COMP 512
Rice University
Spring 2015

Loop Invariant Code Motion

— A Simple Classical Approach —

Citation numbers refer to entries in the EaC2e bibliography.

Background

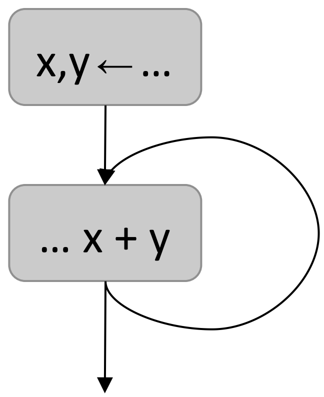
Code in loops executes more often than code outside loops

- An expression is *loop invariant* if it computes the same value, independent of the iteration in which it is computed
- Loop invariant code can often be moved to a spot before the loop
  - ♦ Execute once and reuse the value multiple times
  - ♦ Reduce the execution cost of the loop

Techniques seem to fall into two categories

- Ad-hoc graph-based algorithms
- Complete data-flow solutions
- Think of loop invariant as redundant across iterations
- Always lengthens live range

Loop-invariant Code Motion



Neither x nor y change in loop

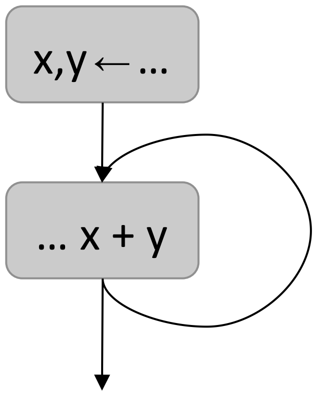
Example

- x + y is invariant in the loop
  - Computed on each iteration
  - Subexpressions don't change
- Move evaluation out of loop
  - Need block to hold it
  - Use existing block or insert new

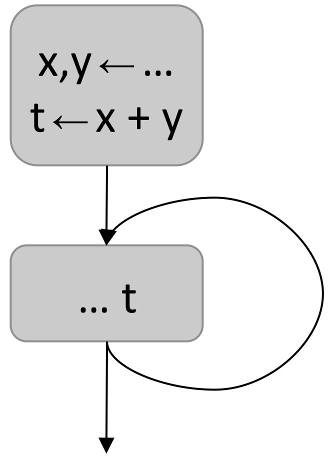
Relationship to redundancy

- x + y is redundant along back edge
- x + y is not redundant from loop entry
- If we add an evaluation on the entry edge, x + y in the loop is redundant

