

## CS738: Advanced Compiler Optimizations

### SSAPRE: SSA based Partial Redundancy Elimination

Amey Karkare

karkare@cse.iitk.ac.in

<http://www.cse.iitk.ac.in/~karkare/cs738>  
Department of CSE, IIT Kanpur



## PRE without SSA

- ▶ Based on well known DF analyses
  - ▶ Availability
  - ▶ Anticipability
  - ▶ Partial Availability
  - ▶ Partial Anticipability
- ▶ Identifies partially redundant computations, make them totally redundant by inserting new computations
- ▶ Remove totally redundant computations (CSE)

## PRE without SSA

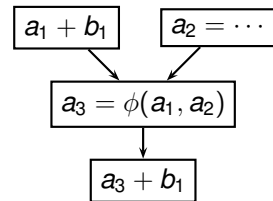
- ▶ Iterative data flow analysis
- ▶ Operates on control flow graph
- ▶ Computes global and local versions of data flow information

## SSAPRE

- ▶ Information flow along SSA edges
- ▶ No distinction between global and local information

## SSAPRE: Challenge

- ▶ SSA form defined for variables
- ▶ How to identify potentially redundant expressions
  - ▶ Expressions having different variable versions as operands



- ▶ Here  $a_1 + b_1$  is same as  $a_3 + b_1$ . Lexically different, but computationally identical

## SSAPRE: Key Idea

- ▶ Redundancy Class Variables (RCVs)
  - ▶ variable (say  $h$ ) to represent computation of an expression (say  $E$ )
- ▶ Computation of expression could represent either a *def* or a *use*
  - ▶ definition of  $E \Rightarrow$  store into  $h$
  - ▶ use of  $E \Rightarrow$  load from  $h$
- ▶ PRE on SSA form of RCVs ( $h$ ) to remove redundancies
- ▶ Final program will be in SSA form

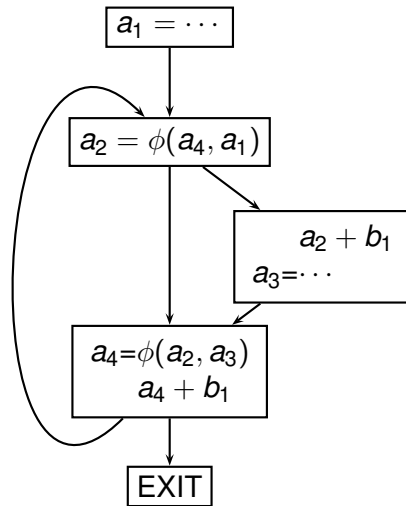
## SSAPRE: Preparations

- ▶ Split all the *critical edges* in the flow graph
  - ▶ Edge from a node with more than one successor to a node with more than one predecessor
  - ▶ WHY is this important?
- ▶ Single pass to identify identical expressions
  - ▶ Ignoring the version number of the operands
  - ▶  $a_3 + b_1$  and  $a_1 + b_1$  are identical

## SSAPRE Steps

- ▶ Six step algorithm
  1.  $\Phi$ -insertion
  2. Renaming
  3. Down-safety computation
  4. WillBeAvail computation
  5. Finalization
  6. Code Motion

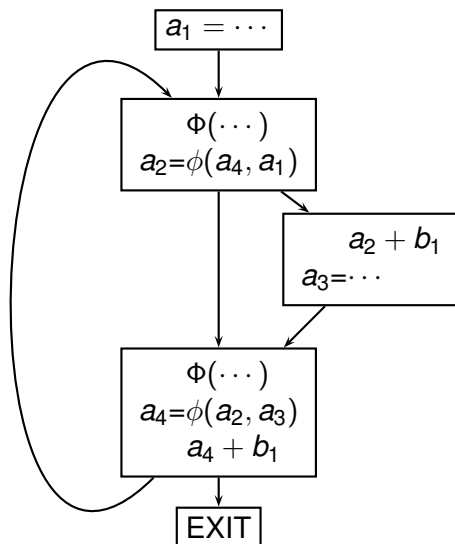
## Running Example



## $\Phi$ -insertion

- ▶  $\Phi$  for an expression  $E$  is required where two potentially different values of an expression merge
- ▶ At iterated dominance frontiers of occurrences of  $E$
- ▶ At each block having a  $\phi$  for some argument of  $E$ 
  - ▶ Potential change in the expression's value

