Introduction to PRE

Partial redundancy elimination

- discovers partially redundant expressions,
- converts to fully redundant, then eliminates redundancy

Tradeoffs with PRE

– data-flow properties are complex and non-intuitive, but . . .

+ dominates classical GCSE and LICM, and does more;

+ it minimizes auxiliary variable lifetimes

+ algorithms give strong guarantees about optimality

⇒ *every optimizing compiler should use it*

Introduction to PRE (cont’d)

References:

J. Knoop, O. Rüthing, and B. Steffen, “Lazy Code Motion,” In *Proc. ACM Symposium on Programming Language Design and Implementation*, 1992.

1. Muchnick, Chapter 13.3 (based on Knoop et al., above).
2. Effective Partial Redundancy Elimination, Preston Briggs and Keith Cooper, *PLDI 1994* (improves how distinct expressions are identified to find more redundancies).

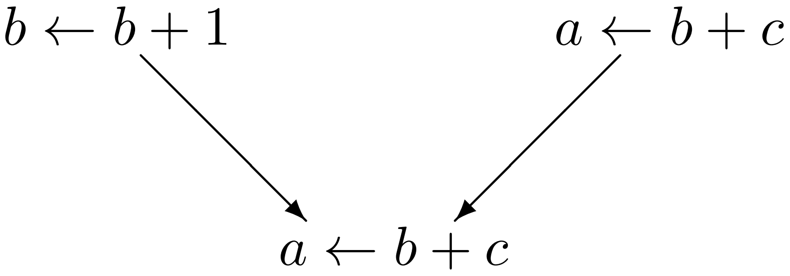
Additional Reading:

1. *Original paper:* E. Morel and C. Renvoise, “Global optimization by suppression of partial redundancies,” CACM 22(2), Feburary, 1979.
2. *Numerous Improvements*, e.g.,
    - Dreschler and Stadel, TOPLAS 10(4), 1988.
    - Dhamdhere TOPLAS, 13 (2), 1991 (practical adaptation).

Partially redundant expressions

An expression is *partially redundant* at p if it is *available* on some, but not all, paths reaching p

Example

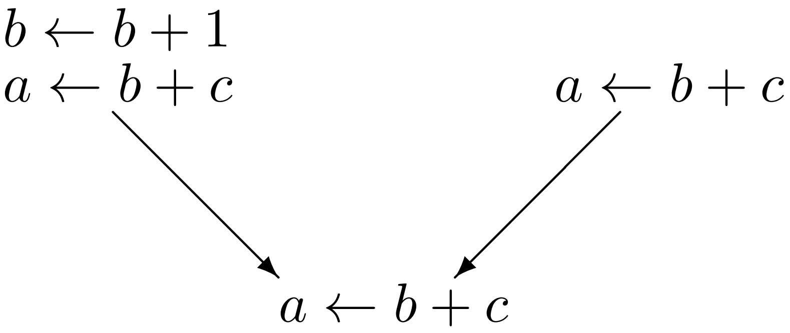


$b \leftarrow b + 1$

$a \leftarrow b + c$

$a \leftarrow b + c$

PRE inserts code to make $b + c$ fully redundant



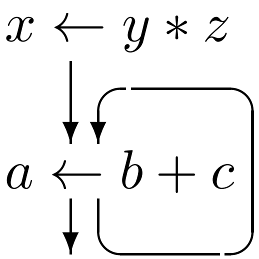
$b \leftarrow b + 1$
$a \leftarrow b + c$

$a \leftarrow b + c$

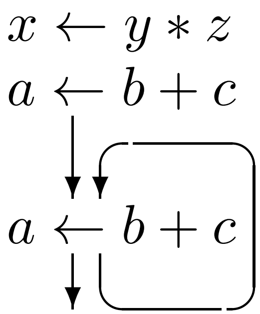
$a \leftarrow b + c$

Loop invariant expressions

Another example



before



after

PRE removes loop invariants

- invariant expression is partially redundant (or redundant)
- PRE converts it to full redundancy
- PRE removes redundant expression

What can be moved?

- ideally, both computation and assignment
- of course, computation is easier

High-level view

How does it work?

