Lecture 5

Partial Redundancy Elimination

I. Redundancy Optimizations
   - Global common subexpression elimination
   - Loop invariant code motion
   - Partial redundancy elimination

II. Lazy Code Motion Algorithm
   - Mathematical concept: a cut set
   - Basic technique (anticipation)
   - 3 more passes to refine algorithm

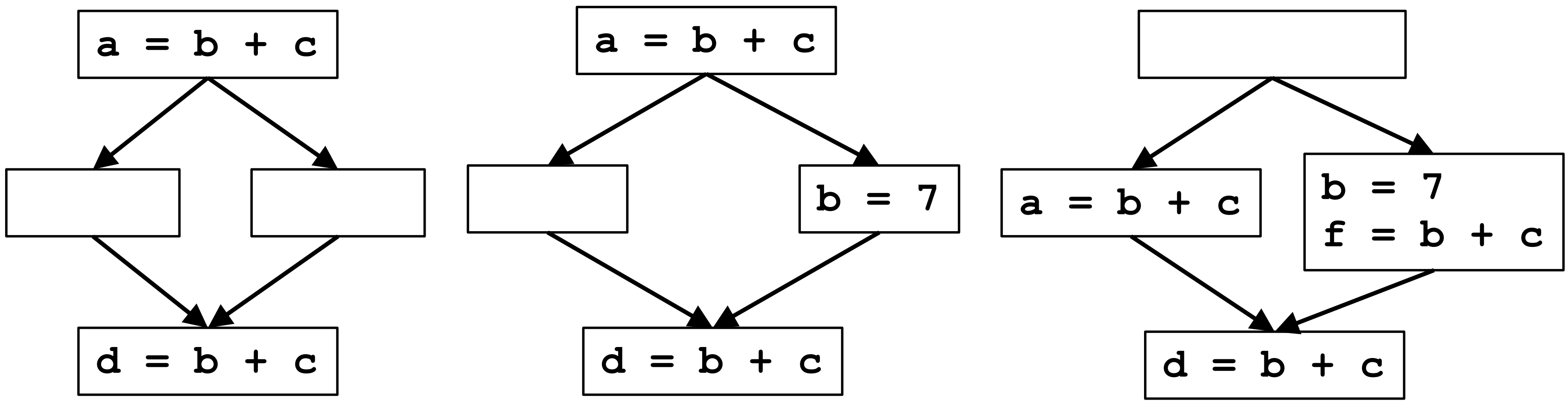
Reading: Chapter 9.5

Overview

- **Redundancy optimizations**
  - Global common subexpression elimination
  - Loop invariant code motion
  - Partial redundancy elimination
- **Unified as one algorithm! (Originally, a 1-bidirectional data flow analysis)**
- **Lazy code motion algorithm**
  - formulated as 4 separate uni-directional passes
    - backward, forward, forward, backward
- **Shows off the power and elegance of data flow**
  - Sees program as a graph
    Uses the mathematical concept of a cut set, used later in instruction scheduling
- **Plan**
  - Simple examples to build up your intuition
  - Introduce cut sets
  - Key: understand what the algorithm does without simulation
  - Details of the algorithm
- **Simple but hard**: Please work out examples after class immediately