## $\Phi$ -insertion



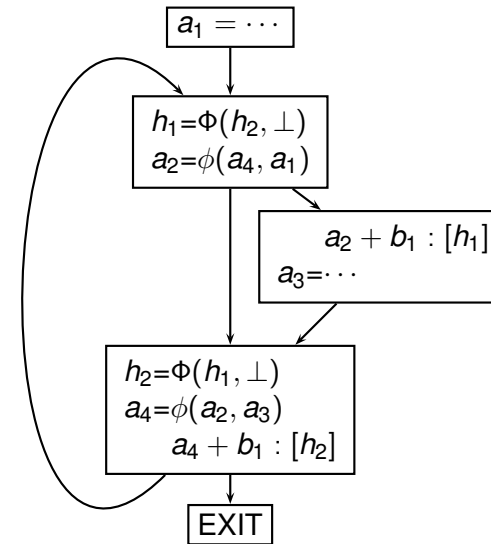
## Rename

- ▶ Similar to SSA variable renaming
- ▶ Stack of every expression is maintained
- ▶ Three kinds of occurrences of  $E$ 
  - ▶ Real occurrences (present in original program)
  - ▶ Results of  $\Phi$  operators inserted
  - ▶ Operands of inserted  $\Phi$
- ▶ After renaming
  - ▶ Identical SSA instances of  $h$  represent identical values of  $E$
  - ▶ A control flow path with two different instances of  $h$  has to cross either an assignment to an operand of  $E$  or a  $\Phi$  of  $h$

## Rename Algorithm

- ▶ Runs with variable renaming
- ▶ When an  $E$  is encountered
  - ▶ if  $E$  is result of  $\Phi$ , assign a new version to  $h$  and push it on  $E$  stack
  - ▶ if  $E$  is the real occurrence
    - ▶ for each operand, compare the version of operand with the top of the rename stack for operand
    - ▶ If all match,  $h$  gets same version as the top of  $E$  stack
    - ▶ If any mismatch, assign a new version to  $h$  and push it on  $E$  stack
- ▶ if  $E$  is operand of  $\Phi$ , in the corresponding predecessor block
  - ▶ for each operand of  $E$ , compare the version of operand with the top of the rename stack for operand
  - ▶ If all match,  $h$  gets same version as the top of  $E$  stack
  - ▶ If any mismatch, replace  $E$  by  $\perp$  in the operand push it on  $E$  stack (WHY?)

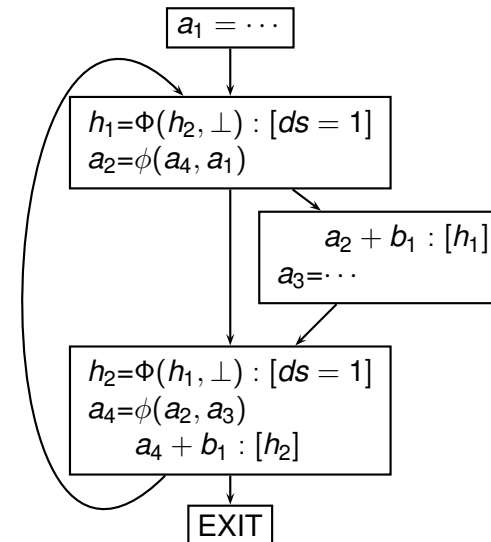
## Rename



## Down-safety

- ▶ Down-safety is same as very-busy (anticipability) property of expressions
  - ▶ Do not want to introduce new computation of  $E$
- ▶ We only need to compute down-safety for inserted  $\Phi$ -operators
- ▶ A  $\Phi$  computation is **NOT** down-safe if
- ▶ there is a path to EXIT from  $\Phi$  along which the result of  $\Phi$  is
  - ▶ either not used
  - ▶ used only as an operand of another  $\Phi$  that itself is **NOT** down-safe
- ▶ *HasRealUse*: Real occurrence of an expression

## Down-safety (ds = ...)



## WillBeAvail

- ▶ The set of  $\Phi$ s where the expression must be available in any computationally optimal placement
- ▶ Computation of *two forward* properties:
  - ▶ *CanBeAvail*:  $\Phi$ s for which  $E$  is either available or anticipable or both
  - ▶ *Later*:  $\Phi$ s beyond which insertion can not be postponed without introducing new redundancy

$$WillBeAvail = CanBeAvail \wedge \neg Later$$

## CanBeAvail

- ▶ Initialized to *true* for all  $\Phi$ s
- ▶ Boundary  $\Phi$ s:
  - ▶ Not Down-safe, and
  - ▶ At least one argument is  $\perp$
- ▶ Set *false* for boundary  $\Phi$ s
- ▶ Propagate *false* value along the chain of def-use to other  $\Phi$ s
  - ▶ exclude edges along which *HasRealUse* is *true*

## Later

- ▶ Determines latest (final) insertion points
- ▶ Initialize *Later* to *true* wherever *CanBeAvail* is *true*, otherwise *false*
- ▶ Assign *false* for  $\Phi$ s with at least one operand with *HasRealUse* flag *true*
- ▶ Propagate *false* value forward to other  $\Phi$ s
- ▶ *Later*  $\Rightarrow$   $\Phi$ s that are *CanBeAvail*, but do not reach any real occurrence of  $E$

