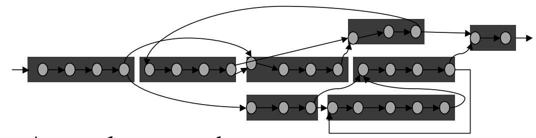
KT-2

P3 / 2006

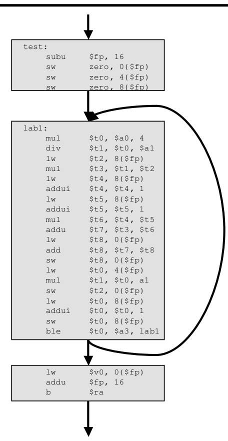
Loop Optimizations

Representing the control flow of a program

- Control forms a graph



- A very large graph
- Observation
  - lot of straight-line connections
  - simplify the graph by grouping some instructions

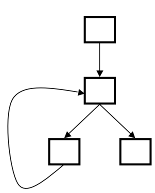


Loop Optimizations

- Important because lots of execution time occurs in loops
- First, we will identify loops
- Then, we will study three optimizations
  - Loop-invariant code motion
  - Strength reduction
  - Induction variable elimination

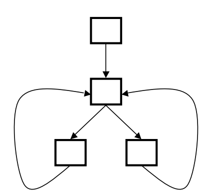
What is a Loop?

- Set of nodes
- Loop header
  - Single node
  - All iterations of loop go through header
- Back edge

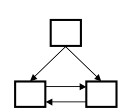


Anomalous Situations

- Two back edges, two loops, one header
- Compiler merges loops



- No loop header, no loop



Defining Loops With Dominators

- Concept of *dominator*
  - Node n dominates a node m if all paths from start node to m go through n
- If d_1 and d_2 both dominate m, then either
  - d_1 dominates d_2, or
  - d_2 dominates d_1 (but not both – look at path from start)
- *Immediate dominator* of m – last dominator of m on any path from start node

Dominator Problem Formulation

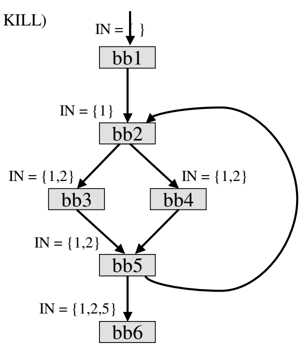
- A cross product of the lattice for each basic block:
  - Lattice per basic block

$\top$ = dominated
↓
$\perp$ = not dominated

- Flow direction: Forward Flow
- Dataflow Equations:
  - GEN = { b_k | b_k is the current basic block }
  - KILL = { }
  - OUT = GEN ∪ (IN - KILL)
  - IN = ∩ OUT

Computing Dominators

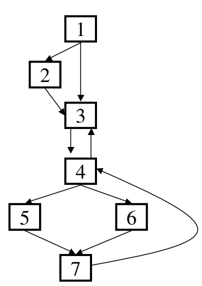
OUT = GEN ∪ (IN - KILL)
IN = ∩ OUT



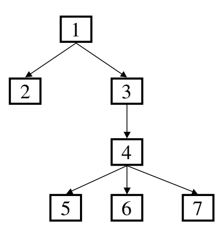
Dominator Tree

- Nodes are nodes of control flow graph
- Edge from d to n if d is the immediate dominator of n
- This structure is a tree
- Rooted at start node

Example Dominator Tree



Control-flow graph

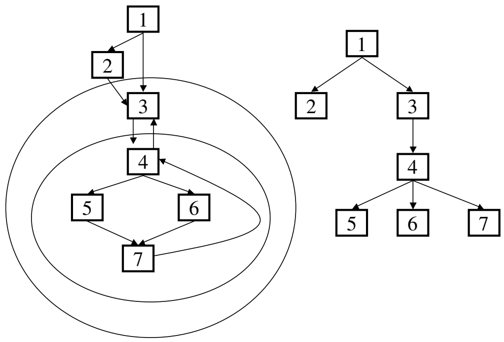


Dominator tree

Identifying Loops

- Unique entry point – header
- At least one path back to header
- Find edges whose heads dominate tails
  - These edges are back edges of loops
  - Given a back edge n→d
  - Loop consists of n plus all nodes that can reach n without going through d
    (all nodes "between" d and n)
  - d is loop header

