Partial Redundancy Elimination

- Global code motion optimization
  - Remove partially redundant expressions
  - Loop invariant code motion
  - Can be extended to do Strength Reduction
- No loop analysis needed
- Bidirectional flow problem

[ALSU 9.5-9.5.2]

Partial Redundancy

- Partially Redundant Computation
  - Occurrence of expression $E$ at $P$ is partially redundant if $E$ is partially available there:
    - $E$ is evaluated along at least one path to $P$, with no operands redefined since.
  - Partially redundant expression can be eliminated if we can insert computations to make it fully redundant.

Loop Invariants are Partial Redundancies

- Loop invariant expression is partially redundant
  - As before, partially redundant computation can be eliminated if we insert computations to make it fully redundant.
  - Remaining copies can be eliminated through copy propagation or more complex analysis of partially redundant assignments.

Redundancy

- A Common Subexpression is a Redundant Computation
  - Occurrence of expression $E$ at $P$ is redundant if $E$ is available there:
    - $E$ is evaluated along every path to $P$, with no operands redefined since.
  - Redundant expression can be eliminated
Partial Redundancy Elimination

- **The Method:**
  1. Insert Computations to make partially redundant expression(s) fully redundant.
  2. Eliminate redundant expression(s).
- **Issues [Outline of Lecture]:**
  1. What expression occurrences are candidates for elimination?
  2. Where can we safely insert computations?
  3. Where do we want to insert them?
- For this lecture, we assume one expression of interest, a+b.
  - In practice, with some restrictions, can do many expressions in parallel.

Which Occurrences Might Be Eliminated?

- In CSE,
  - E is **available** at P if it is previously evaluated along every path to P, with no subsequent redefinitions of operands.
  - If so, we can eliminate computation at P.
- In PRE,
  - E is **partially available** at P if it is previously evaluated along at least one path to P, with no subsequent redefinitions of operands.
  - If so, we might be able to eliminate computation at P, if we can insert computations to make it fully redundant.
- Occurrences of E where E is partially available are candidates for elimination.

Finding Partially Available Expressions

- **Forward Flow problem**
  - Lattice = \( \{0, 1\}^n \), meet is union \( \cup \) (elementwise \( \max \))
  - \( \text{Top} = 0^n \) (= not PAVAIL), \( \text{entry} = 0^n \), \( \text{init} = 0^n \)
  - \( \text{PAVOUT}[b] = (\text{PAVIN}[b] - \text{KILL}[b]) \cup \text{AVLOC}[b] \)
  - \( \text{PAVIN}[b] = \begin{cases} 0^n & b = \text{entry} \\ \bigcup_{p \in \text{preds}(b)} \text{PAVOUT}[p] & \text{otherwise} \end{cases} \)
  - For a block,
    - Expression is **locally available** (AVLOC) if computed & downwards exposed.
    - Expression is killed (KILL) if any assignments to operands.

Partial Availability Example

- For expression a+b
  - \( \text{PAVOUT}[\text{entry}] = 0 \)
  - \( \begin{array}{l}
        \text{a} = \ldots \\
        \text{t1} = \text{a} + \text{b} \\
        \text{KILL} = 1 \\
        \text{AVLOC} = 0 \\
        \text{PAVOUT} = \\
    \end{array} \\
  - \( \begin{array}{l}
        \text{a} = \ldots \\
        \text{t2} = \text{a} + \text{b} \\
        \text{KILL} = 1 \\
        \text{AVLOC} = 1 \\
        \text{PAVOUT} = \\
    \end{array} \\
  - \( \text{PAVOUT}[b] = (\text{PAVIN}[b] - \text{KILL}[b]) \cup \text{AVLOC}[b] \)

Occurrence in loop is partially redundant
Where Can We Insert Computations?

- **Safety**: never introduce a new expression along any path.
  - Insertion could introduce exception, change program behavior.
  - If we can add a new basic block, can insert safely in most cases.
  - Solution: insert expression only where it is anticipated.

- **Performance**: never increase the # of computations on any path.
  - Under simple model, guarantees program won’t get worse.
  - Reality: might increase register lifetimes, add copies, lose.

Finding Anticipated Expressions

- **Backward flow problem**
  - Lattice = \{0, 1\}, meet is intersection (∧), top = 1 (ANT), exit = 0, init = 1
    - \(\text{ANTIN}[i] = \text{ANTLOC}[i] \cup (\text{ANTOUT}[i] - \text{KILL}[i])\)
    - \(\text{ANTOUT}[i] = \begin{cases} 0 & \text{if exit} \\ \text{ANTIN}[i] \cap \text{ANTOUT}[i] & \text{otherwise} \end{cases}\)

- **For a block**
  - Expression locally anticipated (ANTLOC) if defined & upwards exposed

    \[
    \begin{align*}
    a &= \ldots \\
    t_1 &= \ldots \\
    t_2 &= \ldots \\
    t_3 &= \ldots \\
    \end{align*}
    \]

Where Do We Want to Insert Computations?

- Morel-Renovise and variants: “Placement Possible”
  - Dataflow analysis shows where to insert:
    - PPIN = "Placement possible at entry of block or before."
    - PPOUT = "Placement possible at exit of block or before."
  - Insert at earliest place where PPIN = 1.
  - Only place at end of blocks,
    - PPIN really means "Placement possible or not necessary in each predecessor block."
    - Don’t need to insert where expression is already available.