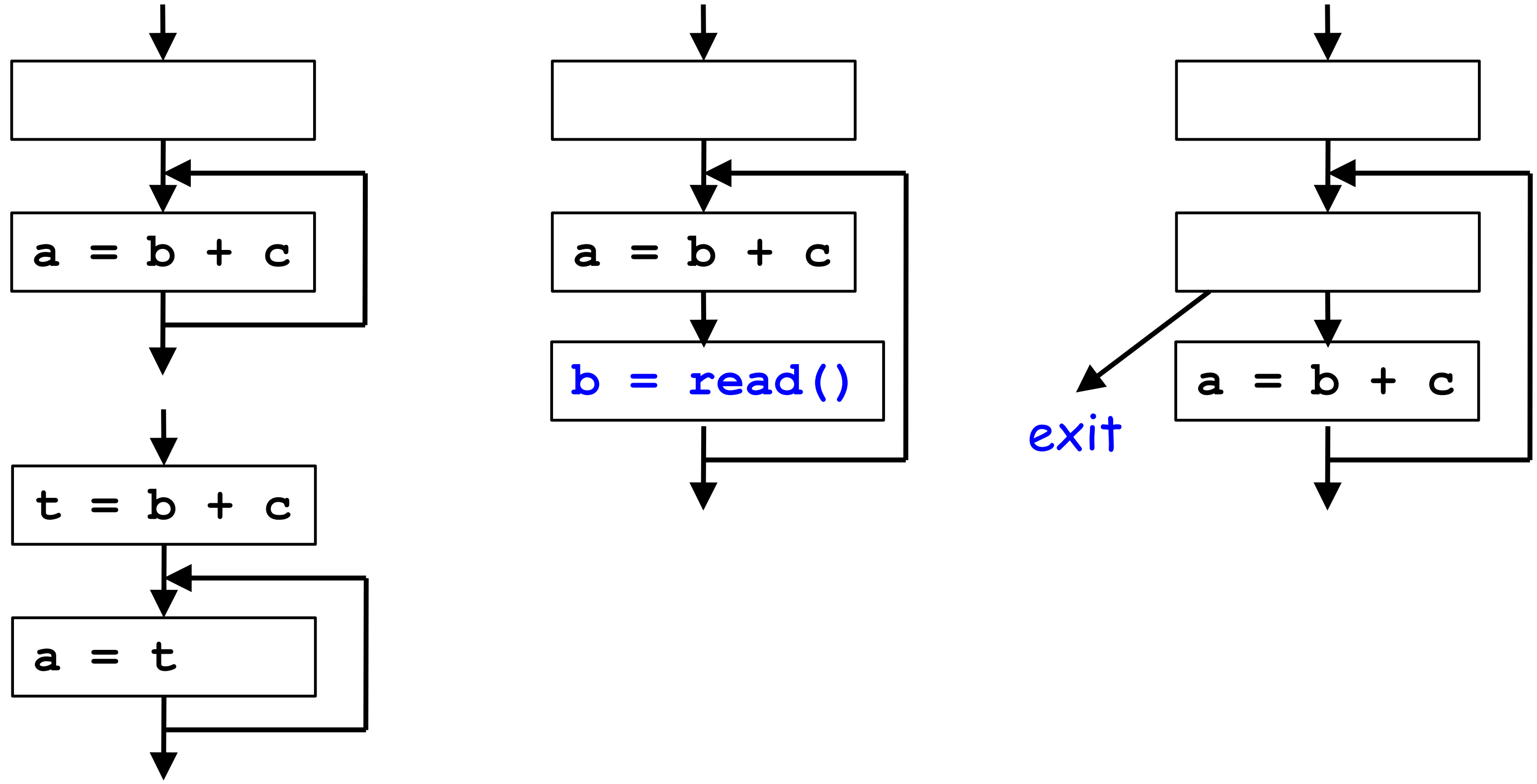
I. Common Subexpression Elimination

Build up intuition about redundancy elimination with examples of familiar concepts



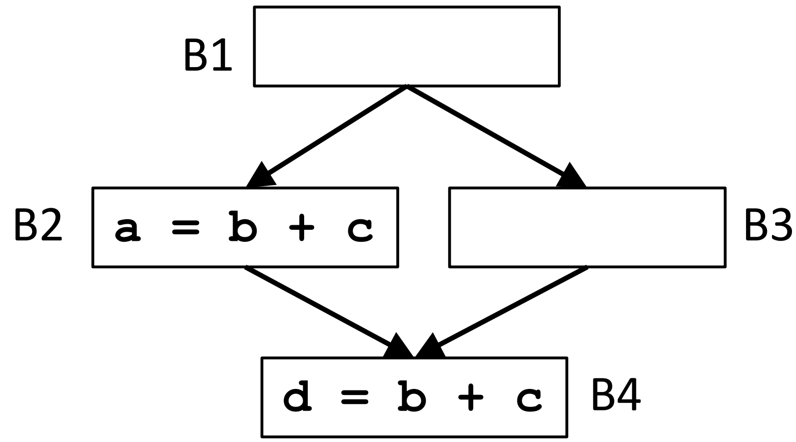
- **A common expression may have different values on different paths!**
- **On every path reaching p,**
  - expression b+c has been computed
  - b, c not overwritten after the expression

Loop Invariant Code Motion



- Given an expression (b+c) inside a loop,
  - does the value of b+c change inside the loop?
  - is the code executed at least once?

