Partial Redundancy Elimination

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design
Partial Redundancy Elimination Transformation

(a)  
1. \(a = c\)  
2. \(x = a + b\)  
3.  
4. \(y = a + b\)  
   \(b = d\)

(b)  
1. \(a = c\)  
2. \(h = a + b\)  
3.  
4. \(y = h\)  
   \(b = d\)
Some Definitions

1. Partially redundant computation (prc)
   - A computation which is performed twice in a certain path

2. Partial redundancy elimination
   - Involves insertions and deletions of computations to ensure that no prc's exist

3. Safety
   - No introduction of computations of new values on any path in the program
Previous Work

1. Morel and Renvoise’s algorithm
   - Bidirectional dataflow analysis, complicated
   - Does not eliminate all \( prc' \)s
   - Redundant code motion (without gain)

2. Dhamdhere and others improved this algorithm

3. Knoop and Steffen’s algorithm
   - Unidirectional dataflow analyses, computationally optimal
   - No redundant code motion
   - Needs some blocks/edges to be split in the beginning
   - It is somewhat unintuitive and complex
Highlights of Our algorithm

- Simple and intuitive, with four unidirectional flows
- Computationally and lifetime optimal
- No edge splitting in the beginning; it is needed only at the end to insert computations
- Yields points of insertion and replacement directly
- Introduces the notions of safe partial availability and safe partial anticipability
Highlights of Our algorithm

- Every safe partially redundant computation offers scope for redundancy elimination
- Any safe partially redundant computation at a point can be made totally redundant by insertion of new computations at proper points
- Computation of any expression that is totally redundant can be replaced by a copy rule
- After the transformation, no expression is recomputed at a point if its value is _available_ (not partially) from previous computations
Properties of Expressions at a Point $p$

- **Availability**
  - Computed along *all* paths reaching $p$ from the start node, with no changes to operands

- **Partial availability**
  - Computed along *atleast* one path to $p$

- **Anticipability**
  - Computed along *all* paths starting from $p$ to the end node, with no changes to operands

- **Partial anticipability**
  - Computed along *atleast* one path from $p$
Partial Availability and Anticipability

Fig. (a) and Fig. (b) - $a + b$ is partially available at entry to 4
Fig. (a) - $a + b$ is partially anticipable at exit of 1
Fig. (b) - $a + b$ is anticipable at exit of 1

\[ a + b \]

(a)

(b)
Properties of Expressions at a Point $p$

- **Safety**
  - Either available or anticipable $p$

- **Safe partial availability**
  - All points on the path of availability from the last computation of the expression to $p$ are safe

- **Safe partial anticipability**
  - All points on the path of anticipability from $p$ to the first computation of the expression are safe

- **Safe partially redundant computation**
  - Locally anticipable and safe partially available at the entry of the node
Safe Partially Available/Anticipable Computation

Fig.(a) - $a + b$ is safe partially anticipable at entry to 3
Fig.(b) - $a + b$ is safe partially available at entry to 4
Safe Partially Redundant Computation

Fig. (a) - \( a + b \) is not safe partially available at entry to 4
Fig. (a) - \( a + b \) is not safe partially anticipable at exit of 1
Fig. (b) - \( a + b \) is safe partially redundant in 4
Special Computations in a Basic Block $i$

- $FIRST_i$
  - Computation before the first modification of operands (from top)
- $LAST_i$
  - Last computation after which no modification of operands takes place

All local redundancies are assumed to have been eliminated already.

Hence, there exist at most one $FIRST_i$ and one $LAST_i$.

All other computations of the same expression are in between these two and are irrelevant to the algorithm.
FIRST and LAST Computations

\[
\begin{align*}
  x &= a + b \\
  y &= a + b
\end{align*}
\]

(no modifications to \(a\) and \(b\) here)

Such situations do not occur since local CSE has been carried out

\[
\begin{align*}
  x &= a + b \\
  a &= \ldots \\
  b &= \ldots \\
  z &= a + b \\
  a &= \ldots \\
  b &= \ldots \\
  y &= a + b
\end{align*}
\]

The modifications to \(a\) and \(b\), and \(z = a + b\) are not relevant. Only \(x = a + b\) and \(y = a + b\) are relevant

(FIRST and LAST computations)
Outline of the Algorithm

Our PRE algorithm identifies all safe PRCs and makes them totally redundant by suitable insertions

1. Compute the predicates, $AV_i$, $ANT_i$, $SAFE_i$, $SPAV_i$, and $SPANT_i$ at entry and exit points of all nodes

2. Mark all points which have both $SPAV$ and $SPANT$ true and consider the paths formed by connecting such adjacent marked points

3. Insertion points: just before $LAST$ in starting points of these paths

4. Insertion edges: those that enter junction nodes on these paths

5. Replacements are for $LAST$ and $FIRST$ computations in the starting and ending points of these paths
Partial Redundancy Transformation

