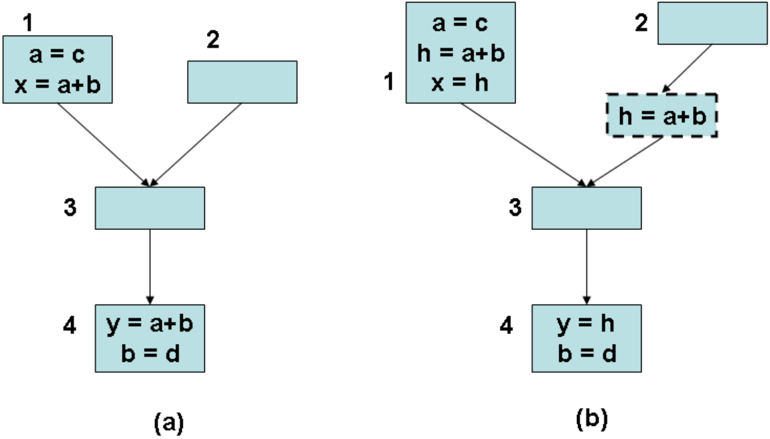
Partial Redundancy Elimination

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

Partial Redundancy Elimination Transformation



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 some what 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

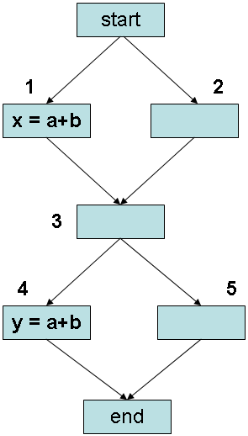
- Every safe partially redundant computation offers scope for redundancy elimination
- Any safe partially redundant computation at a point can be made totally redundant by insertion of new computations at proper points
- Computation of any expression that is totally redundant can be replaced by a copy 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

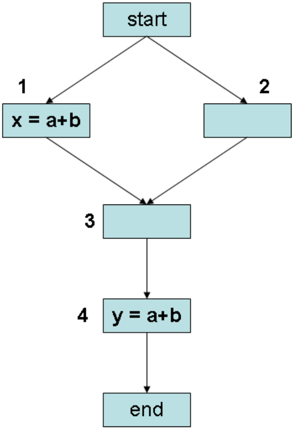
- *Availability*
  - Computed along *all* paths reaching p from the start node, with no changes to operands
- *Partial availability*
  - Computed along *atleast* one path to p
- *Anticipability*
  - Computed along *all* paths starting from p to the end node, with no changes to operands
- *Partial anticipability*
  - 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)



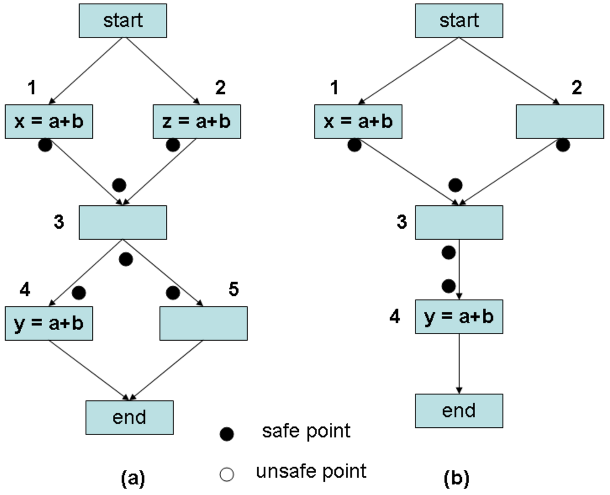
(b)

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

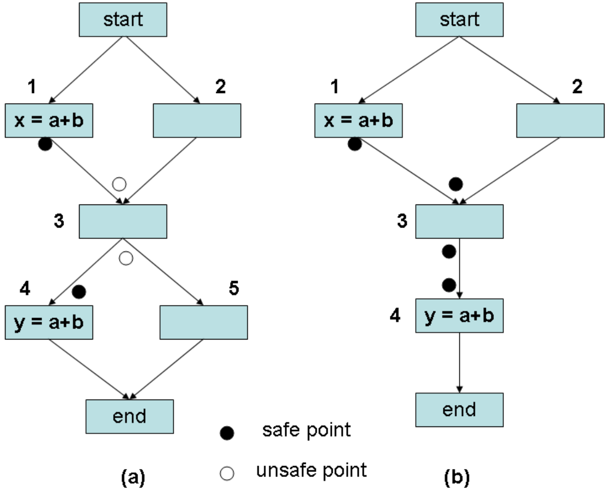
Safe Partially Available/Anticipable Computation