  - \(\text{INSERT}[i] = \text{PPOUT}[i] \cap (\neg \text{PPIN}[i] \cup \text{KILL}[i] \cap \neg \text{AVOUT}[i])\)
  - Can put Can’t move it Not already it here back any further available
  - Remove (upwards-exposed) computations where PPIN=1.

  - \(\text{DELETE}[i] = \text{PPIN}[i] \land \text{ANTLOC}[i]\)
    - Moved Used locally earlier here
Where Do We Want to Insert? Example

\[ t_1 = a + b \]
\[ a = \ldots \]
\[ t_2 = a + b \]

Formulating the Problem

- **PPOUT**: we want to place at output of this block only if
  - we want to place at entry of all successors (correctness & performance)
- **PPIN**: we want to place at input of this block only if (all of):
  - we have a local computation to place, or a placement at the end of this block which we can move up
  - we want to move computation to output of all predecessors where expression is not already available (don't insert at input)
  - we can gain something by placing it here (PAVIN)
- **Forward or Backward?**
  - BOTH!

Problem is bidirectional, but lattice \([0, 1]\) is finite, so
  - as long as transfer functions are monotone, it converges.

Computing “Placement Possible”

- **PPOUT**: we want to place at output of this block only if
  - we want to place at entry of all successors
    - \[ PPOUT[i] = \bigcap_{s \in \text{succ}(i)} PPIN[i] \]
- **PPIN**: we want to place at start of this block only if (all of):
  - we have a local computation to place, or a placement at the end of this block which we can move up
  - we want to move computation to output of all predecessors where expression is not already available (don't insert at input)
  - we gain something by moving it up (PAVIN heuristic)
    - \[ \text{ANTLOC}[i] \bigcup (\text{PPOUT}[p] - \text{KILL}[p]) \]
  - \[ PPIN[i] = \bigcap_{p \in \text{preds}(i)} \left( PPOUT[p] \cup \text{AVOUT}[p] \right) \]

“Placement Possible” Example 1

\[ t_1 = a + b \]
\[ a = \ldots \]
\[ t_2 = a + b \]

\[ \text{KILL} = 1 \]
\[ \text{AVLOC} = 0 \]
\[ \text{ANTLOC} = 0 \]
\[ \text{PAVIN} = 0 \]
\[ \text{PAVOUT} = 0 \]
\[ \text{AVOUT} = 0 \]
\[ PPIN = \]
\[ PPOUT = \]
\[ \text{KILL} = 0 \]
\[ \text{AVLOC} = 1 \]
\[ \text{ANTLOC} = 1 \]
\[ \text{PAVIN} = 1 \]
\[ \text{PAVOUT} = 1 \]
\[ \text{AVOUT} = 1 \]
\[ PPIN[\text{entry}] = 0 \]
\[ PPOUT[\text{entry}] = 0 \]
\[ \text{KILL} = 1 \]
\[ \text{AVLOC} = 1 \]
\[ \text{ANTLOC} = 1 \]
\[ \text{PAVIN} = 1 \]
\[ \text{PAVOUT} = 1 \]
\[ \text{AVOUT} = 1 \]
\[ PPIN[\text{exit}] = 0 \]
“Placement Possible” Example 2

\[
\begin{align*}
\text{a} &= \ldots \\
\text{t1} &= \text{a} + \text{b} \\
\text{a} &= \ldots \\
\text{t2} &= \text{a} + \text{b}
\end{align*}
\]

KILL = 1
AVLOC = 1
ANTLOC = 0
KILL = 1
AVLOC = 0
ANTLOC = 1

\[
\begin{align*}
\text{PAVIN} &= 0 \\
\text{PPOUT}[\text{entry}] &= 0 \\
\text{PPIN} &= 0 \\
\text{PPOUT} &= 0
\end{align*}
\]

PPIN =
PPIN =
PPIN =

\[
\begin{align*}
\text{PPIN}[\text{exit}] &= 0 \\
\text{PPOUT}[\text{entry}] &= 0
\end{align*}
\]

“Placement Possible” Correctness

- Convergence of analysis: transfer functions are monotone
- Safety: Insert only if anticipated
  \[
  \text{PPIN}[i] \subseteq (\text{PPOUT}[i] - \text{KILL}[i]) \cup \text{ANTLOC}[i]
  \]
- Performance: never increase the # of computations on any path
  - \text{DELETE} = \text{PPIN} \cap \text{ANTLOC}
  - On every path from an INSERT, there is a DELETE
  - The number of computations on a path does not increase

Morel-Renvoise Limitations

- Movement usefulness tied to PAVIN heuristic
  - Makes some useless moves, might increase register lifetimes:

\[
\begin{align*}
\text{a+b} &\quad \text{Not anticipated, so incorrect to place here} \\
\text{a+b} &\quad \text{PAVIN & ANTLOC, so PPIN} \\
\text{a+b} &\quad \text{not PPIN for all succ, so not PPOUT} \\
\text{a+b} &\quad \text{PAVIN & ANTLOC, so PPIN} \\
\text{a+b} &\quad \text{not PAVIN so not PPIN}
\end{align*}
\]

- Bidirectional data flow difficult to compute

Friday’s Class

- Lazy Code Motion
  [ALSU 9.5.3-9.5.6]