263-2810: Advanced Compiler Design

6.0 Partial redundancy elimination

*Thomas R. Gross*

Computer Science Department
ETH Zurich, Switzerland
Example

- Expression $E = a_1 + b_1$ in BB1
- Is $E$ in BB3 a “common sub-expression”?
Example

- Expression $E = a_1 + b_1$ in BB1
- $E$ is *not* a “common sub-expression” BB3.
- **Cannot re-use value of E**
  - E in BB3 may not be available

- **Assume program is executed repeatedly**
  - Often we have profile information from past executions
  - In a Virtual Machine -- may at time $T$ have profile information for time $[0, \ldots T)$
Execution frequency considered

- Expression $E = a_1 + b_1$ in BB1 recomputed again and again
- $E$ is not a “common sub-expression” BB3 but we wish it was
Could save re-computation of $E = a_1 + b_1$ in BB1 if path BB0 – BB2 – BB3 contained computation of $E$.

- Sometimes we can arrange this to be the case

We say $E$ is *partially available* at BB3

- Inserting a copy of $E = a_1 + b_1$ into path makes $E$ *fully available* (i.e., redundant)
- Fully available expressions can be optimized
- Common sub-expressions are a special case of fully available expressions
Execution frequency considered

- Expression $E = a_1 + b_1$ is now redundant at BB3
  - Copy inserted into BB2
PRE: Partial redundancy elimination

- An expression $E$ is partially available at point $P$ (in a CFG) if $E$ is computed along some path to $P$.
- An expression $E$ is fully available at point $P$ if $E$ is computed along all paths to $P$.

PRE: change partially redundant into fully redundant

- PRE: Powerful approach that combines
  - Common sub-expression elimination
  - Code hoisting
  - Loop invariant removal
  - Creation of redundancies
- **PRE idea:** insert *copies* to turn partially available expressions into fully available expressions
  - Then optimize by exploiting redundancy
  - Covers also expressions that were fully available to begin with

- **Good idea but**
  - Are there risks ?
  - Can we insert expressions everywhere? .... somewhere?
PRE

- Identify places to insert a copy of an expression
  - Source program: \( E = a + b \)
  - Need to distinguish between different versions of source operands \( a_i, b_j \)
  - \( \Rightarrow \) different versions of expressions

- Transformations to exploit redundancy
Expressions

\[ a_1 = \ldots \]
\[ b_1 = \ldots \]
\[ x_0 = a_1 + b_1 \]

\[ a_2 = \ldots \]
\[ y_1 = a_2 + b_1 \]

\[ b_2 = \ldots \]
\[ z_1 = a_1 + b_2 \]

\[ a_3 = \phi(a_2, a_1) \]
\[ b_3 = \phi(b_1, b_2) \]
\[ y_2 = \phi(y_1, y_0) \]
\[ z_2 = \phi(z_0, z_1) \]
\[ t_9 = a_3 + b_3 \]
Expressions

- Expression $E = a + b$
- Different in BB0, BB1, BB2, BB3

\[
\begin{align*}
a_1 &= \ldots \\
b_1 &= \ldots \\
x_0 &= a_1 + b_1
\end{align*}
\]

\[
\begin{align*}
a_2 &= \ldots \\
y_1 &= a_2 + b_1
\end{align*}
\]

\[
\begin{align*}
b_2 &= \ldots \\
z_1 &= a_1 + b_2
\end{align*}
\]

\[
\begin{align*}
a_3 &= \phi(a_2, a_1) \\
b_3 &= \phi(b_1, b_2) \\
y_2 &= \phi(y_1, y_0) \\
z_2 &= \phi(z_0, z_1) \\
t_9 &= a_3 + b_3
\end{align*}
\]
6.1 SSA and expressions

- Distinguish between different expressions $E$
- Versions for expressions
  - $a+b$ can be $E_1 = a_1 + b_1$, $E_2 = a_2 + b_1$, $E_3 = a_1 + b_2$, or $E_4 = a_3 + b_3$
- Versions allow us to reason about availability of expressions
- Care about expressions defined along a path
  - Need to deal with points where different paths $P_1$, $P_2$ merge
  - Define a merge function for expressions: $\Phi$
  - $\Phi : \text{uppercase } \phi$
Φ function

- Φ function is similar to the φ function for scalars
- Consider point P with paths $P_1, P_2, \ldots, P_m$ that end at P
- $E_k = \Phi (E_i, \ldots, E_j)$ at point P
  - m different paths – m arguments
- Returns the version of the expression defined along the path $P_q$ taken to reach P.
Expressions

- Given an expression $E$ and a point $P$ with a $\Phi$ function
- Consider the arguments to the $\Phi$ function.
- Is (at least some version of) expression $E$ evaluated along each path?
  - For variables, we assumed an initial assignment in START
  - This idea won’t work for expressions
- Need a way to indicate that no version of $E$ is evaluated along a path (i.e., $E$ is not evaluated ever)
  - $\perp$ (bottom): no version of $E$ is evaluated along the path
  - $E_k = \Phi(\perp, E_1, \perp, E_2)$
Expressions

- Given an expression $E$ and a point $P$ with a $\Phi$ function
- Consider the arguments to the $\Phi$ function.
- Is (at least some version of) expression $E$ evaluated along each path?
  - For variables, we assumed an initial assignment in START
  - This idea won’t work for expressions
- Need a way to indicate that no version of $E$ is evaluated along a path (i.e., $E$ is not evaluated ever)
  - $\bot$ (bottom): no version of $E$ is evaluated along the path
  - $E_k \notin (\bot, E_1, \bot, E_2)$ along this path, $E$ is not eval.
Example

\[ a_1 = \ldots \]
\[ b_1 = \ldots \]

\[ E_1 = a_1 + b_1 \]
\[ x_1 = E_1 \]

\[ E_2 = \Phi(\bot, E_1) \]
\[ x_3 = \phi(x_0, x_1) \]
\[ = E_2 \]
6.2 Outline

PRE with SSA in five (easy) steps

- Assumption: Program in SSA form for scalar variables

1. Insert $\Phi$ functions for expressions
   - (Introduce “temporary” to isolate expression
     \[ x = a_1 + b_1 \rightarrow E = a_1 + b_1 \]
     \[ x = E \]
   - Identify places where different versions merge

