6.0 Partial Redundancy Elimination (PRE)

6.1 Introduction

\[ a_0 = \ldots \]
\[ b_2 = \ldots \]

\[ B_0 \]

\[ B_1 \]

\[ x_3 = (x_0, x_1) \]

\[ B_3 \]

\[ E_1 : a_1 + b_2 \]

\[ E_2 : a_1 + b_2 \]

\[ a_0 + b_2 \] is not a common subexpression
Inserting a copy of \( a \times b_2 \) into the pair \( B_0 - B_1 - B_2 \) allows us to recognize \( a \times b_2 \) in \( B_3 \) as a common sub-expression.

\( \leq \) candidate for optimization.

An expression \( E \) that is evaluated along all paths to some block \( B \) is called fully redundant. (in \( B \))

(\( E \) must have same operands \& operators on all paths)

An expression \( E \) that is evaluated along some path(s) to some block \( B \) is called partially redundant.

Idea: insert copies so that \( E \) goes from partially redundant \( \rightarrow \) fully redundant.
PRE using SSA (SSAPRE)

• by adding operations or expected optimization (remove some cycles)

• common subexpression elimination (CSE, shown to be beneficial in many cases) - fully redundant expression elimination is a special case of PRE

• potential of big payoff, not many changes to compile infrastructure

We try to make partially redundant expressions fully redundant - by inserting copies
2 questions for the compiler

when? can an expression be inserted
- proper use of the expression

when? should a copy be inserted
- blocks where in the block

Starting point: compiler identifies expression that are (at least partially) available at some block B

there exist at least one path that computes expression

We use cost function & constraints to select insertion points.
6.2 SSA and expressions

We introduce versions for expression $E_i$:
- $a \Theta_1$ may be $a_{1} + b_{2} \sim a_{3} + b_{5} \rightarrow$ different expression $E_2$

...to argue/reason about availability of expressions.

We introduce a $\Phi$ function (uppercase $\Phi$) for expressions to deal with different versions of expressions.

$\bot$ ("bottom") - special symbol to indicate that there is no version (of some expression $E$) computed along a path.
simple example

\[ a, = \]
\[ b, = \]

\[ E_1 = a_1 + b_2 \]
\[ x_1 = E_1 \]

\[ E_2 = \Phi (x_0, E_1) \]
\[ x_2 = \Phi (x_0, x_1) \]
\[ z_2 = E_2 \]
Before SSA can be done in five steps
- assume SSA for scalars

1) Insert \( \Phi \) nodes
   - record places where different versions of expressions come together

2) Insert version numbers for expressions
   - either on left-hand side of \( \Phi \) node
   - or at an assignment stmt

3) Identify places when it is legal to insert a copy of some expression \( E \)

4) Find place(s) where insertion of a copy of \( E \) is profitable

5) Transform the program based on (what we now) fully redundant expressions.
6.3 Insertion of $\Phi$ nodes

- given a CFG for a program, dominator tree, dominator frontier, $\Phi$ nodes for scalars are inserted, version numbers identified (SSA)

Consider an expression $E = a + b$ in some block $B$.

We must insert a $\Phi$ node (somewhere) if

- $E$ is computed explicitly
- if "$a" or "$b" change in $B$
If expression \( E \) is computed

\[ E = a + b \]

Find \( DF(B) \): \( \{ B_1 \} \)

(Here \( B_1, B_2 \))

Insert a \( I \) node in \( B_1 \)

\[ E = \Phi(, , ) \]

\[ E = \Phi(, , ) \]

\[ B_2 \] [unless there is already an \( I \) node for \( E \) in \( B_i \)]

(This skip covers all explicit computations of expression \( E \) in the program.)
Redefinition of operands

\[ E = a_1 + b_2 \]

If a block \( B \) contains a node for an operand of \( E \) (say "a")
then insert a \( \Phi \) node for \( E \) into block \( B \)

\[ a_3 = \phi(a_2, a_2) \]
\[ E = \Phi(, , ) \]
\[ = a_3 + b_2 \]

(we do this for all expressions \( E \) that contain "a" as an operand)
6.4 Produce version numbers for expression E

- If E appears on the left hand side, get a new version number
- Identify operands of I node (operands are versions of E for
  \[ E = I(\ldots, \ldots) \])

(Recall: for \( \Phi \) nodes for xdata we used an array of stacks
  
  \( \text{stack of ints (versions)} \)
  
  one entry for each scalar)

As expressions involve \( \geq 1 \) operand (here: 2) we need the array of stacks for scalars to identify
  
  \( \text{correct version of the operands of } E \).
  
  (need access to versions of all operands of } E)
We use an array of stacks for expressions. Each stack to indicate current version.

> one entry for each expression

Given an expression $E = a + b$ in some block $B$.

After $\phi$ node processing, we know versions of "$a" and "$b" used in block $B$. Say $a_i$ and $b_j$.

$$E = a_i + b_j$$

If we have seen "$a + b" before then there exists a version of $E$ (say $E_k$) - $E_k$ on top of stack for $E$ (entry in the array of stacks for expressions)
Question:
Is \( E_u \) the correct version for
\[ E = a_i + b_j \]

We process the basic blocks in the CFG in bottom-up order of the dominator tree (like we did for processing \( \Phi \) nodes).

For \( E = a + b \):
- Stack in array of stacks for scolars.
- if (TOP(\( \text{stack-}a \)) = i) and (TOP(\( \text{stack-}b \)) = j) and (TOP(\( \text{stack-}E \)) = k)
  
  then we can use version \( E_u \) of \( E \) and
  
  link occurrence of "atb" to \( E_k \)
If $E_k$ does not use $a_i$ (or does not use $b_i$) then compiler must create a new version of $E$ (say $E_{k+1}$)

$$E_{(k+1)} = a_1 + b_1$$
To identify correct version of \( \Phi \) nodes consider predecessor blocks

\[
E = \Phi (\ldots, \ldots)
\]

check \( x \times (y) \) for a version of \( E \) or \( \forall \) \( x \neq y \)

use \( E_2 \) as first operand for \( \Phi \) inside.

If there is no version (there is no comparable at \( E \))

use \( \bot \) (bottom) as operand in \( \Phi \) node