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

x = a+b
(no modifications to *a* and *b* *here*)
y = a+b

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

x = a+b
a = ...
b = ...
z = a+b
a = ...
b = ...
y = a+b

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)

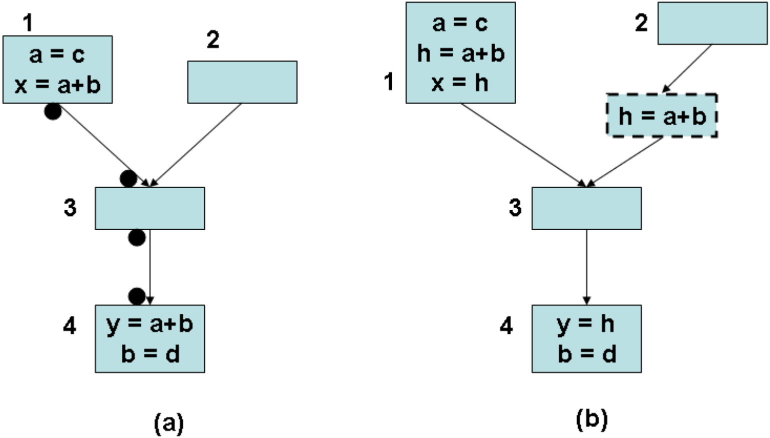
Outline of the Algorithm

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 atleast 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 atleast one computation of the expression in i and no modification of the operands takes place before the first computation

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

Availability

$$
\begin{aligned}
AVIN_i &= \begin{cases} FALSE & \text{if } i = s \\ \prod_{j \in pred(i)} AVOUT_j & \text{otherwise} \end{cases} \\
AVOUT_i &= COMP_i + AVIN_i.TRANSP_i
\end{aligned}
$$

Anticipability

$$
\begin{aligned}
ANTOUT_i &= \begin{cases} FALSE & \text{if } i = e \\ \prod_{j \in succ(i)} ANTIN_j & \text{otherwise} \end{cases} \\
ANTIN_i &= ANTLOC_i + ANTOUT_i.TRANSP_i
\end{aligned}
$$

Safety

$$
\begin{aligned}
SAFEIN_i &= AVIN_i + ANTIN_i \\
SAFEOUT_i &= AVOUT_i + ANTOUT_i
\end{aligned}
$$

Global Properties

Safe Partial availability

$$SPAVIN_i = \begin{cases} FALSE & \text{if } i = s \text{ or } \neg SAFEIN_i \\ \sum_{j \in pred(i)} SPAVOUT_j & \text{otherwise} \end{cases}$$

$$SPAVOUT_i = \begin{cases} FALSE & \text{if } \neg SAFEOUT_i \\ COMP_i + SPAVIN_i.TRANSP_i & \text{otherwise} \end{cases}$$

Safe Partial anticipability

$$SPANTOUT_i = \begin{cases} FALSE & \text{if } i = e \text{ or } \neg SAFEOUT_i \\ \sum_{j \in succ(i)} SPANTIN_j & \text{otherwise} \end{cases}$$

$$SPANTIN_i = \begin{cases} FALSE & \text{if } \neg SAFEIN_i \\ ANTLOC_i & \\ +SPANTOUT_i.TRANSP_i & \text{otherwise} \end{cases}$$

Global Properties

- **Safe Partial Redundancy**
  - For $FIRST_i$ (at entry of i)

$$SPREDUND_{i_f} = ANTLOC_i.SPAVIN_i$$

  - $LAST_i$, 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 $FIRST_i$

$$REDUND_{i_f} = ANTLOC_i.AVIN_i$$

  - For $LAST_i$

$$REDUND_{i_l} = COMP_i.AV_p,$$

where p is the point just before $LAST_i$

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

$$ISOLATED_{i_f} = ANTLOC_i . \neg SPAVIN_i . \neg(TRANSP_i . SPANTOUT_i)$$

$$ISOLATED_{i_l} = COMP_i . \neg SPANTOUT_i . \neg(TRANSP_i . SPAVIN_i)$$

Predicates for Insertion

$INSERT_i$

- True if the point just before the LAST computation in block i is an insertion point
- Interpretation of $INSERT_i$:
  (*expr should be computed in i*) AND (*expr should be useful later*) AND ((*operands should be modified in i*) 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

