# Lecture 12:

# Lazy Code Motion

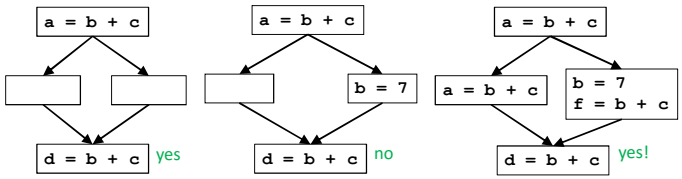
I. Forms of redundancy (quick review)
- global common subexpression elimination
- loop invariant code motion
- partial redundancy

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

[ALSU 9.5.3-9.5.6]



## I. Common Subexpression Elimination

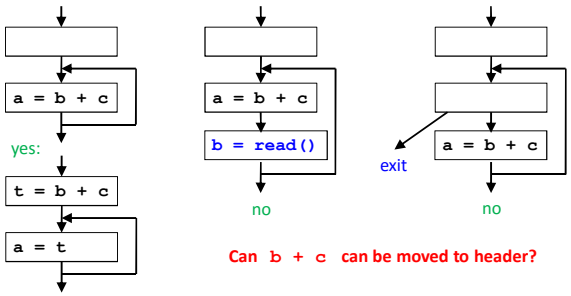



**Which `b + c` in bottom row is a common subexpression?**

- **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

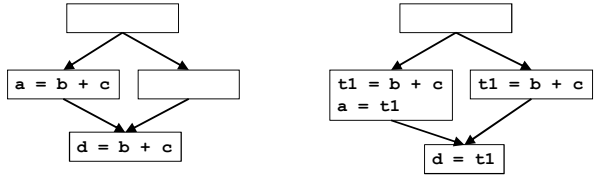



**Can `b + c` can be moved to header?**

- 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?

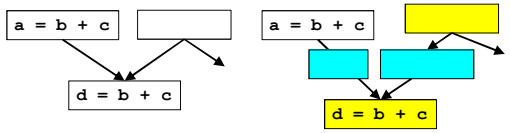
- Partial Redundancy Elimination (PRE)
  - subsumes:
    - global common subexpression (full redundancy)
    - loop invariant code motion (partial redundancy for loops)

## II. Lazy Code Motion

- **Key observations:**
  - A bi-directional data flow problem ("Placement Possible") can be replaced with 4 separate unidirectional data flow problems
    - backward, forward, forward, backward
    - makes it much easier to implement
  - Attempts to minimize register lifetimes (while eliminating redundancy)
- **Big picture**:
  - First calculates the "earliest" set of blocks for insertion
    - this maximizes redundancy elimination
    - but may also result in long register lifetimes
  - Then it calculates the "latest" set of blocks for insertion
    - achieve the same amount of redundancy elimination as "earliest"
    - but hopefully reduces register lifetimes



## Preparing the Flow Graph

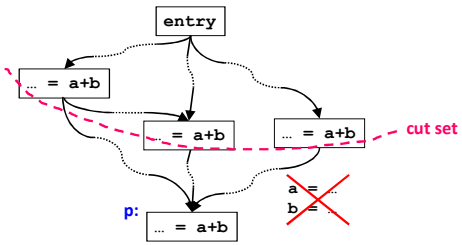



- **Definition: Critical edges**
  - source basic block has multiple successors
  - destination basic block has multiple predecessors
- **Modify the flow graph: (treat every statement as a basic block)**
  - To keep algorithm simple: restrict placement of instructions to the beginning of a basic block
  - 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

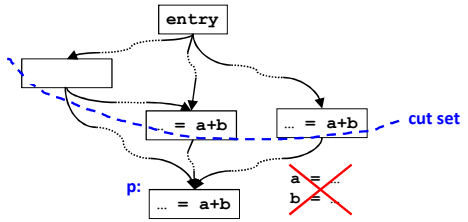
*Key mathematical concept*




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



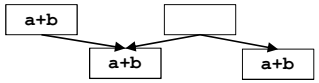
## Partial Redundancy: Completing a Cut Set




- **Partial redundancy at p: redundant on some but not all paths**
  - Add operations to create a cut set containing a+b
  - Note: Moving operations up can eliminate redundancy
- **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, no branches that lead to exit without use
- **Range where a+b is anticipated → Choice**

## Review: Finding Partially Available Expressions using PAVIN

- **Forward flow problem**
  - **Lattice = { 0, 1 }, meet is union (∪), Top = 0, entry = 0, init = 0**
    - PAVOUT[b] = (PAVIN[b] – KILL[b]) ∪ AVLOC[b]
    - PAVIN[b] = $\begin{cases} 0 & b = entry \\ \bigcup_{p \in preds(b)} PAVOUT[p] & otherwise \end{cases}$
- **For a block,**
  - Expression is **locally available (AVLOC)** if computed & downwards exposed.
  - Expression is killed (**KILL**) if any assignments to operands.
- PAVIN heuristic fails to find opportunity to replace bottom a+b's with top right a+b