Partial Redundancy



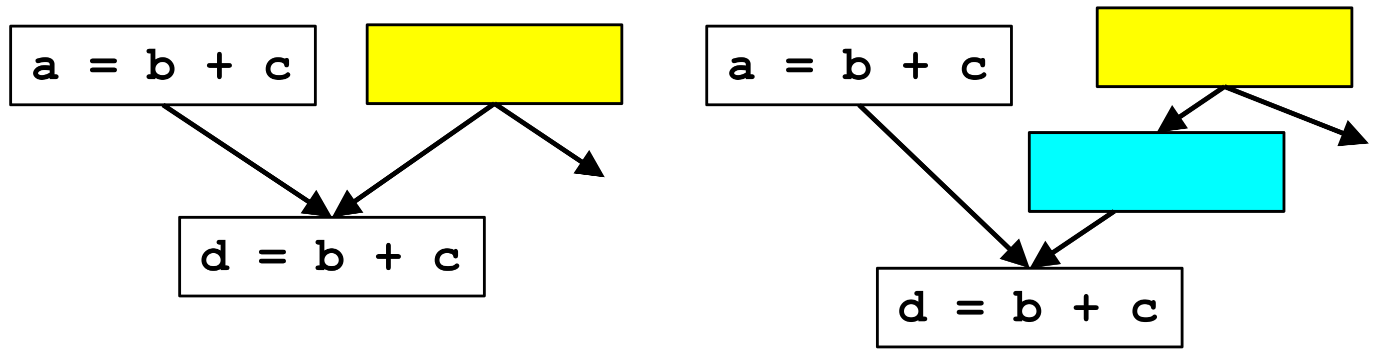
- Can we place calculations of b+c
  such that no path re-executes the same expression

- Partial Redundancy Elimination (PRE)
  - subsumes:
    - global common subexpression (full redundancy)
    - loop invariant code motion (partial redundancy for loops)

Unifying theory: More powerful, elegant → but less direct.

II. Preparing the Flow Graph

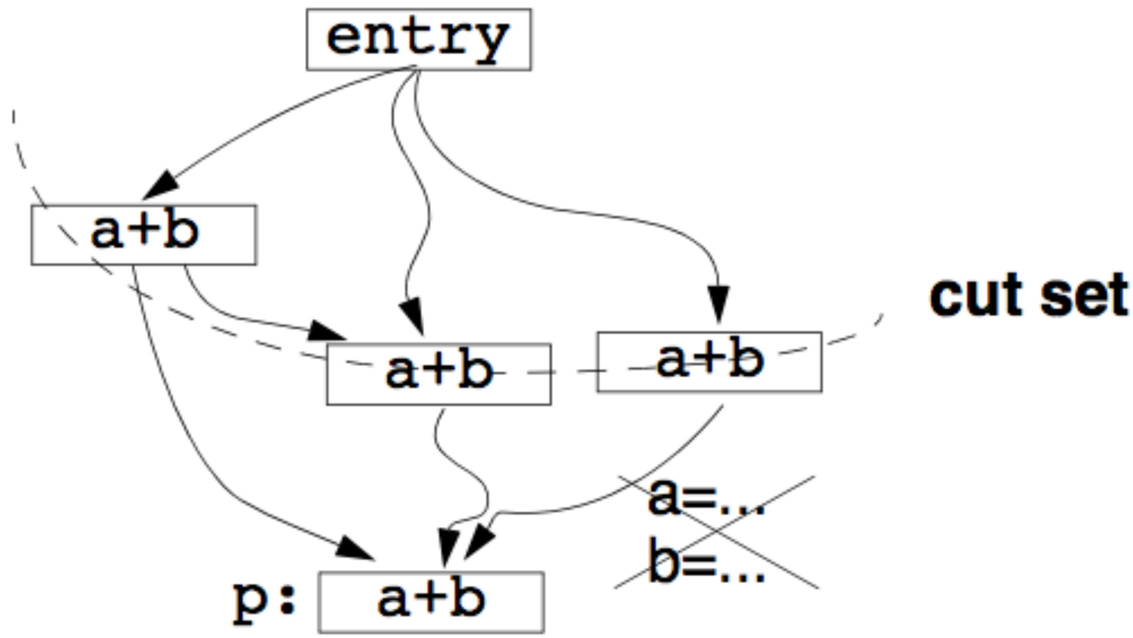
- A simple flow graph modification improves the result



  - Can replace the bi-directional data flow
    with several unidirectional data flows → much easier
- **Definition: Critical edges**
  - source basic block has multiple successors
  - destination basic block has multiple predecessors
- **Modify the flow graph: (treat every statement as a basic block)**
  - To keep algorithm simple:
    restrict placement of instructions to the beginning of a basic block
  - Add a basic block for every edge that leads to a basic block with multiple predecessors (not just on critical edges)