$$INSERT_i \quad = \quad COMP_i.SPANTOUT_i.(\neg TRANSP_i + \neg SPAVIN_i)$$

$INSERT_{(i,j)}$

- True if a computation should be inserted by splitting the edge (i, j)

$$INSERT_{(i,j)} \quad = \quad \neg SPAVOUT_i.SPAVIN_j.SPANTIN_j$$

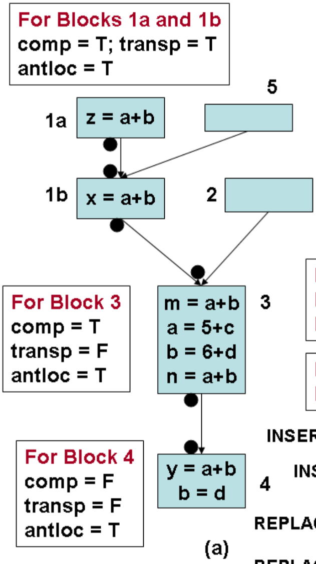
Predicates for Replacement

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

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

$$
\begin{aligned}
REPLACE_{i_f} &= ANTLOC_i.(SPAVIN_i + TRANSP_i.SPANTOUT_i) \\
REPLACE_{i_l} &= COMP_i.(SPANTOUT_i + TRANSP_i.SPAVIN_i)
\end{aligned}
$$

Example 1



$INSERT_{1a}$ = T.T.(F+T) = T
$REPLACE_{1af}$ = T.(F+T.T) = T
$REPLACE_{1al}$ = T.(T+T.F) = T

$INSERT_{1b}$ = T.T.(F+F) = F
$REPLACE_{1bf}$ = T.(T+T.T) = T
$REPLACE_{1bl}$ = T.(T+T.T) = T

$INSERT_{(2,3)}$ = T.T.T = T
$INSERT_{(5,1b)}$ = T.T.T = T

$INSERT_{3}$ = T.T.(T+F) = T
$REPLACE_{3f}$ = T.(T+F.T) = T
$REPLACE_{3l}$ = T.(T+.T) = T

$INSERT_{4}$ = F.F.(T+F) = F
$REPLACE_{4f}$ = T.(T+F.F) = T

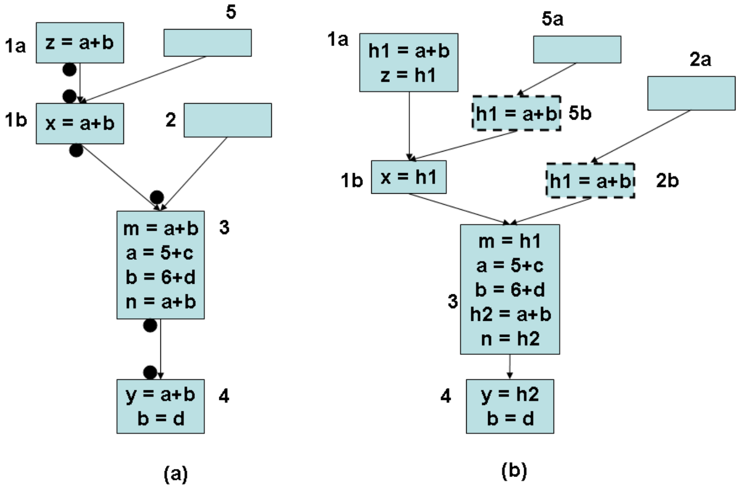
$INSERT_i = COMP_i .SPANTOUT_i .(!TRANSP_i + !SPAVIN_i)$