## Insertion Points

- ▶ Insertions are done for  $\Phi$  operands
- ▶ Along the corresponding predecessor edges
- ▶ Insertion done along  $j^{th}$  predecessor of  $\Phi$  if *Insert* is *true*, i.e.
  - ▶ *WillBeAvail*( $\Phi$ ) == *true*; AND
  - ▶ *Arg<sub>i</sub>* is  $\perp$ ; OR
    - ▶ (*HasRealUse*(*Arg<sub>i</sub>*) == *false*), AND
    - ▶ *Arg<sub>i</sub>* is defined by  $\Phi'$  with *WillBeAvail*( $\Phi'$ ) == *false*

## Finalize

- ▶ Transforms the program with RCVs into a valid SSA form
- ▶ For every real occurrence of  $E$ , decide whether it is a *def* or a *use*
- ▶ For every  $\Phi$  with *WillBeAvail* being *true*, insert  $E$  along incoming edges with *Insert* being *true*
- ▶ For each  $\Phi$  for  $E$ 
  - ▶ If *WillBeAvail* is *true*, it is replaced by SSA temporary with appropriate version ( $h_x$ )
  - ▶ If *WillBeAvail* is *false*, it is not part of SSA form, and is removed

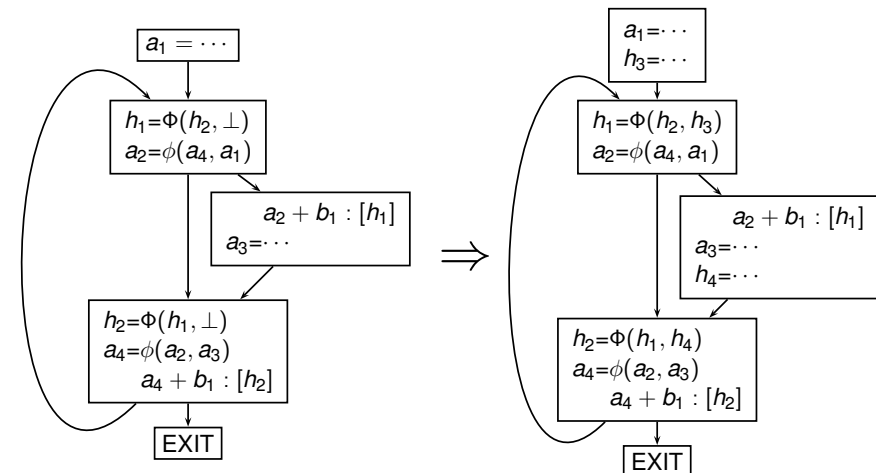
## Finalize: AvailDef

- ▶ AvailDef: Table to mark def of expression occurrences
- ▶ Computed for each class (say  $h_x$ ) of  $E$
- ▶ Preorder traversal of dominator tree

## AvailDef Computation

- ▶ Initialize:  $\text{AvailDef}[x] = \perp \forall x$  (all classes of all expressions)
- ▶ During course of traversal, process occurrence  $x$  of  $E$ 
  - ▶  $\Phi$  occurrence:
    - ▶ If *WillBeAvail* is *false*, ignore.
    - ▶ Otherwise  $\text{AvailDef}[x] = \text{this } \Phi$  (we must be visiting  $x$  for first time) – WHY?
  - ▶ Real occurrence:
    - ▶ If  $\text{AvailDef}[x]$  is  $\perp$ , mark this occurrence as def
    - ▶ Else, if  $\text{AvailDef}[x]$  does not dominate this occurrence, mark this occurrence as def
    - ▶ Else, mark this occurrence as use of  $\text{AvailDef}[x]$
  - ▶  $\Phi$  operand (processed in predecessor block  $P$ )
    - ▶ If *WillBeAvail* of  $\Phi$  is *false*, ignore.
    - ▶ Else, if *Insert* is true for the operand, insert computation of  $E$  in block  $P$ , set it as a def, mark this occurrence as use of inserted.
    - ▶ Else (*Insert* is false), mark this occurrence as use of  $\text{AvailDef}[x]$

## Finalize



## Code Motion

- ▶ For real *def* occurrence of  $E$ , compute  $E$  in a new version of temporary  $t$
- ▶ For real *use* occurrence of  $E$ , replace  $E$  by current version of  $t$
- ▶ For inserted occurrence of  $E$ , compute  $E$  in a new version of temporary  $t$
- ▶ For a  $\Phi$  occurrence, insert appropriate  $\phi$  for  $t$

## Code Motion