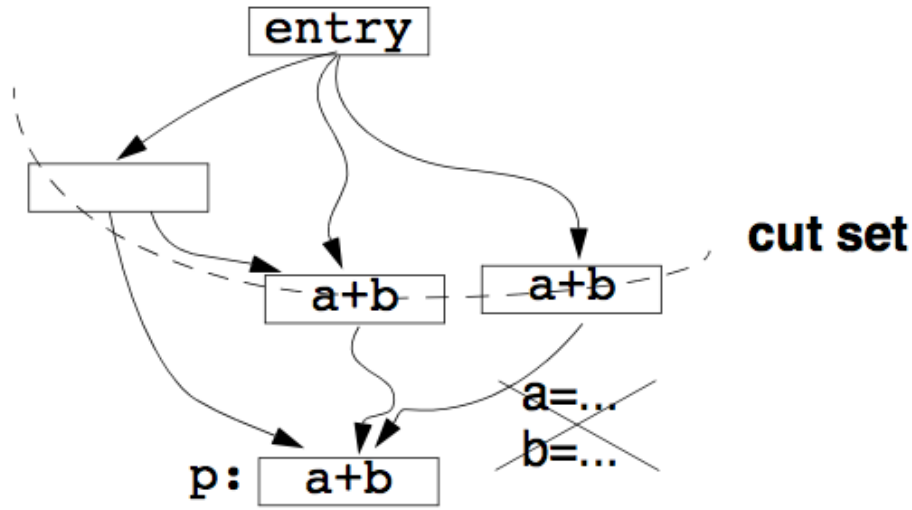
Full Redundancy: A Cut Set in a Graph

Key mathematical concept



- **Full redundancy at p: expression a+b redundant on all paths**
  - a cut set: nodes that separate entry from p
  - a cut set contains calculation of a+b
  - a, b, not redefined

Partial Redundancy: Completing a Cut Set



- **Partial redundancy at p: redundant on some but not all paths**
  - Add operations to create a cut set containing a+b
- **Constraint on placement: no wasted operation**
  - a+b is "anticipated" at B if its value computed at B will be used along ALL subsequent paths
  - a, b not redefined before use, no branches that lead to exit without using a+b
- **Range where a+b is anticipated → Choice**
- ***Greedy***: Place operations at the earliest frontier to maximize reuse

Pass 1: Anticipated Expressions

This pass does most of the heavy lifting in eliminating redundancy

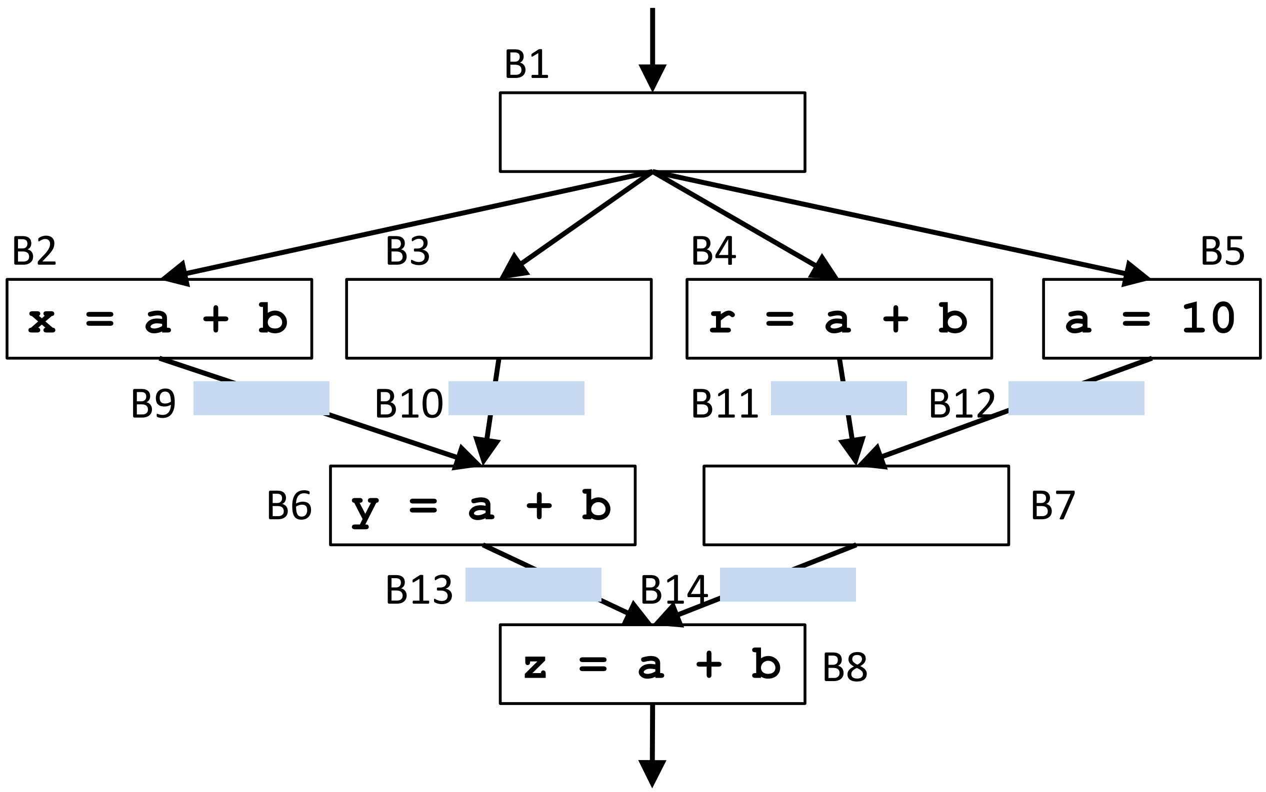
- **Backward pass: Anticipated expressions**
  Anticipated[b].in: Set of expressions anticipated at the entry of b
  - An expression is anticipated if its value computed at point p will be used along ALL subsequent paths

