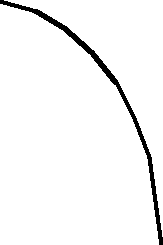
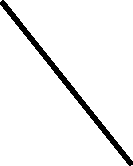
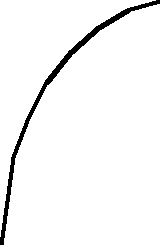
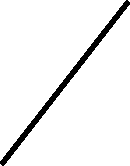
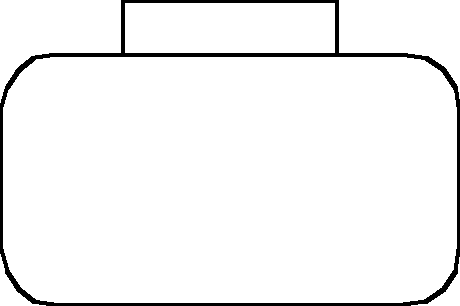
**Compiler Optimization LICM: Loop Invariant Code Motion**

* **If two loops do not have the same header:**
  + they are either disjoint, or
  + one is entirely contained (nested within) the other
    - inner loop: one that contains no other loop.
* **If two loops share the same header:**
  + Hard to tell which is the inner loop
  + Combine as one a

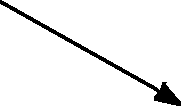


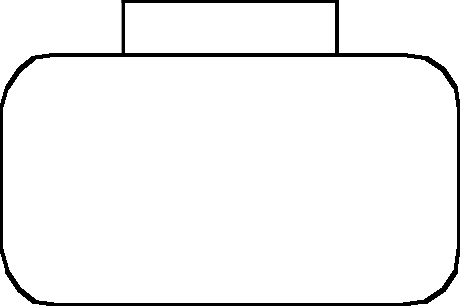
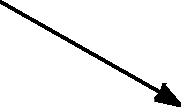
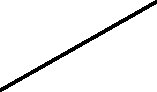
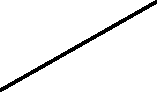
b c



rest of loop

header

* **Optimizations often require code to be executed once before the loop**
* **Create a preheader basic block for every loop**



preheader

rest of loop

header



* **Define loops in graph theoretic terms**
* **Definitions and algorithms for:**
  + Dominators
  + Back edges
  + Natural loops

**Loop-Invariant Computation and Code Motion**

* **A loop-invariant computation:**
  + a computation whose value does not change as long as control stays within the loop
* **Code motion:**
  + to move a statement within a loop to the preheader of the loop

**header**

B, C defined outside



of the loop

Function of loop inv

yes

yes

**A = B + C F = A + 2**

**E = 3**

yes constant

**outside loop**

**D = A + 1**

no One def inside loop,

and one outside

# Algorithm

##### Observations

* + Loop invariant
    - operands are defined outside loop or invariant themselves
  + Code motion
    - not all loop invariant instructions can be moved to

preheader

##### Algorithm

* + Find invariant expressions
  + Conditions for code motion
  + Code transformation

**Detecting Loop Invariant**

**Computation**

* Compute reaching definitions
* Mark INVARIANT if

all the definitions of B and C that reach a statement A=B+C are outside the loop

* + constant B, C?
* Repeat: Mark INVARIANT if
  + all reaching definitions of B are outside the loop, or
  + there is exactly one reaching definition for B, and it is from a loop-invariant statement inside the loop
  + similarly for C

until no changes to set of loop-invariant statements occur.

