Partial Redundancy Elimination

Motivation

```
if() {
    a = x + y;
}
b = x + y;
```

Motivation

```
while(c) {
    a = x + y;
}
```

PRE: Goals and Assumptions

Assumption

Assume that at any statement of the form `a = x + y`, the current value of `x + y` must be placed in `a`. That is, the computation of `x + y` cannot be deferred until a later point where `a` is actually used.

PRE: Goals and Assumptions

Desired Transformation

Introduce a temporary $\mathtt{t_{x+y}}$. Change every statement of the form `a = x + y` into `a =` $\mathtt{t_{x+y}}$. Insert computations of the form $\mathtt{t_{x+y}}$ `= x + y` at some subset S of program points (nodes and edges) such that the same values are assigned to `a` as in the original program. [Safe]

PRE: Goals and Assumptions

Desired Transformation

Introduce a temporary $\mathtt{t_{x+y}}$. Change every statement of the form `a = x + y` into `a =` $\mathtt{t_{x+y}}$. Insert computations of the form $\mathtt{t_{x+y}}$ `= x + y` at some subset S of program points (nodes and edges) such that the same values are assigned to a as in the original program. [Safe]

Goals for S

1. Suppose S' is also safe. No execution path should contain more occurences of $\mathtt{t_{x+y}}$ `= x + y` in S than in S'. [Computationally Optimal]
2. Suppose S' is also safe and computationally optimal. At every program point where $\mathtt{t_{x+y}}$ is live under S, it should also be live under S'. [Lifetime Optimal]

Variations of PRE

Note: this is not an exhaustive list.

- E. Morel and C. Renvoise. *Global optimization by suppression of partial redundancies.* CACM, 1979.
- J. Knoop, O. Rüthing, and B. Steffen. *Lazy code motion.* PLDI 1992.
- K.-H. Drechsler and M.P. Stadel. *A variation of Knoop, Rüthing, and Steffen's Lazy Code Motion.* SIGPLAN Notices, 1993.
- R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, F. C. Chow. *Partial redundancy elimination in SSA form.* ACM TOPLAS 21(3): 627-676, 1999.

Summary of Properties

- Local properties
  - transparent
  - computed
  - locally anticipable
- Global node properties
  - available
  - anticipable
- Global edge properties
  - earliest
  - later
- Final results
  - insert (on edge)
  - delete (from node)

Local Properties

Definition

A basic block b is transparent for expression e if none of e's operands are defined in b.

Definition

An expression e is computed (aka downward exposed aka locally available) in basic block b if it contains a computation of e, and does not define e's operands after the last computation of e.

Definition

An expression e is locally anticipable (aka upward exposed) in basic block b if it contains a computation of e, and does not define e's operands before the first computation of e.

Availability and Anticipability

Definition

An expression e is available at program point p if on every path from the start node to p, e is computed, and e's operands are not defined after the last computation of e.

Compute using dataflow analysis:

1. Forward or Backward?

Availability and Anticipability

Definition

An expression e is available at program point p if on every path from the start node to p, e is computed, and e's operands are not defined after the last computation of e.

Compute using dataflow analysis:

1. Forward or Backward? forward
2. Domain?

Availability and Anticipability

Definition

An expression e is available at program point p if on every path from the start node to p, e is computed, and e's operands are not defined after the last computation of e.

Compute using dataflow analysis:

1. Forward or Backward? forward
2. Domain? (Exprs, $\supseteq$)
3. Merge Operator?

Availability and Anticipability

Definition

An expression e is available at program point p if on every path from the start node to p, e is computed, and e's operands are not defined after the last computation of e.

Compute using dataflow analysis:

1. Forward or Backward? forward
2. Domain? (Exprs, $\supseteq$)
3. Merge Operator? $\cap$
4. Flow Equation?

Availability and Anticipability

Definition

An expression e is available at program point p if on every path from the start node to p, e is computed, and e's operands are not defined after the last computation of e.

Compute using dataflow analysis:

1. Forward or Backward? forward
2. Domain? (Exprs, $\supseteq$)
3. Merge Operator? $\cap$
4. Flow Equation?
   $\text{out}(s) = \text{computed}(s) \cup (\text{in}(s) \cap \text{transparent}(s))$
5. out(Start)?

Availability and Anticipability

Definition

An expression e is available at program point p if on every path from the start node to p, e is computed, and e's operands are not defined after the last computation of e.

Compute using dataflow analysis:

1. Forward or Backward? forward
2. Domain? (Exprs, $\supseteq$)
3. Merge Operator? $\cap$
4. Flow Equation?
   $\text{out}(s) = \text{computed}(s) \cup (\text{in}(s) \cap \text{transparent}(s))$
5. out(Start)? empty set
6. Bottom Element?