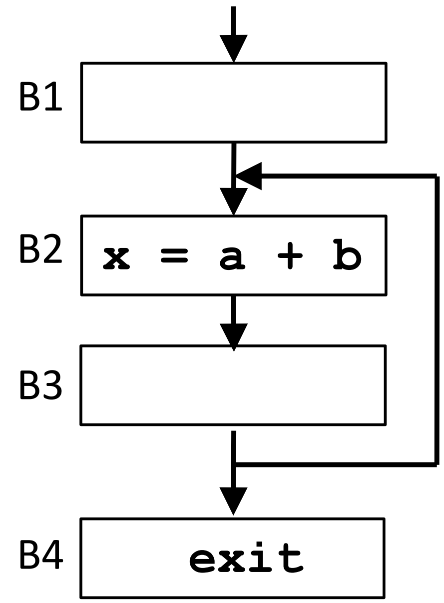
	Anticipated Expressions
Domain	Sets of expressions
Direction	backward
Transfer Function	$f_b(x) = EUse_b \cup (x - EKill_b)$ EUse: used exp, EKill: exp killed
$\wedge$	$\cap$
Boundary	in[exit] = $\emptyset$
Initialization	in[b] = {all expressions}

- **First approximation:**
  - place operations at the earliest frontier of anticipation (boundary from "not anticipated" to "anticipated")

Examples (1)