Two Loops in Example

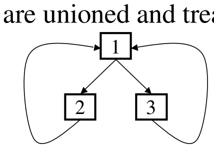


Loop Construction Algorithm

<u>insert</u>(m)
  if m ∉ loop then
    loop = loop ∪ {m};
    push m onto stack;
<u>loop</u>(d,n)
  loop = ∅; stack = ∅; insert(n);
  while stack not empty do
    m = pop stack;
    for all p ∈ pred(m) do insert(p);

Nested Loops

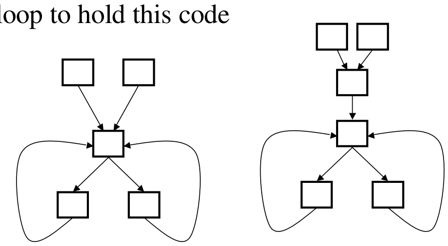
- If two loops do not have same header then
  - Either one loop (inner loop) is contained in the other (outer loop)
  - Or the two loops are disjoint
- If two loops have same header, typically they are unioned and treated as one loop



Two loops:
{1,2} and {1,3}
Unioned: {1,2,3}

Loop Preheader

- Many optimizations stick code before the loop
- Put a special node (loop preheader) before the loop to hold this code



Loop Optimizations

- Now that we have the loop, we can optimize it!
- Loop invariant code motion
  - Stick loop invariant code in the header

Loop Invariant Code Motion

If a computation produces the same value in every loop iteration, move it out of the loop.

```
for  i = 1 to N
   x = x + 1
   for  j = 1 to N
      a[i,j] = 100*N + 10*i + j + x
```

```
t1 = 100*N
for  i = 1 to N
   x = x + 1
   t2 = t1 + 10*i + x
   for  j = 1 to N
      a[i,j] =  t2 + j
```

Detecting Loop Invariant Code

- A statement is *loop-invariant* if operands are
  - Constant,
  - Have all reaching definitions outside loop, or
  - Have exactly one reaching definition, and that definition comes from an invariant statement
- Concept of exit node of loop
  - node with successors outside loop

Loop Invariant Code Detection Algorithm

for all statements in loop

  if operands are constant or have all reaching definitions outside loop, mark statement as invariant

do

  for all statements in loop not already marked invariant

    if operands are constant, have all reaching definitions outside loop, or have exactly one reaching definition from invariant statement

    then mark statement as invariant

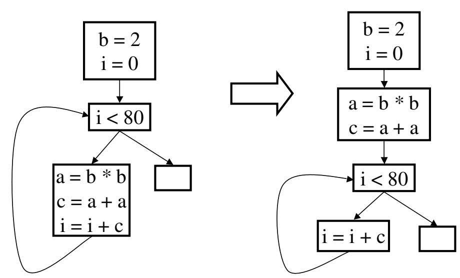
until there are no more invariant statements

Loop Invariant Code Motion

- Conditions for moving a statement s: x = y+z into loop header:
  - The node containing s dominates all exit nodes of loop
    - If it does not, some use after loop might get wrong value
    - Alternate condition: definition of x from s reaches no use outside loop (but moving s may increase run time)
  - No other statement in loop assigns to x
    - If one does, assignments might get reordered
  - No use of x in loop is reached by definition other than s
    - If one is, movement may change value read by use

Order of Statements in Preheader

Preserve data dependences from original program (can use order in which discovered by algorithm)



Induction Variables

Example:

```
for j = 1 to 100
   *(&A + 4*j) = 202 - 2*j
```

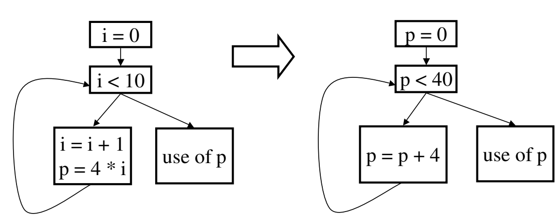
Base induction variable:

J = 1, 2, 3, 4, …..

Derived induction variable &A+4*j:

&A+4*j = &A+4, &A+8, &A+12, &A+16, ….

Induction Variable Elimination



What are induction variables?