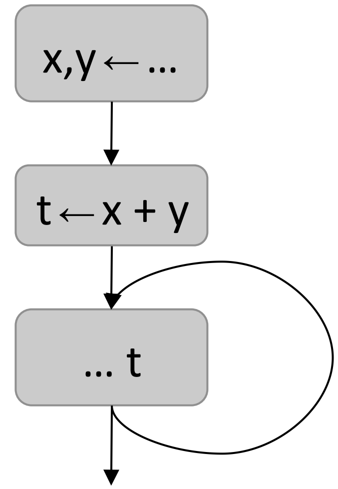
Loop-invariant Code Motion



Neither x nor y
change in loop



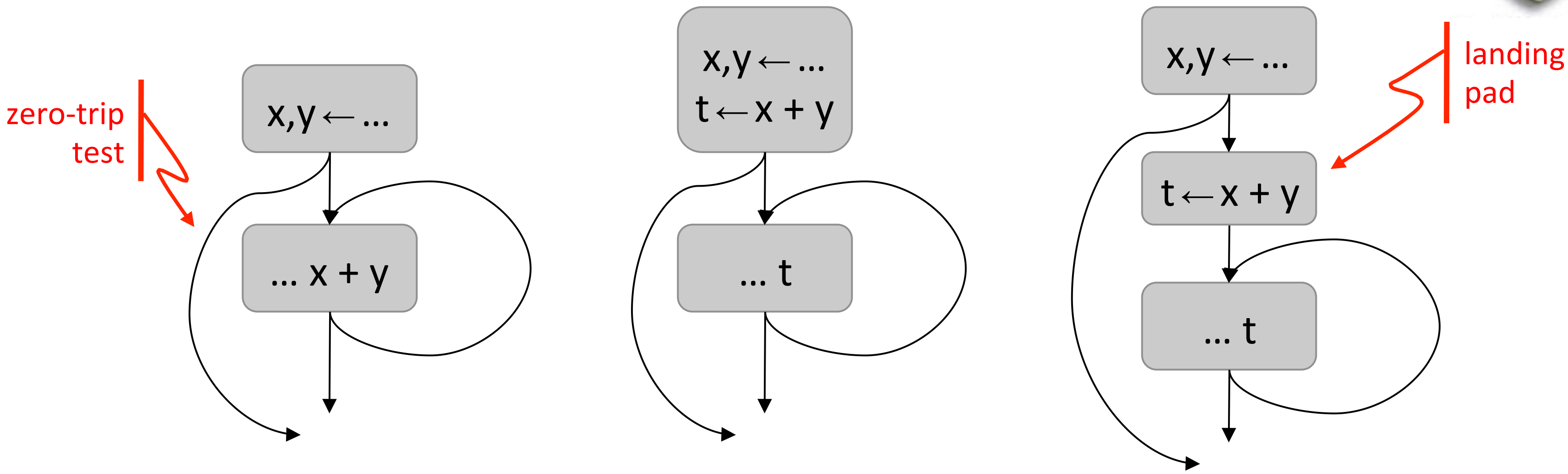
Option 1
Move x + y into
predecessor



Option 2
Create a block
for x + y

What's the difference?

Loop-invariant Code Motion



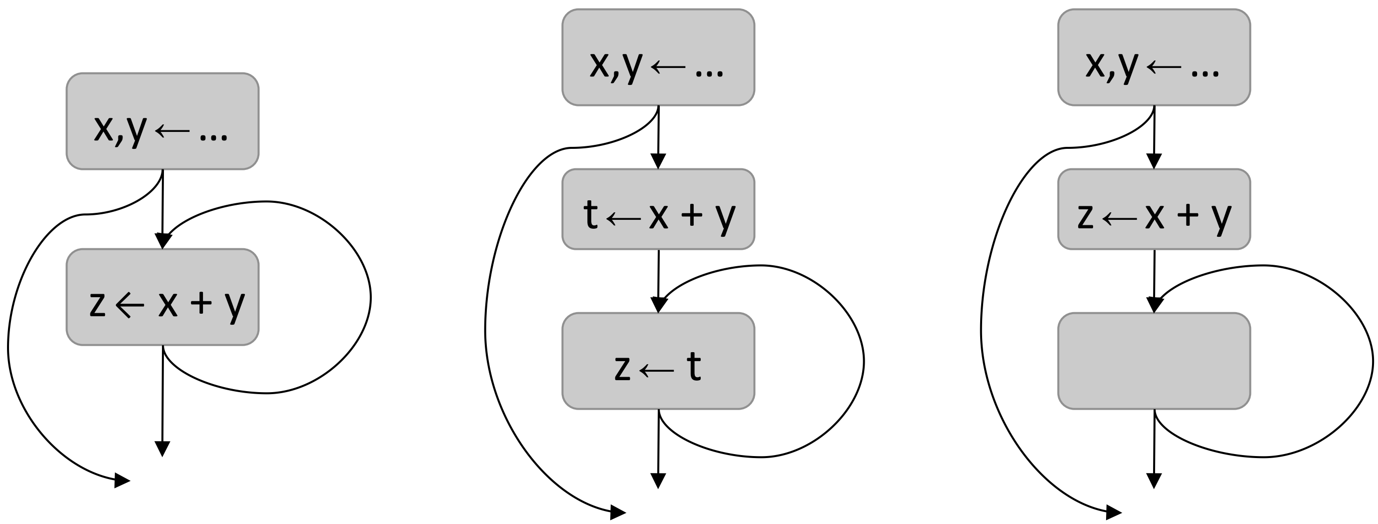
Speculative

Conservative

In practice, many loops have a *zero-trip* test

- Determines whether to run the first iteration
- Moving x + y above zero-trip test is *speculative*
  - Lengthens path around the loop
  - LICM techniques often create a *landing pad*

