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

\[
\begin{align*}
\text{t1} &= a + b \\
\text{t2} &= a + b \\
\text{t3} &= a + b
\end{align*}
\]

• 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

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

• 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 \textit{CSE},
  – E is \textit{available} at P if it is previously evaluated along \textit{every} path to P, with no subsequent redefinitions of operands.
  – If so, we can eliminate computation at P.

• In \textit{PRE},
  – E is \textit{partially available} at P if it is previously evaluated along \textit{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.

• \textit{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\[i\] = \(\begin{cases} 
  0 & i = \text{entry} \\
  \cup_{p \in \text{preds}(i)} \text{PAVOUT}[p] & \text{otherwise}
\end{cases}\)

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

\[
\begin{align*}
  a = \ldots \\
  \ldots = a + b \\
  a = \ldots
\end{align*}
\]
Partial Availability Example

- For expression $a + b$.

- 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 (\( \cap \)), *top* = 1 (ANT), *exit* = 0

  • ANTIN[i] = ANTLOC[i] \cup (ANTOUT[i] - KILL[i])
  • ANTOUT[i] = \left\{ \begin{array}{ll}
      0 & i = \text{exit} \\
      \cap \text{ANTIN}[s] & \text{otherwise}
    \end{array} \right.

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

\[
\begin{align*}
  a &= \ldots \\
  \ldots &= a + b \\
  a &= \ldots
\end{align*}
\]
Anticipation Example

• For expression \( a+b \).

  \[
  \begin{align*}
  &a = \ldots \quad \text{KILL} = 1 \quad \text{ANTIN} = \ldots \\
  &\phantom{a} \quad \text{ANTLOC} = 0 \quad \text{ANTOUT} = \ldots
  \end{align*}
  \]

  \[
  \begin{align*}
  &t1 = a + b \quad \text{KILL} = 0 \quad \text{ANTIN} = \ldots \\
  &\phantom{t1} \quad \text{ANTLOC} = 1 \quad \text{ANTOUT} = \ldots
  \end{align*}
  \]

  \[
  \begin{align*}
  &a = \ldots \quad \text{KILL} = 1 \quad \text{ANTIN} = \ldots \\
  &t2 = a + b \quad \text{ANTLOC} = 0 \quad \text{ANTOUT} = \ldots
  \end{align*}
  \]

• 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] ∩ (¬PPIN[i] ∪ KILL[i]) ∩ ¬AVOUT[i]

• DELETE[i] = PPIN[i] ∩ ANTLOC[i]

– Remove (upwards-exposed) computations where PPIN=1.
Where Do We Want to Insert? Example

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

 PPIN = PPOUT =

 PPIN = PPOUT =

 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
- **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”

\[ \text{PPOUT: we want to place at output of this block only if} \]
\[ \quad \text{– we want to place at entry of all successors} \]
\[ \quad \text{PPOUT}[i] = \begin{cases} 
0 & i = \text{entry} \\
\cap PPIN[s] & \text{otherwise} 
\end{cases} \quad s \in \text{succ}(i) \]

\[ \text{PPIN: we want to place at start of this block only if (all of)}: \]
\[ \quad \text{– we have a local computation to place, or a placement at the end of} \]
\[ \quad \text{this block which we can move up} \]
\[ \quad \text{– we want to move computation to output of all predecessors where} \]
\[ \quad \text{expression is not already available (don’t insert at input)} \]
\[ \quad \text{– we gain something by moving it up (PAVIN heuristic)} \]
\[ \quad \text{PPIN}[i] = \begin{cases} 
0 & i = \text{exit} \\
\cap \left( \text{ANTLOC}[i] \cup (\text{PPOUT}[i] - \text{KILL}[i]) \right) \cap \left( \text{PPOUT}[p] \cup \text{AVOUT}[p] \right) \cap \text{PAVIN}[i] & \text{otherwise} 
\end{cases} \quad p \in \text{preds}(i) \]
“Placement Possible” Example 1

\[
t_1 = a + b
\]

\[
a = \ldots
\]

\[
t_2 = a + b
\]

<table>
<thead>
<tr>
<th>KILL</th>
<th>PAVIN</th>
<th>PPIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KILL</th>
<th>AVLOC</th>
<th>PAVOUT</th>
<th>PPOUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KILL</th>
<th>AVLOC</th>
<th>PAVOUT</th>
<th>PPOUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
“Placement Possible” Example 2

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

\[ a = \ldots \]

\[ t_2 = a + b \]

KILL = 1 \hspace{1cm} PAVIN = 0 \hspace{1cm} PPIN = \\
AVLOC = 1 \hspace{1cm} PAVOUT = 1 \hspace{1cm} PPOUT = \\
ANTLOC = 0 \hspace{1cm} AVOUT = 1 \hspace{1cm} PPOUT = \\

KILL = 1 \hspace{1cm} PAVIN = 0 \hspace{1cm} PPIN = \\
AVLOC = 0 \hspace{1cm} PAVOUT = 0 \hspace{1cm} PPOUT = \\
ANTLOC = 0 \hspace{1cm} AVOUT = 0 \hspace{1cm} PPOUT = \\

KILL = 0 \hspace{1cm} PAVIN = 1 \hspace{1cm} PPIN = \\
AVLOC = 1 \hspace{1cm} PAVOUT = 1 \hspace{1cm} PPOUT = \\
ANTLOC = 1 \hspace{1cm} AVOUT = 1 \hspace{1cm} PPOUT = \\
“Placement Possible” Correctness

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

\[
\text{PPIN}[i] \subseteq (\text{PPOUT}[i] - \text{KILL}[i]) \cup \text{ANTLOC}[i]
\]

\[
\text{PPOUT}[i] = \begin{cases} 
0 & i = \text{exit} \\
\bigcap_{s \in \text{succ}(i)} \text{PPIN}[s] & \text{otherwise}
\end{cases}
\]

- **INSERT** $\subseteq$ **PPOUT** $\subseteq$ **ANTOUT**, so insertion is safe.

- **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 of data flow]

  - Doesn't find some eliminations:

    ![Diagram of data flow]

- 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.