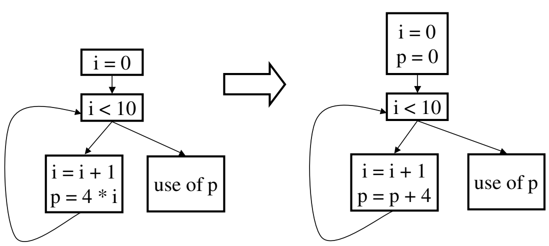
- x is an *induction variable* of a loop L if
  - variable changes its value on every loop iteration
  - the value is a function of number of iterations of the loop
- In many programs, this function is often a linear function

  Example: for loop index variable j, function d + c*j

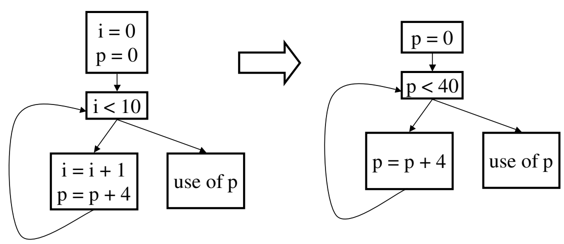
What is an Induction Variable?

- Base induction variable
  - Only assignments in loop are of form i = i ± c
- Derived induction variables
  - Value is a linear function of a base induction variable
  - Within loop, j = c*i + d, where i is a base induction variable
  - Very common in array index expressions – an access to a[i] produces code like p = a + 4*i

Strength Reduction for Derived Induction Variables



Elimination of Superfluous Induction Variables



Three Algorithms

- Detection of induction variables
  - Find base induction variables
  - Each base induction variable has a family of derived induction variables, each of which is a linear function of base induction variable
- Strength reduction for derived induction variables
- Elimination of superfluous induction variables

Output of Induction Variable Detection Algorithm

- Set of induction variables
  - base induction variables
  - derived induction variables
- For each induction variable j, a triple <i,c,d>
  - i is a base induction variable
  - the value of j is i*c+d
  - j belongs to family of i

Induction Variable Detection Algorithm

Scan loop to find all base induction variables

do

- Scan loop to find all variables k with one assignment of form k = j*b where j is an induction variable with triple <i,c,d>
  - make k an induction variable with triple <i,c*b,d>
- Scan loop to find all variables k with one assignment of form k = j±b where j is an induction variable with triple <i,c,d>
  - make k an induction variable with triple <i,c,b±d>

until no more induction variables are found

Strength Reduction

```
t = 202
for j = 1 to 100
  t = t - 2
  *(abase + 4*j) = t
```

Base induction variable:

J = 1, 2, 3, 4, …..

1 1 1

Derived induction variable 202 - 2*j

t = 202, 200, 198, 196, …..

-2 -2 -2

Derived induction variable abase+4*j:

abase+4*j = abase+4, abase+8, abase+12, abase+16, ….

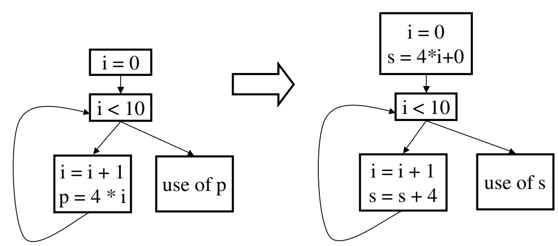
4 4 4

Strength Reduction Algorithm

for all derived induction variables j with triple <i,c,d>

- Create a new variable s
- Replace assignment j = c*i + d with j = s
- Immediately after each assignment i = i + e, insert statement s = s + c*e (c*e is constant)
- Place s in family of i with triple <i,c,d>
- Insert s = c*i + d into preheader

Strength Reduction for Derived Induction Variables



Example

```
double A[256], B[256][256]
j = 1


while(j<100)
  A[j] = B[j][j]
  j = j + 2
```

```
double A[256], B[256][256]
j = 1
a = &A + 8
b = &B + 2056   // 2048+8
while(j<100)
  *a = *b
  j = j + 2
  a = a + 16
  b = b + 4112  // 4096+16
```


Induction Variable Elimination

Choose a base induction variable i such that only uses of i are in

- termination condition of the form i < n
- assignment of the form i = i + m

Choose a derived induction variable k with <i,c,d>

Replace termination condition with k < c*n+d

Induction Variable Wrap-up

There is lots more to induction variables

- more general classes of induction variables
- more general transformations involving induction variables

Compiler Optimization Summary

- Wide range of analyses and optimizations
- Dataflow analyses and corresponding optimizations
  - reaching definitions, constant propagation
  - live variable analysis, dead code elimination
- Induction variable analyses and loop optimizations
  - Strength reduction
  - Induction variable elimination
  - Important because lots of time is spent in loops