Fig.(a) - $a + b$ is safe partially redundant in 4
Fig.(b) - $a + b$ is made totally redundant by the new block
Local Properties

- **TRANSP$_i$** (transparency)
  - True for an expression in a node $i$, if its operands are not modified by the execution of statements in node $i$

- **COMP$_i$** (locally available)
  - True if there is at least one computation of the expression in $i$ and no modification of operands takes place during and after the computation

- **ANTLOC$_i$** (locally anticipable)
  - True if there is at least one computation of the expression in $i$ and no modification of the operands takes place before the first computation

Y.N. Srikant  Partial Redundancy Elimination
Local Properties

TRANS

No assignments to a and b here

COMP

\[ x = a + b \]

No assignments to a and b here

x cannot be a or b

ANTLOC

No assignments to a and b here

x = a + b

a+b is the expression under consideration
Global Properties

Availability

\[
AVIN_i = \begin{cases} 
\text{FALSE} & \text{if } i = s \\
\prod_{j \in \text{pred}(i)} AVOUT_j & \text{otherwise}
\end{cases}
\]

\[
AVOUT_i = \text{COMP}_i + AVIN_i \cdot \text{TRANSP} \]

Anticipability

\[
\text{ANTOUT}_i = \begin{cases} 
\text{FALSE} & \text{if } i = e \\
\prod_{j \in \text{succ}(i)} \text{ANTIN}_j & \text{otherwise}
\end{cases}
\]

\[
\text{ANTIN}_i = \text{ANTLOC}_i + \text{ANTOUT}_i \cdot \text{TRANSP}
\]

Safety

\[
\text{SAFEIN}_i = AVIN_i + \text{ANTIN}_i
\]

\[
\text{SAFEOUT}_i = AVOUT_i + \text{ANTOUT}_i
\]
Global Properties

Safe Partial availability

\[ \text{SPAVIN}_i = \begin{cases} \text{FALSE} & \text{if } i = s \text{ or } \neg \text{SAFEIN}_i \\ \sum_{j \in \text{pred}(i)} \text{SPAVOUT}_j & \text{otherwise} \end{cases} \]

\[ \text{SPAVOUT}_i = \begin{cases} \text{FALSE} & \text{if } \neg \text{SAFEOUT}_i \\ \text{COMP}_i + \text{SPAVIN}_i \cdot \text{TRANSP}_i & \text{otherwise} \end{cases} \]

Safe Partial anticipability

\[ \text{SPANTOUT}_i = \begin{cases} \text{FALSE} & \text{if } i = e \text{ or } \neg \text{SAFEOUT}_i \\ \sum_{j \in \text{succ}(i)} \text{SPANTIN}_j & \text{otherwise} \end{cases} \]

\[ \text{SPANTIN}_i = \begin{cases} \text{FALSE} & \text{if } \neg \text{SAFEIN}_i \\ \text{ANTLOC}_i + \text{SPANTOUT}_i \cdot \text{TRANSP}_i & \text{otherwise} \end{cases} \]
Global Properties

- **Safe Partial Redundancy**
  - For $FIRST_i$ (at entry of i)
    \[ SPREDUND_i = ANTLOC_i \cdot SPAVIN_i \]
  - $LAST_i$, when it is distinct from $FIRST_i$, cannot be safe partially redundant, because the computations of the expression between these makes $ANTLOC_i$ false

- **Total Redundancy**
  - For $FIRST_i$
    \[ REDUND_i = ANTLOC_i \cdot AVIN_i \]
  - For $LAST_i$
    \[ REDUND_i = COMP_i \cdot AV_p, \]
    where $p$ is the point just before $LAST_i$
Global Properties

- **Isolatedness**
  A computation is *isolated*, if it is neither safe partially available nor safe partially anticipable at that point

\[
\text{ISOLATED}_i = \text{ANTLOC}_i \cdot \neg \text{SPAVIN}_i \cdot \neg (\text{TRANS}_i \cdot \text{SPANTOUT}_i)
\]

\[
\text{ISOLATED}_i = \text{COMP}_i \cdot \neg \text{SPANTOUT}_i \cdot \neg (\text{TRANS}_i \cdot \text{SPAVIN}_i)
\]
**Predicates for Insertion**

**$\text{INSERT}_i$**
- True if the point just before the LAST computation in block $i$ is an insertion point
- Interpretation of $\text{INSERT}_i$:
  
  $\text{(expr should be computed in } i\text{) AND (expr should be useful later) AND ((operands should be modified in } i\text{) OR (expr should not be available from above))}$

- This is possible only for the first node on the path and those intermediate nodes where the operands of the expr are modified and the expr is recomputed

$$\text{INSERT}_i = \text{COMP}_i . \text{SPANTOUT}_i . (\neg \text{TRANSP}_i + \neg \text{SPAVIN}_i)$$