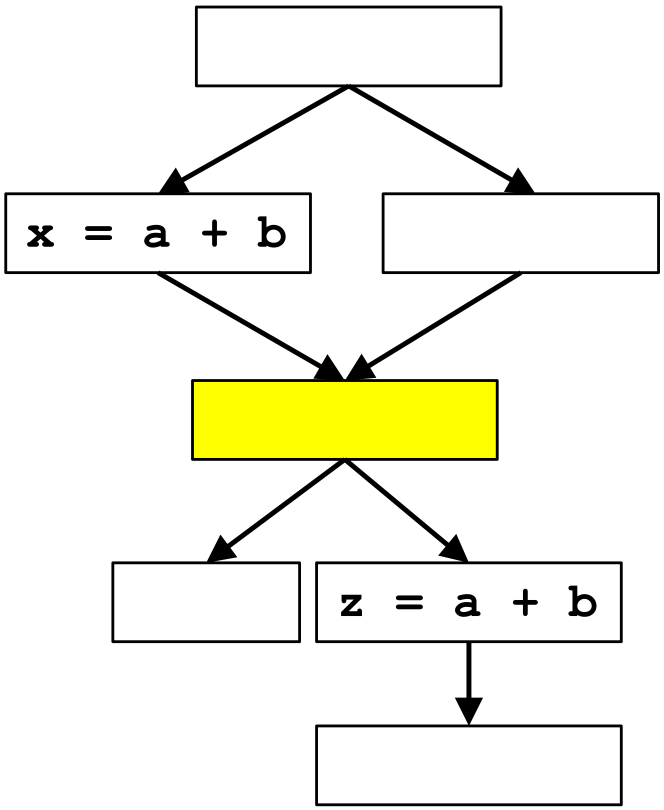
See the algorithm in action





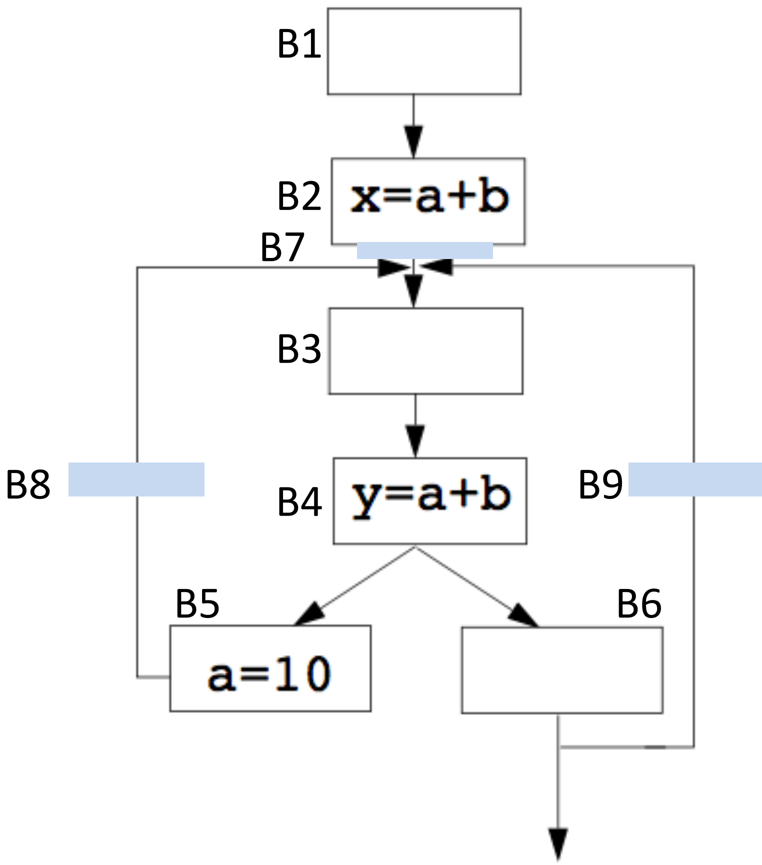
Examples (2)

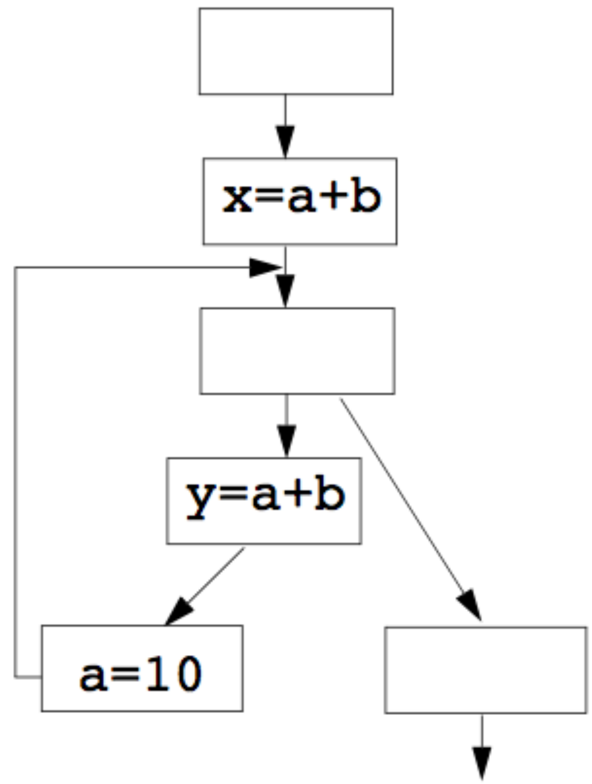
Can Partial Redundancy Elimination eliminate all partial redundancy?



Examples (3)

Do you know how the algorithm works without simulating it?

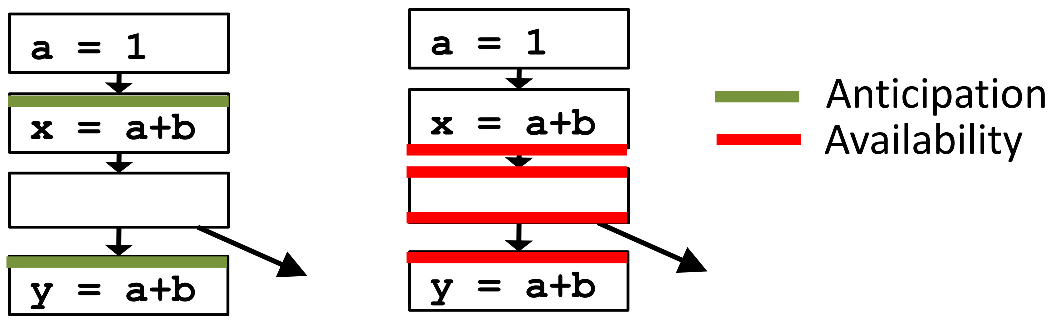




Pass 2: Place As Early As Possible

There is still some redundancy left!