$INSERT_{(i,j)} = !SPAVOUT_i .SPAVIN_j .SPANTIN_j$

$REPLACE_{if} = ANTLOC_i .(SPAVIN_i + TRANSP_i .SPANTOUT_i)$

$REPLACE_{il} = COMP_i .(SPANTOUT_i + TRANSP_i :SPAVIN_i)$

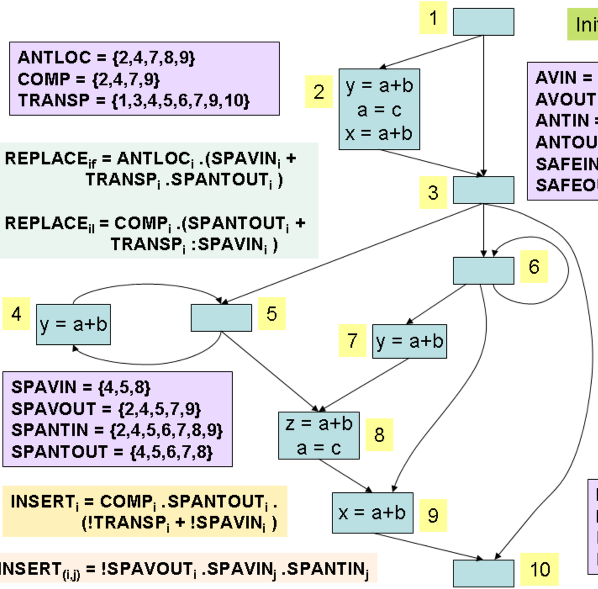
Example 1



5
1a
z = a+b
1b
x = a+b
2
m = a+b
a = 5+c
b = 6+d
n = a+b
3
y = a+b
b = d
4
(a)
5a
1a
h1 = a+b
z = h1
2a
h1 = a+b
5b
1b
x = h1
h1 = a+b
2b
m = h1
a = 5+c
b = 6+d
h2 = a+b
n = h2
3
4
y = h2
b = d
(b)

Example 2

Initial program



ANTLOC = {2,4,7,8,9}
COMP = {2,4,7,9}
TRANSP = {1,3,4,5,6,7,9,10}

AVIN = Φ
AVOUT = {2,4,7,9}
ANTIN = {2,4,5,6,7,8,9}
ANTOUT = {4,5,6,7,8}
SAFEIN = {2,4,5,6,7,8,9}
SAFEOUT = {2,4,5,6,7,8,9}

$REPLACE_{if} = ANTLOC_i .(SPAVIN_i + TRANSP_i .SPANTOUT_i)$

$REPLACE_{il} = COMP_i .(SPANTOUT_i + TRANSP_i :SPAVIN_i)$

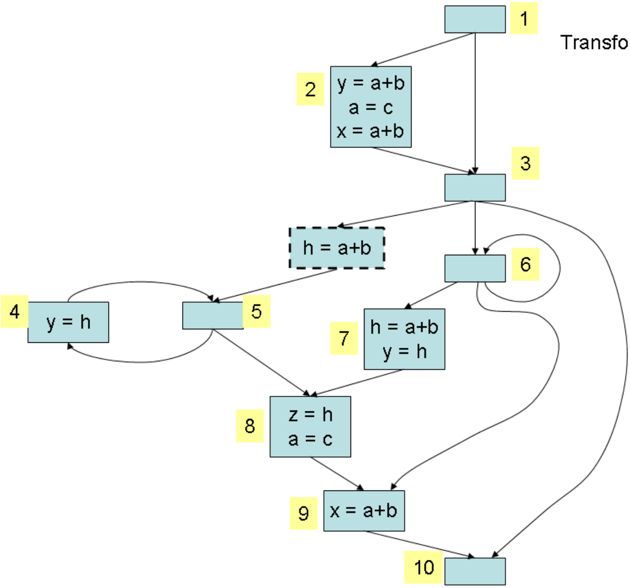
SPAVIN = {4,5,8}
SPAVOUT = {2,4,5,7,9}
SPANTIN = {2,4,5,6,7,8,9}
SPANTOUT = {4,5,6,7,8}

$INSERT_i = COMP_i .SPANTOUT_i .(!TRANSP_i + !SPAVIN_i)$

$INSERT_{(i,j)} = !SPAVOUT_i .SPAVIN_j .SPANTIN_j$

$INSERT_i$ = {7}
$INSERT_{(i,j)}$ = {(3,5)}
$REPLACE_{if}$ = {4,7,8}
$REPLACE_{il}$ = {4,7}

Example 2


1
Transformed program
2
y = a+b
a = c
x = a+b
3
h = a+b
6
4
y = h
5
7
h = a+b
y = h
8
z = h
a = c
9
x = a+b
10

Example 2

Solution:

- Insertion just before the *last computation* in node 7
- Insertion on edge (3, 5)
- Replacement of the *first computation* in nodes 4, 7, and 8
- Replacement of the *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 corr esponding to the transformation are replaced

Lemma 4 After the transformation no path contains more computations of an expression tha n 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