Availability and Anticipability

Definition

An expression e is available at program point p if on every path from the start node to p, e is computed, and e's operands are not defined after the last computation of e.

Compute using dataflow analysis:

1. Forward or Backward? forward
2. Domain? (Exprs, $\supseteq$)
3. Merge Operator? $\cap$
4. Flow Equation?
   $\text{out}(s) = \text{computed}(s) \cup (\text{in}(s) \cap \text{transparent}(s))$
5. out(Start)? empty set
6. Bottom Element? $\perp =$ all expressions

Availability and Anticipability

Definition

An expression e is anticipable at program point p if on every path from p to the end node, e is computed, and e's operands are not defined before the first computation of e.

Compute using dataflow analysis:

1. Forward or Backward?

Availability and Anticipability

Definition

An expression e is anticipable at program point p if on every path from p to the end node, e is computed, and e's operands are not defined before the first computation of e.

Compute using dataflow analysis:

1. Forward or Backward? backward
2. Domain?

Availability and Anticipability

Definition

An expression e is anticipable at program point p if on every path from p to the end node, e is computed, and e's operands are not defined before the first computation of e.

Compute using dataflow analysis:

1. Forward or Backward? backward
2. Domain? (Exprs, $\supseteq$)
3. Merge Operator?

Availability and Anticipability

Definition

An expression e is anticipable at program point p if on every path from p to the end node, e is computed, and e's operands are not defined before the first computation of e.

Compute using dataflow analysis:

1. Forward or Backward? backward
2. Domain? (Exprs, $\supseteq$)
3. Merge Operator? $\cap$
4. Flow Equation?

Availability and Anticipability

Definition

An expression e is anticipable at program point p if on every path from p to the end node, e is computed, and e's operands are not defined before the first computation of e.

Compute using dataflow analysis:

1. Forward or Backward? backward
2. Domain? (Exprs, $\supseteq$)
3. Merge Operator? $\cap$
4. Flow Equation?
   $\text{in}(s) = \text{locally anticipable}(s) \cup (\text{out}(s) \cap \text{transparent}(s))$
5. in(Exit)?

Availability and Anticipability

Definition

An expression e is anticipable at program point p if on every path from p to the end node, e is computed, and e's operands are not defined before the first computation of e.

Compute using dataflow analysis:

1. Forward or Backward? backward
2. Domain? (Exprs, $\supseteq$)
3. Merge Operator? $\cap$
4. Flow Equation?
   $\text{in}(s) = \text{locally anticipable}(s) \cup (\text{out}(s) \cap \text{transparent}(s))$
5. in(Exit)? empty set
6. Bottom Element?

Availability and Anticipability

Definition

An expression e is anticipable at program point p if on every path from p to the end node, e is computed, and e's operands are not defined before the first computation of e.

Compute using dataflow analysis:

1. Forward or Backward? backward
2. Domain? (Exprs, $\supseteq$)
3. Merge Operator? $\cap$
4. Flow Equation?
   $\text{in}(s) = \text{locally anticipable}(s) \cup (\text{out}(s) \cap \text{transparent}(s))$
5. in(Exit)? empty set
6. Bottom Element? $\perp =$ all expressions

Earliest Placement

The edge (i, j) is the earliest point where we should compute the expression e if

- e is needed on all paths from j to the end node,
- e is not available at the end of i, and
  - a computation before i would get invalidated in i, or
  - e is not needed on some other edge out of i.

$$
\begin{aligned}
\text{earliest}(i, j) \quad = \quad & \text{anticipable}_{\text{in}}(j) \\
& \cap \overline{\text{available}_{\text{out}}(i)} \\
& \cap (\overline{\text{transparent}(i)} \cup \overline{\text{anticipable}_{\text{out}}(i)})
\end{aligned}
$$

Latest Placement

A computation of e can be moved from before a block b to after b, as long as it needs to be computed on all incoming edges of b, and e is not needed in b.

$$
\begin{aligned}
\text{later}(i, j) &= \text{earliest}(i, j) \\
&\cup \left(\overline{\text{locally anticipable}(i)} \cap \bigcap_{k \in \text{pred}(i)} \text{later}(k, i) \right)
\end{aligned}
$$

The Transformation

Insert computation as late as possible:

$$\text{insert}(i,j) \;=\; \text{later}(i,j) \cap \overline{\bigcap_{k \in \text{pred}(j)} \text{later}(k,j)}$$

$e \in \text{insert}(i,j)$ means compute e in edge (i,j).

Remove locally anticipable computations where value is already known:

$$\text{delete}(j) \;=\; \text{locally anticipable}(j) \cap \overline{\bigcap_{i \in \text{pred}(j)} \text{later}(i,j)}$$

$e \in \text{delete}(j)$ means remove first computation of e from j.