- **First approximation: frontier between "not anticipated" & "anticipated"**
- **Complication: Anticipation may oscillate**



- **An anticipation frontier may cover a subsequent frontier.**
- **Once an expression has been anticipated, it is "available" to subsequent frontiers → no need to re-evaluate.**
- e will be **available at p** if
  e has been "anticipated but not subsequently killed" on all paths reaching p
- **Place expression at the earliest point anticipated and not already available**
  - earliest(b) = anticipated[b].in - available[b].in

Available Expressions

- e will be **available at p** if
  e has been "anticipated but not subsequently killed" on all paths reaching p

	Available Expressions
Domain	Sets of expressions
Direction	forward
Transfer Function	$f_b(x) = (\text{Anticipated}[b].\text{in} \cup x) - \text{EKill}_b$
$\wedge$	$\cap$
Boundary	out[entry] = $\emptyset$
Initialization	out[b] = {all expressions}

Early Placement

- **earliest(b)**
  - set of expressions added to block b under early placement
- **Place expression at the**
  earliest point anticipated and not already available
  - earliest(b) = anticipated[b].in - available[b].in
- **Algorithm**
  - For all basic block b,

    if x+y ∈ earliest[b]

    at beginning of b:

    let t be the unique variable representing x+y

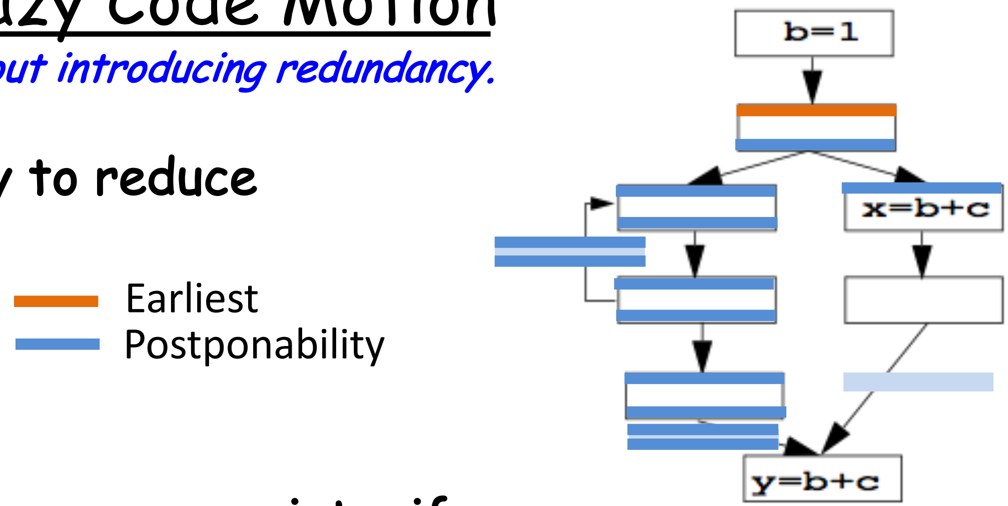
    add t = x+y,

    replace every original x+y in the program by t

Pass 3: Lazy Code Motion

Let's be lazy without introducing redundancy.

