forall \ \phi \in 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 \( \phi \) node)
                VisitPhi(E->sink)
            else if (E->sink has 1 or more executable in-edge)
                VisitInst(E->sink)
    else if (E is a flow edge && ExecFlag[E] == false)
        if (E is a flow edge && ExecFlag[E] == false)
            VisitPhi(\( \phi \)) \forall \ \phi \in 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 \( \phi \) 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)

\textbf{VisitPhi(\(\phi\)) :}

\begin{itemize}
  \item for (all operands \(U_k\) of \(\phi\))
  \item if (ExecFlag\([\text{InEdge}(k)]\) == true)
  \item LatCell(\(\phi\)) \(\cap\) = LatCell(\(U_k\))
  \item if (LatCell(\(\phi\)) changed)
  \item add SSAOutEdges(\(\phi\)) to SSAWL
\end{itemize}

\textbf{VisitInst(\(S\)) :}

\begin{itemize}
  \item val = Evaluate(\(S\))
  \item if (\(S\) is Assignment)
  \item LatCell(\(S\)) = val
  \item if (LatCell(\(S\)) changed)
  \item add SSAOutEdges(\(S\)) to SSAWL
  \item else // \(S\) must be a Branch
  \item Add one or both outgoing edges to FlowWL
\end{itemize}

SCCP Example

\textbf{Example 3: Needs simultaneous condition evaluation + constant propagation}

\begin{itemize}
  \item B0: \(I_0 = 1\)
  \item B1: if \(I_0 < N_0\)
  \item B2: \(I_1 = \phi(I_0, I_4)\)
  \item B3: \(I_2 = f(\ldots)\)
  \item B4: if \(J_0 > 0\)
  \item B5: \{ \(I_3 = J_0\) \}
  \item B6: \(I_4 = \phi(I_2, I_1)\)
  \item B7: goto B1
  \item B8: \ldots
\end{itemize}

SCCP Example

Some Steps of SCCP Algorithm

<table>
<thead>
<tr>
<th>Edge</th>
<th>Call</th>
<th>LatVal</th>
<th>Edges Inserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (S \rightarrow B0)</td>
<td>VisitInst(I0)</td>
<td>I0 = 1</td>
<td>I0 -&gt; #, I0 -&gt; I1, B0 -&gt; B1</td>
</tr>
<tr>
<td>2 (I_0 \rightarrow #)</td>
<td>VisitInst(I0)</td>
<td>—</td>
<td>B1 -&gt; B2, B1 -&gt; B8</td>
</tr>
<tr>
<td>3 (I_0 \rightarrow I_1)</td>
<td>VisitPhi(I1)</td>
<td>I1 = 1 (\cap) = 1</td>
<td>I1 -&gt; J0</td>
</tr>
<tr>
<td>4 (I_1 \rightarrow J_0)</td>
<td>VisitLat(I0)</td>
<td>J0 = 1</td>
<td>J0 = (f(\ldots))</td>
</tr>
<tr>
<td>5 (J_0 \rightarrow {f(\ldots)})</td>
<td>VisitLat(I0)</td>
<td>—</td>
<td>B4 -&gt; B5 (not B4 -&gt; B6)</td>
</tr>
<tr>
<td>6 (B4 \rightarrow B5)</td>
<td>VisitLat(I1)</td>
<td>I1 = 1</td>
<td>I1 -&gt; I4, B5 -&gt; B6</td>
</tr>
<tr>
<td>7 (I_4 \rightarrow I_4)</td>
<td>VisitLat(I1)</td>
<td>I4 = (\top \cap) = 1</td>
<td>I4 -&gt; I4</td>
</tr>
<tr>
<td>8 (I_4 \rightarrow I_1)</td>
<td>VisitLat(I1)</td>
<td>I4 = 1 (\cap) = 1</td>
<td>— (I4 unchanged)</td>
</tr>
</tbody>
</table>

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

\textbf{S:} \(X = A + B\) // enclosed in natural loop L

\textbf{Goals}

\begin{itemize}
  \item Move evaluation of value \((A + B)\) out of \(L\), if legal
  \item Move def of value \((X)\) out of \(L\), if legal
\end{itemize}

\textbf{Safety}

\begin{itemize}
  \item 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
  \item Transformation: \textit{Next slide}
\end{itemize}
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 transformations (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 \( L_1 \):

\[(E1)\] 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 \( Y \) out of \( L_1 \):

\[(D1)\] \( S \) 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 \( S \) \( \times \) expr in \( L \))
  \( \text{IsInvariant} = \text{true}; \)
  for (all operands \( u \) in \( S \))
    if (any defs reaching \( u \) are within \( L \))
      if (more than one def reaches \( u \) || (the single def reaching \( u \) is not constant and not invariant))
        \( \text{IsInvariant} = \text{false}; \) break
    if (\( \text{IsInvariant} = \text{false} \))
      Mark \( S \) invariant

Examples for Loop-Invariant Code Motion

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;
}
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
    else
      replace S with X = tmp
  else
    delete S

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 U = {e1, ..., eN}
Use lexically identical expressions; apply reassociation first
(3) Available expressions, \( \text{AVAIL}_{\text{in}}(B) \), for each block B

\[ \text{EverRedundant}[i] = \text{false}, \ \forall 1 \leq i \leq N; \]
for each block B
  for each statement S: X = Y op Z in B
    if \( (e_j = 'Y op Z' \in \text{AVAIL}_{\text{in}}(B) \) and \( e_j \) is not killed before S in B)
      [ EverRedundant[j] = true
        Create new temporary \( \text{tmp}_j \)
        Replace S with \( X = \text{tmp}_j \) ]
for each block $B$
  for each original statement $T : X = Y \text{ op } Z$ in $B$
    if $\text{EverRedundant}[k]$ // where $e_k = "Y \text{ op } Z"$
      { replace $T$ with the pair:
          $\text{tmp}_j = Y \text{ op } Z$
          $W = \text{tmp}_j$
        }