Loop-invariant Code Motion



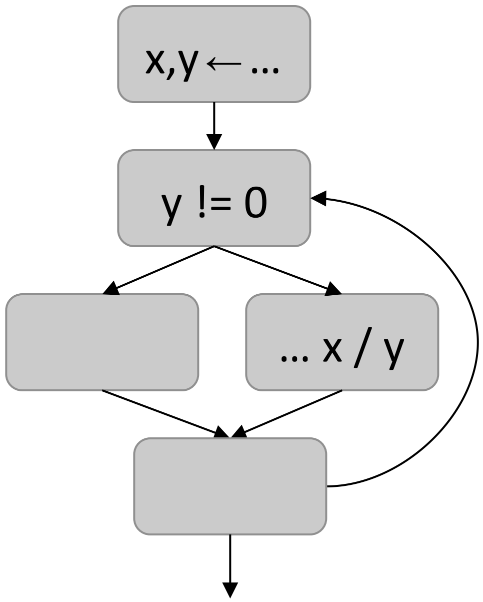
Move expression

Move assignment

Another difference between methods

- Some methods move expression evaluation, but not assignment
  - Easier safety conditions, easier to manage transformation
  - Leaves a copy operation in the loop (*may coalesce*)
- Other methods move the entire assignment
  - Eliminates copy from loop, as well

Loop-invariant Code Motion



None of x, y, or z change in loop

Note that y is invariant in this classic example, so we could move the test, as well. See Cytron, Lowry, & Zadeck.

Control flow

- Another source of speculation
  - ♦ Move it & lengthen other path
  - ♦ Don't move it

```
while (…)
{
  if (y != 0)
     then z = x/y
     else z = x
}
```

- Divergence may be an issue
- Don't want to move the op if doing so introduces an exception
- If that path is hot, then y != 0 and we might move it.

Would like to use branch probabilities

Loop-invariant Code Motion

Pedagogical Plan

1. A simple and direct algorithm (*today*)
2. Lazy code motion, a data-flow solution based on availability
3. Cytron, Lowry, & Zadeck's **SSA**-based approach (*probably not*)

Which is best?

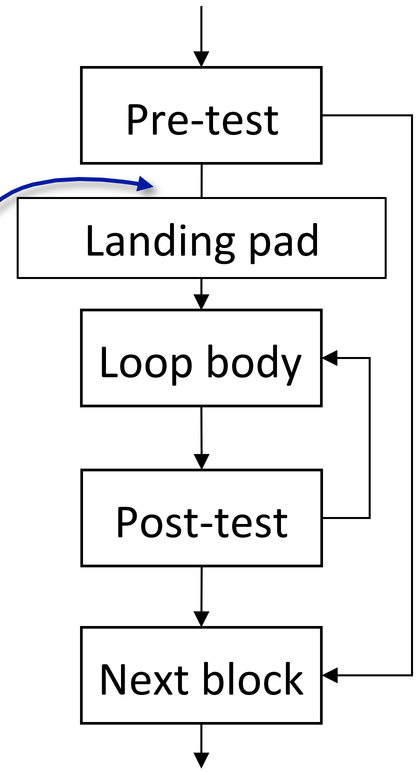
- The authors of 2 & 3 would each argue for their technique
- In practice, each has strengths and weaknesses
- Taught 3 last year and had someone implement it in **LLVM**
  - ♦ Surprising set of complications in **SSA** implementation
    - → Moving around all of those definitions proved problematic
    - → Creating new names, recreating **SSA** name space … too much work for the benefits
  - ♦ Cannot recommend **CLZ** in practice

Loop Shape Is Important

Loops

- Evaluate condition before loop (if needed)
- Evaluate condition after loop
- Branch back to the top (if needed)

Merges test with last block of loop body



COMP 412 teaches this loop shape. It creates a natural place to insert a landing pad.

while, for, do, & *until* all fit this basic model

For tail recursion, unroll the recursion once …

Paper by Mueller & Whalley in SIGPLAN '92 PLDI about converting arbitrary loops to this form.