Delay without creating redundancy to reduce register pressure

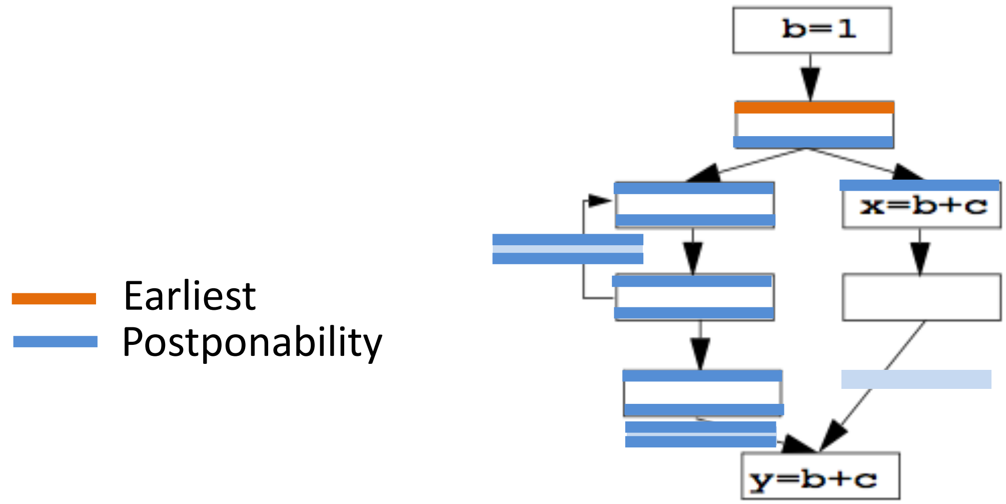


An expression e is postponable at a program point p if

- all paths leading to p
  have seen the earliest placement of e but not a subsequent use

	Postponable Expressions
Domain	Sets of expressions
Direction	forward
Transfer Function	$f_b(x) = (\text{earliest}[b] \cup x) - \text{EUse}_b$
$\wedge$	$\cap$
Boundary	out[entry] = $\emptyset$
Initialization	out[b] = {all expressions}

Latest: Latest Frontier of "postponable" Cut Set

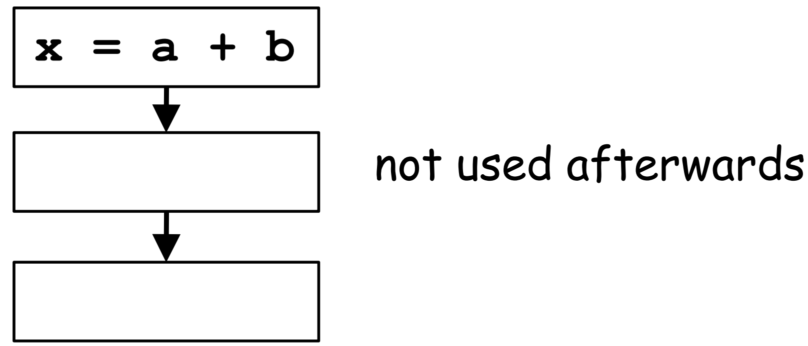


- latest[b] = (earliest[b] $\cup$ postponable.in[b]) $\cap$

  (EUse$_b$ $\cup$ $\neg$($\bigcap_{s \in succ[b]}$(earliest[s] $\cup$ postponable.in[s])))
  - OK to place expression: earliest or postponable.in
  - Need to place at b if either
    - used in b, or
    - not OK to place in one of its successors
- Note: If postponable.out[b] and $\neg$ postponable.in[s], b is empty and there is only one successor s

Pass 4: Cleaning Up

Finally… this is easy, it is like liveness



- **Eliminate temporary variable assignments unused beyond the current block**
- **Compute: Used.out[b]: sets of used (live) expressions at exit of b.**

	Used Expressions
Domain	Sets of expressions
Direction	backward
Transfer Function	$f_b(x) = (EUse[b] \cup x) - latest[b]$
$\wedge$	$\cup$
Boundary	in[exit] = ∅
Initialization	in[b] = ∅

Code Transformation

Original version: For each basic block b,

if x+y $\in$ earliest[b]

at beginning of b:

let t be the unique variable representing x+y
add t = x+y,

replace every original x+y in the program by t

New version: For each basic block b,

if (x+y) $\in$ (latest[b] $\cap \neg$ used.out[b]) { }

else

if x+y $\in$ latest[b]

at beginning of b:

let t be the unique variable representing x+y
add t = x+y,

replace every original x+y in the program by t

4 Passes for Partial Redundancy Elimination

- *Heavy lifting*: **Cannot introduce operations not executed originally**
  - Pass 1 (backward): Anticipation: range of code motion
  - Placing operations at the frontier of anticipation gets most of the redundancy
- ***Squeezing the last drop of redundancy*:**
  An anticipation frontier may cover a subsequent frontier
  - Pass 2 (forward): Availability
  - Earliest: anticipated, but not yet available
- ***Push the cut set out -- as late as possible***
  To minimize register lifetimes
  - Pass 3 (forward): Postponability: move it down provided it does not create redundancy
  - Latest: where it is used or the frontier of postponability
- ***Cleaning up***
  - Pass 4: **Remove temporary assignment**

Remarks

- **Powerful algorithm**
  - Finds many forms of redundancy in one unified framework
- **Illustrates the power of data flow**
  - Multiple data flow problems