- Use five distinct data-flow problems
- Computationally optimal placement: *anticipatable, earliest*
- Lifetime-optimal placement: *latest* (in 2 steps), *isolated*
- Insert the code and remove the redundant expressions

Scope

PRE works with *lexically identical* expressions:

- expressions
- loads
- constants
- *not* stores, copies, or calls

Strategies to make PRE more effective

Problems

- Associativity, commutativity
- Algebraic identities
- Complex Expressions

Solution: Two pre-passes to make PRE more effective.

Effective Partial Redundancy Elimination, Preston Briggs and Keith Cooper, PLDI 1994*.*

Global Reassociation

- Reorder expressions to expose constants and loop-invariant terms, and to normalize order

Global Value Numbering

- Use algorithm similar to AWZ to identify “equivalent” variables
- Do *not* perform the GVN optimization: placement may be poor!

Problem: Critical Edges

Definition:

A *critical edge* in a flow graph is an edge from a node with multiple successors to a node with multiple predecessors.

So what's the problem?

Splitting critical edges

Split every edge leading to a node with more than one predecessor.
Note: Not just critical edges
⇒ sufficient to insert computations at node entries only

Overview of PRE Algorithm

Assume each basic block is a single statement.

Informally ...

1. **Anticipatable** : *aka "very busy," "down-safe"*
   e is anticipatable at p if
   $\Rightarrow$ Computing e at p would be useful along any path from p.

2. **Earliest** :
   e is *earliest* at p if there is some path from s to p on which e cannot be computed "anticipatably" and correctly
   i.e., there is some path q from s to n on which e is not anticipatable on any point on q, i.e.,
   - e would give a different value if computed at that point, or
   - e would not be used on some path from that point

   $\Rightarrow$ p is an earliest point to compute e.

Overview of PRE Algorithm (cont'd)

A computationally optimal placement

Compute expression e at each point p such that e is both *anticipatable* and *earliest* at p.
This is optimal in the sense that it:

- eliminates redundant expressions on every path from s to `exit`, and
- never computes an expression on any path on which it was not computed before.

Problem

May compute expressions very early on some paths
$\Rightarrow$ significantly increases register pressure

Improved PRE Algorithm - Lazy Code Motion

Informally . . .

3. Latest:
   p is a *latest* point to compute e if placing e at p is computationally optimal *AND*, on every path from p, any later optimal point on the path would be after some use of e.
   $\Rightarrow$ cannot move e later on any path from p
4. Isolated:
   p is an *isolated* point to compute e if it is optimal, and that value of e is only used immediately after p.
   $\Rightarrow$ unnecessary to allocate a new temporary at p

Improved PRE Algorithm - Lazy Code Motion (cont'd)

A lifetime-optimal placement

Compute expression e at each point p such that e is latest at p and not isolated at p.

$\Rightarrow$ *This is computationally optimal and requires the shortest lifetimes for all temporary variables introduced*

Preliminaries

Local properties of basic block b

Used : e is used in b if its value is computed in b

Killed : e is killed in b if any of its operands may be modified in b

transp(b) :

$e \in transp(b)$ if the operands of e are not modified in block b.
(We say block b is *transparent* to e).

Preliminaries (cont'd)

ANTloc(b) :

$e \in$ *ANTloc*(b) if e is computed at least once in b and its operands are not modified before its first computation.
(We say e is *locally anticipatable* in block b).
$\Rightarrow$ can move first evaluation to the start of b

Note that *ANTloc*(b) = Used(b) if b has a single statement.

Example :

a $\leftarrow$ b + c
d $\leftarrow$ a + e

the following properties hold

transp = {$\mathcal{U}$ $-$ {*expressions using a or d*} }
ANTloc = {b+c}

Anticipatable

ANT(p) :

$e \in \mathit{ANT}(p)$ if e is used before being killed on every path from point p to the `exit`.

e is *globally anticipatable* at point p.

(Also called *down-safe* at p.)