Loop-invariant Code Motion

Preliminaries

- Need a list of loops, ordered inner to outer in each loop nest
  - ♦ Define "natural loop" as one formed by a back edge with respect to **DOM**
    - → *An edge (t,h) is a back edge if h* ***DOM*** *t*
    - → *h is the loop's header and t is a tail of the loop*
    - → *The loop's body includes all predecessors back to h*
  - ♦ Natural loops can nest
    - → *Given two headers,* h_1 *and* h_2, *the loops are nested if* h_1 ***DOM*** h_2 *or* h_2 ***DOM*** h_1
    - → *If* h_1 *&* h_2 *are unrelated by* ***DOM****, loops are not nested*
    - → *Inner to outer order is defined by* ***DOM***
- Need a landing pad on each loop
  - ♦ Insert above loop header & redirect inbound edges

Given back edge (h,t)

```
body ← { h }
stack ← empty
push(t)
while (stack is nonempty) {
    n ← pop()
    if n ∉ body then {
        body ← body ∪ { n }
        for each x ∈ pred(n)
        push(x)
    }
}
```

Find the loop body formed by a back edge (*t*,*h*)

Loop-invariant Code Motion

A Simple Algorithm

```
for each l in LOOPLIST do
   FactorInvariants(l )
```

```
FactorInvariants(loop)
   MarkInvariants(loop)
   for each expr e ∈ loop      (x ← y + z)
      if x is marked invariant then
         begin
            allocate a new name t
            replace o with x ← t
            insert t ← e in landing pad
               for loop
      end
```

```
MarkInvariants(loop )
   LOOPDEF ← ∪_{block b ∈ loop} DEF(b)
   for each op o ∈ loop      (x ← y + z)
      mark x as invariant
      if y ∈ LOOPDEF
        then mark x as variant
      if z ∈ LOOPDEF
        then mark x as variant
```

Example

Consider the following simple loop nest

```
do i ← 1 to 100
   do j ← 1 to 100
      do k ← 1 to 100
         a(i,j,k) ← i  * j * k
         end
      end
   end
```

Dimensions addressed	*Multiplies*
3,000,000	2,000,000

Original Code

Example

Consider the following simple loop nest

	Dimensions addressed	*Multiplies*
do i ← 1 to 100		
do j ← 1 to 100		
t_1 ← addr(a(i,j))	20,000	
t_2 ← i * j		10,000
do k ← 1 to 100		
t_1(k) ← t_2 * k	1,000,000	1,000,000
end		
end		
end		

After LICM on the Inner Loop

Example

Consider the following simple loop nest

	Dimensions addressed	Multiplies
do i ← 1 to 100		
t_3 ← *addr(a(i))*	100	
do j ← 1 to 100		
t_1 ← addr(t_3(j))	10,000	
t_2 ← i * j		10,000
do k ← 1 to 100		
t_1(k) ← t_2 * k	1,000,000	1,000,000
end		
end		
end		

After Doing Middle Loop

Safety of Loop-invariant Code Motion

What happens if we move an expression that generates an exception?

- Maybe the fault happens at a different time in execution
- Maybe the original code would not have faulted

Ideally, the compiler should delay the fault until it would occur

Options

- Mask fault & add a test to loop body
- Replace value with one that causes fault when used
- Block read access to the faulted value
- Patch the executable with a bad opcode
- Generate two copies of the loop and rerun with original code
- Never move an evaluation that can fault

Applies to all the algorithms

Profitability of Loop-invariant Code Motion

- Does the loop body always execute?
  - ♦ Placement of the landing pad is crucial
- Lengthen any paths?
  - ♦ Conservative code motion: not a problem
  - ♦ Speculative code motion: might be an issue
    - → Rely on estimates of branch frequency?
- Register pressure?
  - ♦ Target of moved operation has longer live range
  - ♦ Might shorten live ranges of operands of moved op

Backus' dilemmna

Applies to all the algorithms

Shortcomings of the Simple LICM Algorithm

- Moves code out of conditionals in a speculative fashion
  - ♦ Ignores control flow inside the loop
  - ♦ Can imagine a more sophisticated approach based on control-dependence
- Moves only evaluation, not assignment
  - ♦ To move assignments would require a more complex approach to naming both arguments and results (*such as the SSA name space?*)
- Only finds first order invariants

```
for i ← 1 to n
    for j ← 1 to m
        …
        x ← a * b        First order invariant
        y ← x * c        Second order invariant
        …
        end
    end
```

Need to iterate on a loop until it stabilizes, then move on