**D = A + 1**

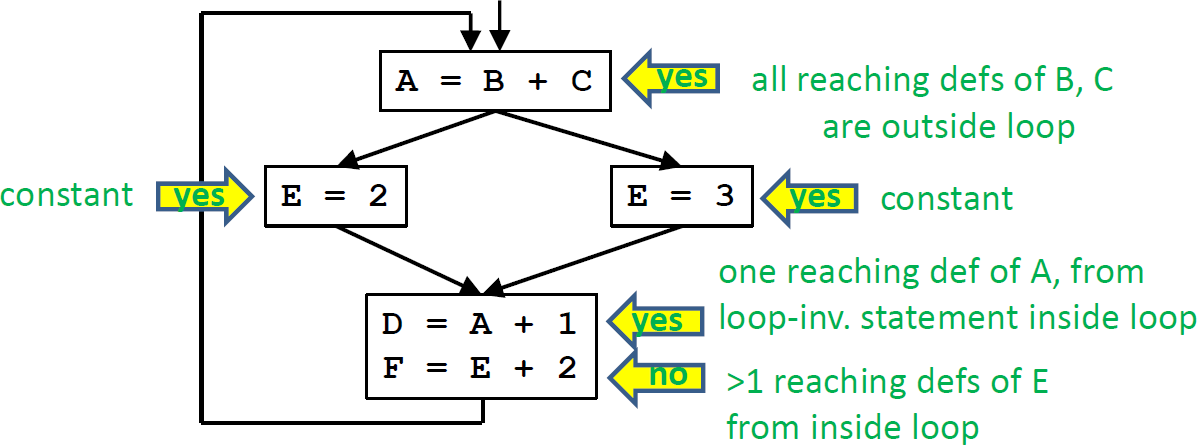
**F = E + 2**

**E = 3**

**A = B + C**

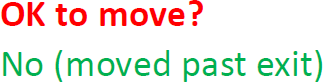
**E = 2**

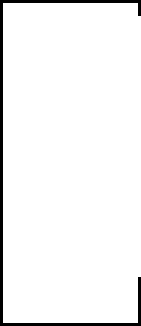




# Conditions for Code Motion

* **Correctness: Movement does not change semantics of program**
* **Performance: Code is not slowed down**

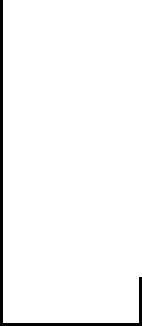




**.**

**.**

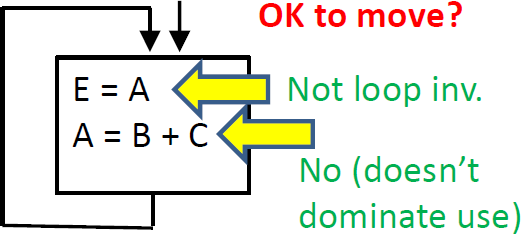
A = B + C



**.**

**.**

A = B + C

* **Basic idea: defines once and for all**
* control flow: once?

Code dominates all exists

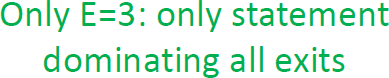
* other definitions: for all? No other definition
* other uses: for all?

Dominates use or no other reaching defs to use

Given: a set of nodes in a loop

* **Compute reaching definitions**
* **Compute loop invariant computation**
* **Compute dominators**
* **Find the exits of the loop (i.e., nodes with successor outside loop)**
* **Candidate statement for code motion:**
  + loop invariant
  + in blocks that dominate all the exits of the loop
  + assign to variable not assigned to elsewhere in the loop
  + in blocks that dominate all blocks in the loop that use the variable assigned
* **Perform a depth-first search of the blocks**
  + Move candidate to preheader if all the invariant operations it depends upon have been moved

**D = A + 1 F = E + 2**

**D = A + 1 outside loop**

**E = 3**

**A = B + C**

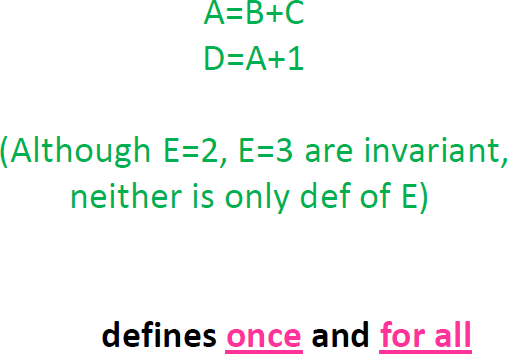
**E = 2**



**E = 3**

**A = B + C**

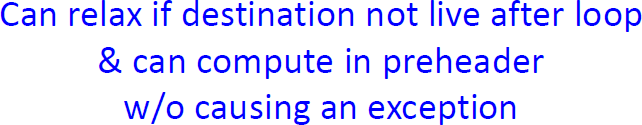
**header**





# More Aggressive Optimizations

* Gamble on: most loops get executed
  + Can we relax constraint of dominating all exits?