## 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 | $f_b(x) = EUse_b \cup (x - EKill_b)$<br>EUse: used exp, EKill: exp killed |
| ∧ | ∩ |
| Boundary | in[exit] = ∅ |
| Initialization | in[b] = {all expressions} |

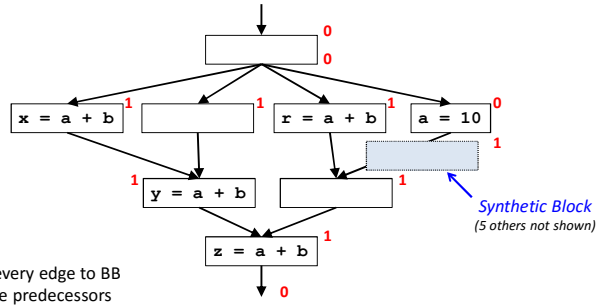
- **First approximation:**
  - place operations at the frontier of anticipation (boundary between not anticipated and anticipated)



## Example 1

*See the algorithm in action*

**IN[i] = EUse[i] ∪ (OUT[i] - EKill[i])**
**Meet = ∩**




Add BB for every edge to BB with multiple predecessors

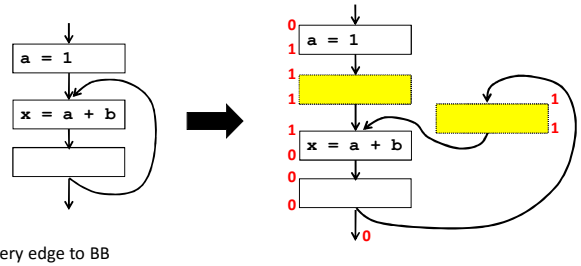
- What is the result if we insert at the frontier of anticipation?



## Example 2

(Loop Invariant Code)

**IN[i] = EUse[i] ∪ (OUT[i] - EKill[i])**
**Meet = ∩**

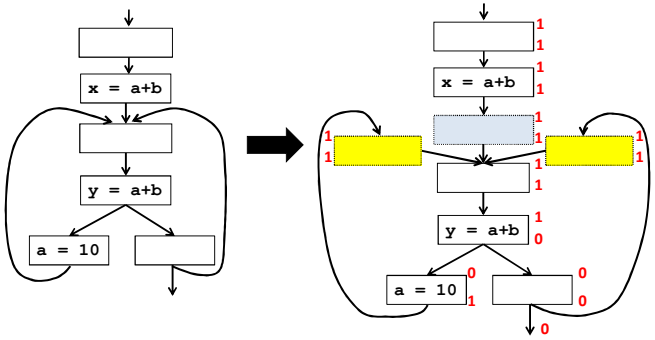



Add BB for every edge to BB with multiple predecessors

- Was inserting at the frontier of anticipation the right thing to do in this case?
  - doesn't eliminate redundancy within loop (why not?)

## Example 3
(More Complex Loop)

**IN[i] = EUse[i] ∪ (OUT[i] - EKill[i])**
**Meet = ∩**

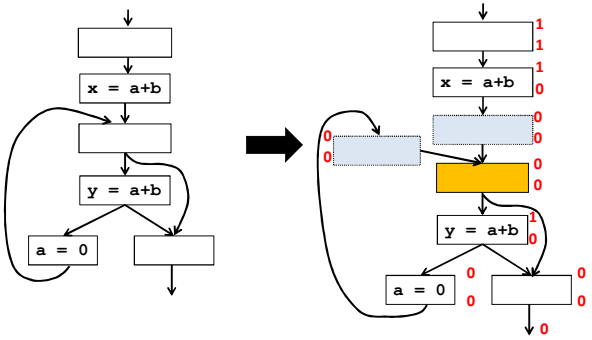



- Where would we ideally like to insert "**a+b**" in this case? only in added block on left
- What happens if we insert at the frontier of anticipation? Insert in both yellow blocks



## Example 4
(Variation on Previous Loop)

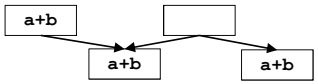
**IN[i] = EUse[i] ∪ (OUT[i] - EKill[i])**
**Meet = ∩**




- Is there any opportunity to eliminate redundancy here?
  no: unsafe to insert in left added block (a+b not anticipated there)
  (e.g. "a+b" could be "b/a" & orange block could be "if a > 0")



## Example 5
Bad case for PAVIN



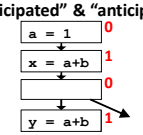

Works for Anticipated Expression



## Pass 2: Place As Early As Possible

*There is still some redundancy left!*

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




- Pretend we calculate expression e whenever it is anticipated
- e will be **available at p** if e has been "anticipated but not subsequently killed" on all paths reaching p

| | (will be) Available Expressions |
|---|---|
| Domain | Sets of expressions |
| Direction | forward |
| Transfer Function | $f_b(x) = (\text{Anticipated[b].in} \cup x) - \text{EKill}_b$ |
| ∧ | ∩ |
| Boundary | out[entry] = ∅ |
| Initialization | out[b] = {all expressions} |

## Early Placement

- **earliest(b)**
  - set of expressions added to block b under early placement
  - calculated from results of first 2 passes
- **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:
      create a new variable t
      t = x+y,
      replace every original x+y by t

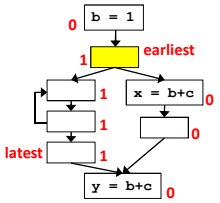
**Result:**
- Maximized redundancy elimination
- Placed as early as possible
- But: register lifetimes?



