Lecture 10

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
References

**Redundancy**

- A **Common Subexpression** is a **Redundant Computation**

  \[ t_1 = a + b \]

  \[ t_2 = a + b \]

  \[ t_3 = a + b \]

- **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

- Partially Redundant Computation

\[ t_1 = a + b \]

\[ t_3 = a + b \]

- 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

\[ a = \ldots \]

\[ t_1 = a + b \]

- 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.
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 \}, meet is union (\( \cup \)), Top = 0 (not PAVAIL), entry = 0
    • PAVOUT\[i\] = (PAVIN\[i\] - KILL\[i\]) \( \cup \) AVLOC\[i\]
    • PAVIN\[i\] = \begin{cases} 0 & \text{if } i = \text{entry} \\ \bigcup_{p \in \text{preds}(i)} \text{PAVOUT}[p] & \text{otherwise} \end{cases}

• For a block,
  • Expression is locally available (AVLOC) if downwards exposed.
  • Expression is killed (KILL) if any assignments to operands.

\[
\begin{array}{c}
a = \ldots \\
\ldots = a + b \\
a = \ldots
\end{array}
\quad
\begin{array}{c}
\ldots = a + b \\
a = \ldots
\end{array}
\quad
\begin{array}{c}
\ldots = a + b \\
a = \ldots
\end{array}
\quad
\begin{array}{c}
\ldots = a + b \\
a = \ldots
\end{array}
\]
Partial Availability Example

- For expression $a+b$.

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

- 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 intersect (\( \cap \)), top = 1 (ANT), exit = 0
  
    • \( \text{ANTIN}[i] = \text{ANTLOC}[i] \cup (\text{ANTOUT}[i] - \text{KILL}[i]) \)
    
    • \( \text{ANTOUT}[i] = \begin{cases} 0 & \text{if } i = \text{exit} \\ \cap \text{ANTIN}[s] & \text{otherwise} \end{cases} \quad s \in \text{succ}(i) \)

• For a block,
  - Expression **locally anticipated** (ANTLOC) if upwards exposed.
Anticipation Example

• For expression $a+b$.

$\begin{align*}
  a &= \ldots \\
  t_1 &= a + b \\
  a &= \ldots
\end{align*}$

$\begin{align*}
  \text{KILL} &= 1 & \text{ANTIN} &= \\
  \text{ANTLOC} &= 0 & \text{ANTOUT} &= \\
  \text{KILL} &= 0 & \text{ANTIN} &= \\
  \text{ANTLOC} &= 1 & \text{ANTOUT} &= \\
  \text{KILL} &= 1 & \text{ANTIN} &= \\
  \text{ANTLOC} &= 0 & \text{ANTOUT} &=
\end{align*}$

• Expression is anticipated at end of first block.
• Computation may be safely inserted there.
Where Do We Want to Insert Computations?

- **Morel-Renvoise 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 PP = 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.
    - INSERT[i] = PPOUT[i] \(\cap (\neg PPIN[i] \cup KILL[i]) \cap \neg AVOUT[i]\)
  - Remove (upwards-exposed) computations where PPIN=1.
    - DELETE[i] = PPIN[i] \(\cap ANTLOC[i]\)
Where Do We Want to Insert? Example

\[
a = \ldots
\]

\[
t1 = a + b
\]

\[
a = \ldots
\]

\[
t2 = a + b
\]

PPIN =

PPOUT =

PPIN =

PPOUT =

PPIN =

PPOUT =

PPIN =

PPOUT =
Formulating the Problem

- **PPOUT**: we want to place at output of this block only if
  - we want to place at entry of all successors

- **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
    
    $\{0\} \quad i = \text{exit}$
    
    $\bigcap_{s \in \text{succ}(i)} \text{PPIN}[s] \quad \text{otherwise}$

- **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)
    
    $\{0\} \quad i = \text{exit}$
    
    $\bigcap_{p \in \text{preds}(i)} \bigcap \text{P(PPOUT}[p] \text{ AVOUT}[p]) \quad \text{otherwise}$
"Placement Possible" Example 1

a = ...

KILL = 1  PAVIN = 0  PPIN =       
AVLOC = 0  PAVOUT = 0       
ANTLOC = 0  AVOUT = 0  PPOUT =       

KILL = 0  PAVIN = 1  PPIN =       
AVLOC = 1  PAVOUT = 1       
ANTLOC = 1  AVOUT = 0  PPOUT =       

KILL = 1  PAVIN = 1  PPIN =       
AVLOC = 1  PAVOUT = 1       
ANTLOC = 0  AVOUT = 1  PPOUT =
"Placement Possible" Example 2

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

\[ t_2 = a + b \]

\[
\begin{align*}
\text{KILL} &= 1 & \text{PAVIN} &= 0 & \text{PPIN} &=  \\
\text{AVLOC} &= 1 & \text{PAVOUT} &= 1 & \text{PPOUT} &=  \\
\text{ANTLOC} &= 0 & \text{AVOUT} &= 1 & \text{PPOUT} &= \\
\end{align*}
\]

\[
\begin{align*}
\text{KILL} &= 1 & \text{PAVIN} &= 0 & \text{PPIN} &=  \\
\text{AVLOC} &= 0 & \text{PAVOUT} &= 0 & \text{PPOUT} &=  \\
\text{ANTLOC} &= 0 & \text{AVOUT} &= 0 & \text{PPOUT} &=  \\
\end{align*}
\]

\[
\begin{align*}
\text{KILL} &= 0 & \text{PAVIN} &= 1 & \text{PPIN} &=  \\
\text{AVLOC} &= 1 & \text{PAVOUT} &= 1 & \text{PPOUT} &=  \\
\text{ANTLOC} &= 1 & \text{AVOUT} &= 1 & \text{PPOUT} &=  \\
\end{align*}
\]
“Placement Possible” Correctness

- **Convergence** of analysis: transfer functions are monotone.
- **Safety**: Insert only if anticipated.

\[
\begin{align*}
PPIN[i] & \subseteq (PPOUT[i] - KILL[i]) \cup ANTLOC[i] \\
SPOUT[i] &= \begin{cases} 
0 & i = \text{exit} \\
\cap_{s \in \text{succ}(i)} PPIN[s] & \text{otherwise}
\end{cases}
\end{align*}
\]

- **Performance**: never increase the # of computations on any path
  - **DELETE** = \( PPIN \cap ANTLOC \)
  - 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:

![Diagram](image1)

- Doesn’t find some eliminations:

![Diagram](image2)

- Bidirectional data flow difficult to compute.
Related Work

• Don't need heuristic
  – Dhamdhere, Drechsler-Stadel, Knoop, et.al.
  – use restricted flow graph or allow edge placements.

• Data flow can be separated into unidirectional passes
  – Dhamdhere, Knoop, et. al.

• Improvement still tied to accuracy of computational model
  – Assumes performance depends only on the number of computations along any path.
  – Ignores resource constraint issues: register allocation, etc.
  – Knoop, et.al. give “earliest” and “latest” placement algorithms which begin to address this.

• Further issues:
  – more than one expression at once, strength reduction, redundant assignments, redundant stores.