**$\text{INSERT}_{(i,j)}$**
- True if a computation should be inserted by splitting the edge $(i, j)$

$$\text{INSERT}_{(i,j)} = \neg \text{SPAVOUT}_i . \text{SPAVIN}_j . \text{SPANTIN}_j$$
Predicates for Replacement

$REPLACE_{i_f}$ (respectively $REPLACE_{i_l}$)

- True if $FIRST_i$ (respectively $LAST_i$) should be replaced

$$REPLACE_{i_f} = \text{ANTLOC}_i.(SPAVIN_i + TRANSP_i.SPANTOUT_i)$$

$$REPLACE_{i_l} = \text{COMP}_i.(SPANTOUT_i + TRANSP_i.SPAVIN_i)$$
Example 1

For Blocks 1a and 1b
comp = T; transp = T
antloc = T

1a
\[ z = a + b \]
1b
\[ x = a + b \]

2

For Block 3
comp = T
transp = F
antloc = T

m = a + b
\[ a = 5 + c \]
\[ b = 6 + d \]
\[ n = a + b \]

3

\[ \text{INSERT}_{1a} = T \cdot (F + T) = T \]
\[ \text{REPLACE}_{1af} = T \cdot (F + T \cdot T) = T \]
\[ \text{REPLACE}_{1al} = T \cdot (T + T \cdot F) = T \]
\[ \text{INSERT}_{1b} = T \cdot (F + F) = F \]
\[ \text{REPLACE}_{1bf} = T \cdot (T + T \cdot T) = T \]
\[ \text{REPLACE}_{1bl} = T \cdot (T + T \cdot T) = T \]
\[ \text{INSERT}_{(2,3)} = T \cdot T \cdot T = T \]
\[ \text{INSERT}_{(5,1b)} = T \cdot T \cdot T = T \]
\[ \text{INSERT}_3 = T \cdot (T + F) = T \]
\[ \text{REPLACE}_{3f} = T \cdot (T + F \cdot T) = T \]
\[ \text{REPLACE}_{3l} = T \cdot (T + T \cdot T) = T \]
\[ \text{INSERT}_4 = F \cdot F \cdot (T + F) = F \]
\[ \text{REPLACE}_{4f} = T \cdot (T + F \cdot F) = T \]

4

\[ \text{INSERT}_i = \text{COMP}_i \cdot \text{SPANTOUT}_i \cdot (!\text{TRANS}_i + !\text{SPAV}_i) \]
\[ \text{INSERT}_{(i,j)} = !\text{SPAV}_i \cdot \text{SPAV}_j \cdot \text{SPANT}_j \]

\[ \text{REPLACE}_{if} = \text{ANTLOC}_i \cdot (\text{SPAV}_i \cdot \text{TRANS}_i \cdot \text{SPANTOUT}_i) \]
\[ \text{REPLACE}_{ii} = \text{COMP}_i \cdot (\text{SPANTOUT}_i + \text{TRANS}_i \cdot \text{SPAV}_i) \]
Example 1

(a) z = a+b

(b) h1 = a+b

Y.N. Srikant
Partial Redundancy Elimination
Example 2

Transformed program

1

2

y = a + b
a = c
x = a + b

3

6

7

h = a + b
y = h

4

5

8

9

10

h = a + b
z = h
a = c
x = a + b

Y.N. Srikant
Partial Redundancy Elimination
Example 2

Solution:
- Insertion just before the \textit{last computation} in node 7
- Insertion on edge (3, 5)
- Replacement of the \textit{first computation} in nodes 4, 7, and 8
- Replacement of the \textit{last computations} in nodes 4 and 7

Question:
- Why should we not split edge (1,3) and place the computation $h = a + b$? Why only on the edge (3,5)?

Answer:
- It is not safe. The path 1-3-10 had no computation of $a + b$ before transformation and by placing a computation on the edge (1,3), we are introducing one
- However, this solution works for all “valid” inputs
Correctness Results

Lemma 1 All insertions of computations corresponding to the transformation are done at safe points.

Lemma 2 All candidate computations which are safe partially redundant become totally redundant after insertions corresponding to the transformation.

Lemma 3 Only those candidate computations which would be redundant after insertions corresponding to the transformation are replaced.

Lemma 4 After the transformation no path contains more computations of an expression than it contained before.

Theorem 1 The algorithm performs partial redundancy elimination correctly.
**Optimality Results**

**Lemma 5** A candidate computation is not replaced by the transformation if and only if it is an isolated computation.

**Theorem 2** The transformation is computationally optimal, *i.e.*, there does not exist any other correct transformation with less number of computations of an expression on any path.

**Theorem 3** The transformation is lifetime optimal, *i.e.*, the transformation keeps the live ranges of the newly introduced temporaries to the minimum.