## Pass 3: Lazy Code Motion

*Let's be lazy without introducing redundancy*

- **Delay creating redundancy to reduce register pressure**

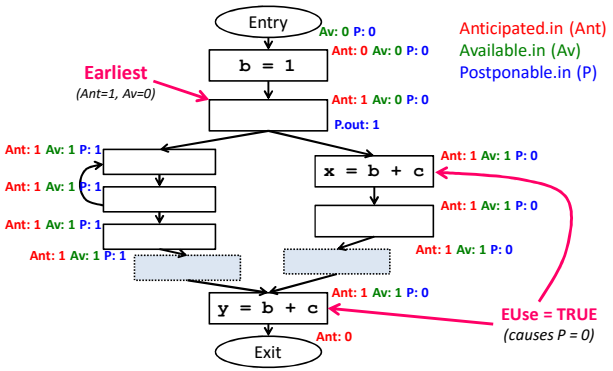



- **An expression e is postponable at a program point p if**
  - all paths leading to p have seen earliest placement of e but not a subsequent use

| | Postponable Expressions |
|---|---|
| Domain | Sets of expressions |
| Direction | forward |
| Transfer Function | $f_b(x) = (earliest[b] \cup x) - EUse_b$ |
| ∧ | ∩ |
| Boundary | out[entry] = ∅ |
| Initialization | out[b] = {all expressions} |



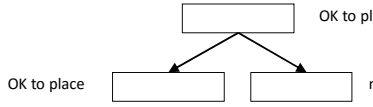
## Example Illustrating "Postponable"



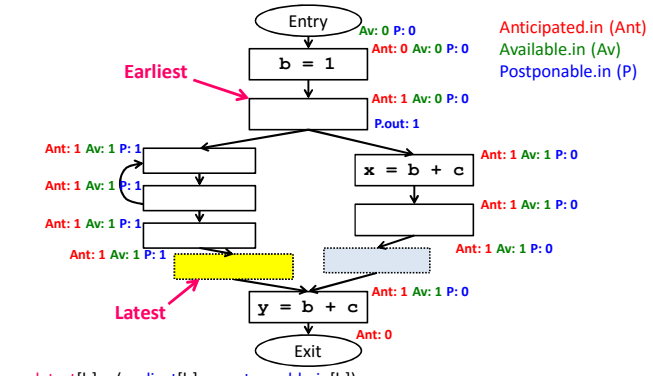



## Latest: frontier at the end of "postponable" cut set

- latest[b] = (earliest[b] ∪ postponable.in[b]) ∩
  (EUse$_b$ ∪ ¬(∩$_{s \in succ[b]}$(earliest[s] ∪ postponable.in[s])))
  - OK to place expression: earliest or postponable
  - Need to place at b if either
    - used in b, or
    - not OK to place in one of its successors
- Works because of pre-processing step (an empty block was introduced to an edge if the destination has multiple predecessors)
  - if b has a successor that cannot accept postponement, b has only one successor
  - The following does not exist:



## Example Illustrating "Latest"

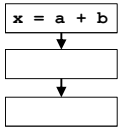



- latest[b] = (earliest[b] ∪ postponable.in[b]) ∩
  (EUse$_b$ ∪ ¬(∩$_{s \in succ[b]}$(earliest[s] ∪ postponable.in[s])))



## Pass 4: Cleaning Up

*Finally… this is easy, it is like liveness (for expressions)*




not used afterwards

- **Eliminate temporary variable assignments unused beyond 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] ∪ x) - latest[b] |
| ∧ | ∪ |
| Boundary | in[exit] = ∅ |
| Initialization | in[b] = ∅ |



## Code Transformation

- For all basic blocks b,

  if (x+y) ∈ (latest[b] ∩ used.out[b])

  at beginning of b:

  add new t = x+y

  replace every original x+y by t



## 4 Passes for Partial Redundancy Elimination

1. ***Safety*: 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
2. ***Squeezing the last drop of redundancy*:**
   **An anticipation frontier may cover a subsequent frontier**
   - Pass 2 (forward): Availability
   - Earliest: anticipated, but not yet available
3. ***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
4. ***Cleaning up***
   - Pass 4 (backward): **Remove unneeded temporary assignments**

## Remarks

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



## Monday's Class

- Region-Based Analysis [ALSU 9.7]
- Reminder: Assignment #2 due Wednesday midnight