**A = B + C E = A + D**

**exit**

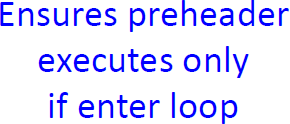
**D = …**

* Landing pads

**While p do s**  **if p {**

**}**

**preheader repeat**

**s**

**until not p;**

# LICM Summary

##### Precise definition and algorithm for loop invariant computation

* **Precise algorithm for code motion**
* **Use of reaching definitions and dominators in optimizations**

**Induction Variables and**

**Strength Reduction**

1. Overview of optimization
2. Algorithm to find induction variables

**i = 0**

**FOR i = 0 to 100**

###### A[i] = 0;

**L2: IF i>=100 GOTO L1**

###### t1 = 4 \* i t2 = &A + t1

**\*t2 = 0 i = i+1 GOTO L2**

###### L1:

* A basic induction variable is
  + a variable X whose only definitions within the loop are

assignments of the form:

X = X+c or X = X-c,

where **c** is either a constant or a loop-invariant variable.

* An **induction variable** is
* a basic induction variable, or
* a variable defined once within the loop, whose value is a linear function of some basic induction variable at the time of the definition: A = c1 \* B + c2
* The FAMILY of a basic induction variable B is
* the set of induction variables A such that each time A is assigned in the loop, the value of A is a linear function of B.

### Strength reduction:

* + A is an induction variable in family of basic induction variable B (A = c1 \*B + c2)
    - Create new variable: A’
    - Initialization in preheader: A’= c1 \* B + c2;
    - Track value of B: add after B=B+x: A’=A’+x\*c1;
    - Replace assignment to A: A=A’

1. Optimizing non-basic induction variables
   * copy propagation
   * dead code elimination
2. Optimizing basic induction variables
   * Eliminate basic induction variables used only for
     + calculating other induction variables and loop tests
   * Algorithm:
     + Select an induction variable A in the family of B, preferably with simple constants (A = c1 \* B + c2).
     + Replace a comparison such as

**if B > X goto L1**

with

**if (A’ > c1** \* **X + c2) goto L1** (assuming c1 is positive)

* + - if B is live at any exit from the loop, recompute it from A’

– After the exit, B = (A’ - c2) / c1

* **A BASIC induction variable in a loop L**
  + a variable X whose only definitions within L are assignments of the form: X = X+c or X = X-c, where c is either a constant or a loop-invariant variable.

###### Algorithm: can be detected by scanning L

* Example:

**k = 0;**

**for (i = 0; i < n; i++) { k = k + 3;**

**… = m;**

**if (x < y)**

**k = k + 4; if (a < b)**

**m = 2 \* k;**

**k = k – 2;**

**… = m;**

*Each iteration may execute a different number of increments/decrements!!*

* **Key idea:**
  + For each induction variable A, (A = c1\*B+c2 at time of definition)
    - variable A’ holds expression c1\*B+c2 at all times
    - replace definition of A with A=A’ only when executed
* **Result:**
  + Program is correct
  + Definition of A does not need to refer to B

###### Let B be a basic induction variable

* + Find all induction variables A in family of B:
    - A = c1 \* B + c2

(where B refers to the value of B at time of definition)

###### Conditions:

* + If A has a single assignment in the loop L, and assignment

is one of:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **=** | **B** | **\*** | **c** |
| **A** | **=** | **c** | **\*** | **B** |
| **A** | **=** | **B** | **/** | **c** |

**A = B + c A = c + B A = B – c A = c – B**

(assuming **A** is real)

* + OR, ... (next page)

**(continued)**

Let D be an induction variable in the family of B (D = c1\* B

+ c2)

* + - If A has a single assignment in the loop L, and assignment is one of:

**A = D \* c A = c \* D**

**A = D / c** (assuming **A** is real)

**A = D + c A = c + D A = D – c A = c – D**

* + - No definition of D outside L reaches the assignment to A
    - Between the lone point of assignment to D in L and the assignment to A,

there are no definitions of B

# Summary

###### Precise definitions of induction variables

* **Systematic identification of induction variables**

###### Strength reduction

* **Clean up:**
  + eliminating basic induction variables
    - used in other induction variable calculations
    - replacement of loop tests
  + eliminating other induction variables
    - standard optimizations