ANTin(b), ANTout(b) :

$\mathit{ANT}(p)$ at entry and exit of block b

$$\mathit{ANTout}(b) = \begin{cases} \phi & \text{if } b \text{ is an exit block} \\ \displaystyle\bigcap_{j \in \text{succ}(b)} \mathit{ANTin}(j) & \text{otherwise} \end{cases}$$

$$\mathit{ANTin}(b) = \mathit{ANTloc}(b) \bigcup (\mathit{ANTout}(b) \bigcap \mathit{transp}(b))$$

Earliestness

EARLin(b) :

$e \in \mathit{EARLin}(b)$ if there is some path q from s to ENTRY(b) where no node prior to b on q *both* evaluates e *and* produces the same value as e would produce at ENTRY(b).
We say e is *earliest* at ENTRY(b).

$$\mathit{EARLin}(b) = \begin{cases} \mathcal{U} & \text{if } b = s \\ \bigcup\limits_{j \in \text{pred}(b)} \mathit{EARLout}(j) & \text{otherwise} \end{cases}$$

$$\mathit{EARLout}(b) = \overline{\mathit{transp}(b)} \bigcup (\mathit{EARLin}(b) \bigcap \overline{\mathit{ANTin}(b)})$$

Theorem 3.9:

It is **computationally optimal** to compute e at entry to block b if $e \in \mathit{ANTin}(b) \bigcap \mathit{EARLin}(b)$

Delayedness

Used to compute *Latest*

DELAY(p) :

$e \in$ *DELAY* (p) if, for every path from s to p, there is a Safe-Earliest computational point of e on the path (may be p itself), say p_{SE}, and there are no subsequent uses of e on that path (i.e., between p_{SE} and p)

Think: e should be *delayed* at least up to point p.
Note: DELAY(p) preserves down-safety, and therefore preserves computational optimality! *Why?*

Delayedness (cont'd)

DELAYin(b), DELAYout(b) :

$\mathit{DELAY}(p)$ at entry and exit of b respectively.

$$\mathit{DELAYin}(b) \;=\; (\mathit{ANTin}(b) \bigcap \mathit{EARLin}(b)) \bigcup \begin{cases} \phi & \text{if } b = s \\ \bigcap_{j \in \text{pred}(b)} \mathit{DELAYout}(j) & \text{otherwise} \end{cases}$$

$$\mathit{DELAYout}(b) = \mathit{DELAYin}(b) \bigcap \overline{\mathit{ANTloc}(b)}$$

Latestness

LATEST(p) :

$e \in$ *LATEST* (p) if p is a *computationally optimal* point for computing e and, for every path q from p to `exit`, any later optimal point on q occurs after a use of e on q.
Say: e is *latest* at point p

LATESTin(b) :

LATEST (p) at entry to block b.

$$\mathit{LATESTin}(b) = \mathit{DELAYin}(b) \bigcap \left(\mathit{ANTloc}(b) \bigcup \overline{\bigcap_{j \in \text{succ}(b)} \mathit{DELAYin}(j)}\right)$$

Isolatedness

ISOLout(b) :

$e \in \mathit{ISOLout}(b)$ if and only if, on every path from a successor block to `exit`, a use of e is preceded by an optimal computation point of e.

ISOLin(b) :

Similar, but think of b itself as being a successor block of the entry point of b.

$$\mathit{ISOLout}(b) = \begin{cases} \phi & \text{if } b \text{ is an exit block} \\ \bigcap_{j \in \text{succ}(b)} \mathit{ISOLin}(j) & \text{otherwise} \end{cases}$$

$$\mathit{ISOLin}(b) = \mathit{LATESTin}(b) \bigcup (\mathit{ISOLout}(b) \bigcap \overline{\mathit{ANTloc}(b)})$$

Culmination

The lifetime-optimal points

$$\mathit{OPT}(b) \quad = \quad \mathit{LATESTin}(b) \bigcap \overline{\mathit{ISOLout}(b)}$$

The redundant expressions

$$\mathit{REDN}(b) \quad = \quad \mathit{ANTloc}(b) \bigcap \overline{\mathit{LATESTin}(b) \bigcap \mathit{ISOLout}(b)}$$

The Transformation for an Expression e

- Introduce a new auxiliary variable, h, for e
- Replace e with h in each block b such that $e \in \mathit{REDN}(b)$
- Insert $h = e$ at entry of each node b such that $e \in \mathit{OPT}(b)$

Implementation of Lazy Code Motion

Dataflow Analyses

- Use bit vectors and the iterative algorithm
- Use actual basic blocks, not individual statements
- Again, number the expressions *carefully*
  - Use global reassociation and global value numbering
  - Textually identical expressions → same number

Code generation

- Split critical edges and other edges to blocks with multiple successors
- May need to insert code at entry *or interior* of basic blocks
- *Important*: Check for zero-trip loops when moving computations out of loops