263-2810: Advanced Compiler Design

6.0 Partial redundancy elimination

Thomas R. Gross

Computer Science Department
ETH Zurich, Switzerland
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
3. **Identify places where is 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
6.4 Version numbers for expressions

- If E appears on the LHS: decide if a new version is needed or if the current version should be used
  - “current”: Version computed in a dominating node and no operand has been redefined
- For the operands of a Φ function:
  - Identify correct version
Recall: for functions (for scalars) we used a stack of versions

- Array $\mathcal{V}$ of stacks, with $\mathcal{V}$ the set of all variables

For $\mathcal{E}$, the set of all expressions of interest in the program

- stack[$\mathcal{E}$]: one stack for each expression
- $E \in \mathcal{E}$: stack[$E$] is a 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 stack[$E$]

- $E_k = a_i + b_j$
- $a_i, b_j$ must be current versions of $a, b$
- 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.,

- \( \text{stack}(a).top = i \)
- \( \text{stack}(b).top = j \)
- and \( \text{stack}(E) = k \) with \( E_k \neq a_i + b_j \) (i.e., operand versions changed but we do not find a new version for \( E \)) then
  - Get new version (say \( m \))
  - \( \text{stack}(E).push(m) \)
  - Use version \( m \) of \( E \): \( E_m \)
Example

\[
\begin{align*}
a_1 &= \ldots \hfill \text{B''} \\
b_1 &= \ldots \\
E_2 &= a_1 + b_1
\end{align*}
\]

\[
\begin{align*}
E &= a_1 + b_1 \\
\quad &\downarrow \\
\quad &\downarrow \\
a_2 &= \ldots \\
E &= a_2 + b_1 \hfill \text{B'}
\end{align*}
\]
Example

\[ E_2 = a_1 + b_1 \]

\[ a_1 = \ldots \]
\[ b_1 = \ldots \]
\[ \overline{E_2} = a_1 + b_1 \]

\[ a_2 = \ldots \]
\[ \overline{E_4} = a_2 + b_1 \]

\[ f = \Phi (E_2, E_4) \]

\[ a + 5 \]
Operands of $\Phi$ function

- **Visit nodes in depth-first order**
  - Use dominator tree
  - Like when handling $\phi$ functions

- **After processing a block (node), record expression version for $\Phi$ function(s) in successor blocks that are not dominated**
  - Make sure argument to $\Phi$ function reflects expression set along corresponding path
Example

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

\[ a_2 = \phi(a_4, a_1) \]

\[ E = a_2 + b_1 \]
\[ a_3 = \ldots \]

\[ a_4 = \phi(a_2, a_3) \]

\[ E = a_4 + b_1 \]

EXIT
Example

$E_2 = \frac{\perp}{a_4}$

$E_4 = a_2 + b_1$

$E_5 = a_4 + b_1$
6.5 Are copies legal at point P?

- Given a program in SSA format with $\phi$ and $\Phi$ functions
- Versions of scalars and expressions have been determined

- Given a basic block, P point at the start of block.
  A $\Phi$ function for E at point P with (one or more) $\perp$ operands indicates that
  - Value of expression E is undefined if control reaches P along path that corresponds to $\perp$ operand
  - Expression E is defined if control reaches P along a path that corresponds to version $E_k$ of E

- Predecessor basic block that corresponds to $\perp$ operand is a candidate to place a copy of E
Example

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

\[ E_1 = a_1 + b_1 \]

\[ E_2 = \Phi(\bot, E_1) \]

\[ = E_2 \]

EXIT
A copy of E can be inserted into B
  - Or any of its predecessors
  - (As long as operands of E are available)

Is inserting a copy acceptable?
- A copy of E can be inserted into B
  - Or any of its predecessors
  - (As long as operands of E are available)

- Is inserting a copy (into B) legal?

- “Gold standard”: insertion of a copy must not change the program (results)
Detour: “must not change the program”

- Given program $P$, transformed program $T(P) = P'$
- Transformation is legal if
  - $P$ computes the same result as $P'$
    - Same output
    - Returns no new errors
    - Throws no new exceptions
    - Termination behavior unchanged
- Allow that $P$ encounters an error earlier or later
- Allow that $P$ throws an exception earlier or later
- Note: values stored in memory by $P$ may differ from values stored by $P'$
  - May accept that values after error/exception differ
Example

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

\[ E_1 = a_1 + b_1 \]

\[ E_2 = \Phi (\bot, E_1) \]

\[ = E_2 \]

EXIT
Example

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

\[ E_1 = a_1 + b_1 \]

\[ E_2 = \Phi (\perp, E_1) \]

\[ a_2 = \ldots \]
\[ E_3 = a_2 + b_1 \]

EXIT
Legal copies

- Could we insert a copy of E into B?
Legal copies

- Could we insert a copy of E into B?

- We do not know anything about the effect of E.
  - Might throw an exception
  - Might raise an error (overflow, memory protection error, ...)

- A copy of E can be inserted into B if E is evaluated along all paths from B to EXIT
Example

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

\[ E_1 = a_1 + b_1 \]

\[ E_2 = \Phi(\bot, E_1) \]

\[ a_2 = \ldots \]
\[ E_3 = a_2 + b_1 \]
- Our model of legality: insert into B only if there is a copy of E on all paths from B to EXIT
  - Blocks with this property are called “downsafe”
  - Earlier papers use the phrase “E anticipated in B”

- Insertion legal: iff B is downsafe
Downsafety

- Check if there is a path from B to EXIT that does not include E
- If $E$ occurs only as the operand to a $\Phi$ function in block $B'$. Check that $E$ is used in $B'$ or $B'$ is downsafe.

We say a $\Phi$ function in block $B'$ is downsafe if $E$ appears on the RHS of a statement in $B'$ or $B'$ is downsafe.

- Can insert copies for $\Phi$ functions that are downsafe
**Downsafety**

- Need to check that along all paths from B to EXIT there is a *real* occurrence of E or a downsafe $\Phi$ function

- **Simple algorithm:**
  - Start at EXIT
  - Visit recursively all predecessor nodes, until all nodes have been visited
  - Mark a $\Phi$ function as downsafe iff a real occurrence or a downsafe $\Phi$ function appears on all paths to EXIT
  - After visiting all nodes: downsafe $\Phi$ functions are marked

- **Downsafety is a necessary condition**
Example

\[ E_2 = \Phi(E_1, \perp) \]

\[ E_3 = \Phi(E_2, \perp) \]

\[ = E_1 \]

\[ = E_2 \]

\[ = E_3 \]

EXIT
Example

\[ E_2 = \Phi(E_1, \bot) \]

\[ E_3 = \Phi(E_2, \bot) \]

\[ E_1 \]

\[ E_2 \]

\[ E_3 \]

EXIT

Note: A red cross indicates not a place for a copy.