2. Identify/set version numbers for expressions
   - Expressions on LHS: set version number
   - Operands of functions: find correct version
(5 steps continued)

3. Identify places where it is legal to insert a copy of an expression
   - “legal” is defined later

4. Find places where the insertion on a copy is profitable
   - Only places that are legal can be considered
   - Expect removal of operations
   - Expect reduction in cycles
   - Metrics for profitability to be discussed

5. Exploit full redundancy of expressions
   - Transform program to reuse computed values
Outline

PRE with SSA in five (easy) steps

- Assumption: Program in SSA form for scalar variables

1. Insert $\Phi$ functions for expressions
2. Identify/set version numbers for expressions

Similar to SSA for scalars
6.3 Insertion of $\Phi$ functions

- Given the CFG of a program.
  - SSA format for scalars needs dominator tree, dominance frontier.
  - Keep for insertion of $\Phi$ functions
- Consider an expression $E = a + b$ in some block $B$.
- Must insert a $\Phi$ function *somewhere* if
  - $E$ is computed explicitly *or*
  - One of the operands of $E$ (i.e., $a$ or $b$) is changed in block $B$. 
Part 1: E computed in B

- Find dominance frontier $DF(B)$
  - Here \{B1, B2\}
- Insert a $\Phi$ function (unless already present)
Part 1: E computed in B

- Find dominance frontier DF(B)
  - Here \{B1, B2\}
- Insert a \( \Phi \) function (unless already present)
- This step deals with all explicit computations of E in some block
Part 2: Redefinition of operand(s)

- Consider \( E = a + b \) and \( a \) (or \( b \)) is defined in block \( B' \)
Part 2: Redefinition of operand(s)

- Consider \( E = a + b \) and \( a \) (or \( b \)) is defined in block \( B' \)
  - There must be a \( \phi \) function in nodes in \( DF(B') \)
  - Consider \( B \subseteq DF(B') \)

\[
\begin{align*}
a_1 &= \ldots \\
b_1 &= \ldots \\
E &= a_1 + b_1
\end{align*}
\]

\[
\begin{align*}
a_2 &= \ldots \\
E &= a_1 + b_1
\end{align*}
\]

\[
\begin{align*}
a_3 &= \phi(a_1, a_2) \\
    &= a_3 + b_1
\end{align*}
\]
Part 2: Redefinition of operand(s)

- Insert into $B \ (\in DF(B'))$ a $\Phi$ function for all expressions $E$ that contain $a$ as an operand.
6.5 Version numbers for expressions

- If E appears on the LHS: get a new version
- For the operands of a Φ function:
  - Identify correct version

- Recall: for functions (for scalars) we used a stack of versions
  - Array [variable] of stacks
- Use stack for operands of expression E to figure out which version is used
Consider \( \mathcal{E} \), the set of all expressions of interest in the program

- \( \text{stack}[\mathcal{E}] \): one stack for each expression

\( \mathcal{E} \subseteq \mathcal{E} \text{ stack}[\mathcal{E}] : \) stack of versions (integers)

Given an expression \( E = a + b \), after turning program into SSA for scalars, versions of \( a \) and \( b \) are known

- \( E = a_i + b_j \)

If we have processed \( E = a_i + b_j \) before (with these versions of \( a, b \)) then we use the version \( k \) of \( E \) on top of \( \text{stack}[\mathcal{E}] \)

- \( E_k = a_i + b_j \)
- Must be the current version of \( E \)
• So use version \( k \) of \( E \) if
  - \( \text{stack}(a).\text{top} = i \)
  - \( \text{stack}(b).\text{top} = j \)
  - and \( \text{stack}(E) = k \) with \( E_k = a_i + b_j \)

• If \( E = a_i + b_j \) has not been processed before, i.e.,
  - \( \text{stack}(a).\text{top} = i \)
  - \( \text{stack}(b).\text{top} = j \)
  - and \( \text{stack}(E) = k \) with \( E_k \neq a_i + b_j \) (i.e., operand versions changed but we did not find a new version for \( E \)) then
    - Get new version (say \( m \))
    - \( \text{stack}(E).\text{push}(m) \)
    - Use version \( m \) for \( E \)
\begin{align*}
    E_k & = a_i + b_j \\
    a_c & = b_j \quad \text{for } B \\
    \overline{B} & \text{ dom } B \\
    \overline{B} & \text{ dom } B' \\
    \text{Stack}[a] &= \langle \ldots, i \rangle \\
    \text{Stack}[b] &= \langle \ldots, j \rangle \\
    \text{Stack}[E] &= \langle \ldots, k \rangle \\
    k & = \text{top}(\text{Stack}[E]) \\
    i & = \text{top}(\text{Stack}[a]) \\
    j & = \text{top}(\text{Stack}[b]) \\
    \text{top}(\text{Stack}(E)) & = E_k \equiv (a_i + b_j)
\end{align*}
Example

\[ a_1 = \ldots \]
\[ b_1 = \ldots \]
\[ E_2 = a_1 + b_1 \]

\[ E = a_1 + b_1 \]

\[ a_2 = \ldots \]
\[ E = a_2 + b_1 \]