Loop-invariant Code Motion

15-745 Optimizing Compilers
Spring 2006

Peter Lee

Reminders

- Task 1 test programs are due
  - Task 1 due in one week
  - See the **Internals doc** and **Advice Column**
- Read 7.1-4 (control-flow analysis) and 13.2 (loop-invariant code motion)

Loops

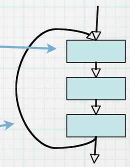
- Loops are **extremely** important
  - the "90-10" rule
- Loop optimization involves
  - understanding control-flow structure
  - sensitivity to side-effecting operations
  - extra care in some transformations such as register spilling

Classical loop optimizations

- **Hoisting of loop-invariant computations**
  - pre-compute before entering the loop
- **Elimination of induction variables**
  - change p=i*w+b to p=b+w, when w,b invariant
- **Elimination of null and array-bounds checks**
  - use laws of arithmetic to prove integer range
- **Loop unrolling**
  - to reduce number of control transfers
- **Loop permutation**
  - to improve cache memory performance

Finding loops

* To optimize loops, we need to find them!
* Specifically:
  * loop-header node(s)
    * nodes in a loop that have immediate predecessors not in the loop
  * back edge(s)
    * control-flow edges to previously executed nodes
  * all nodes in the loop body



Control-flow analysis

* L3 has only well-structured control-flow constructs
* Finding L3 loops is easy
  * the translator can mark every header node and back edge when creating the IR
* But many languages have goto and other complex control, so loops can be hard to find
* Determining the control structure of a program is called **control-flow analysis**

Task note

* We will describe here the classical approach to control-flow analysis for imperative, first-order languages
* This is a general approach, suitable even for languages with goto
* But for L3, it is much easier simply to have the translator identify any loops it creates

Terminology alert

* dominators and dominator trees
* back edge
* loop header
* natural loop

Dominators

* a dom b
  * node a dominates b if every possible execution path from entry to b includes a
* a sdom b
  * a strictly dominates b if a dom b and a≠b
* a idom b
  * a immediately dominates b if there is no c such that a sdom c and c sdom b

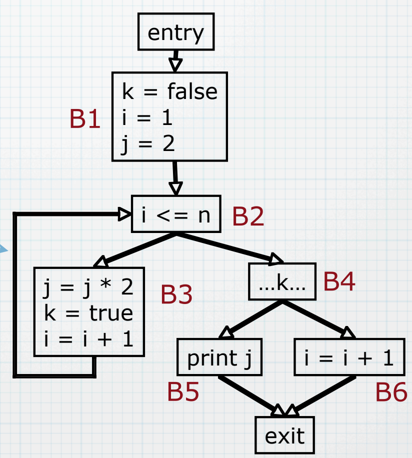
Some properties

* idom(n) is unique
* The dom relation is a partial ordering
  * reflexive, antisymmetric, and transitive

Back edges and loop headers

A control-flow edge from node b to a is a back edge if a dom b

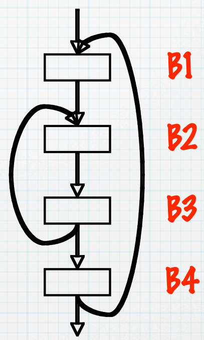
Furthermore, in that case node a is a loop header



Natural loop

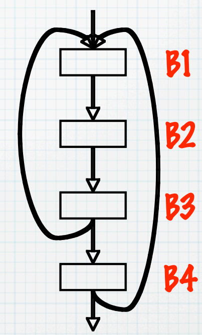
* Consider a back edge from node n to node h
* The natural loop of n→h is the set of nodes L such that for all x∈L:
  * h dom x and
  * there is a path from x to n not containing h

A simple example...



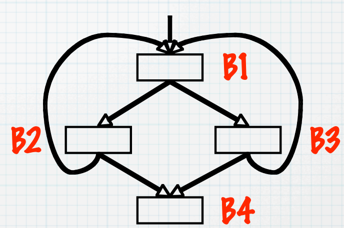
nested loops

What about this case?



loop with "continue"

What about this case?



conditional in loop

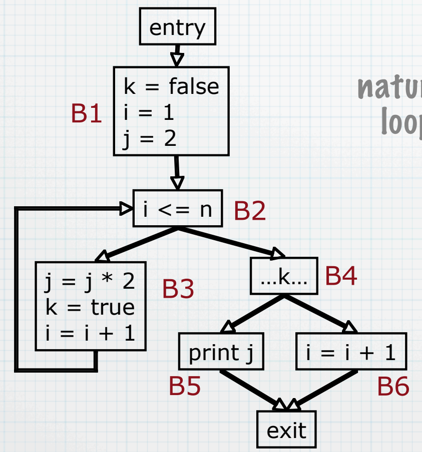
Nested loops

* Normally we will want to focus attention on the inner-most loops
* This requires identifying not only the loops, but the nesting structure

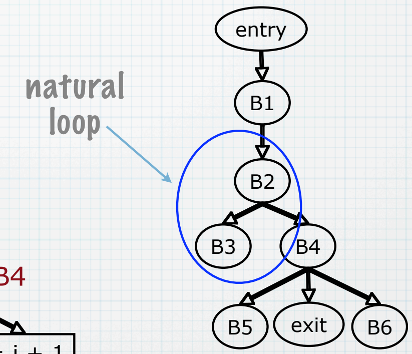
Dominator trees

* Observe: Every node has at most one immediate dominator
* Therefore: the immediate dominator relation defines a tree structure
  * node **n** is the parent of node **m** if **n idom m**

control-flow graph



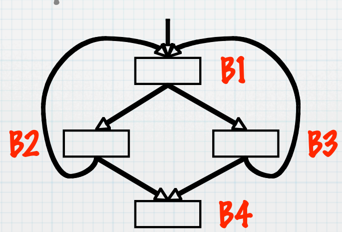
dominator tree



Back edges point to ancestors in the tree

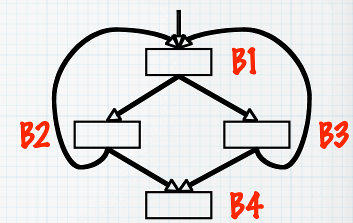
Limitations of natural loops

* The notion of natural loop is only approximate
* Specifically, consider the case of two natural loops with the same header:




```
while (...)
  if (p) {...}
  else {...}
```

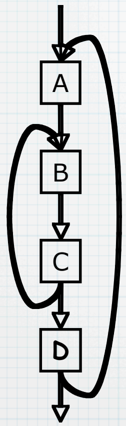
What if p is loop invariant?

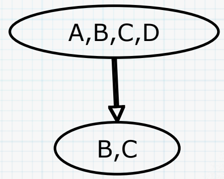



```
while (...)
  if (i<j) {...; i++;}
  else if (i>j) {...; i--;}
```

Nested loops?

In general, when there is a shared header, will consider this a single loop



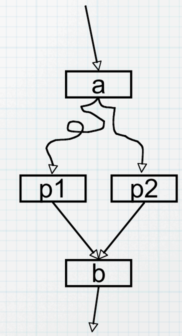


In a **loop-nest tree**, each node represents the blocks of a loop, and parent nodes are enclosing loops

The leaves of the tree are the inner-most loops

Computing dominators

* Observe: if **a dom b**, then
  * **a** = **b**, or
  * **a** is the only immediate predecessor of **b**, or
  * **b** has more than one immediate predecessor, all of which are dominated by **a**



$$\mathrm{dom}(b) = \{b\} \cup \bigcap_{p \in \mathrm{pred}(b)} \mathrm{dom}(p)$$

Simple algorithm

```
dom(entry)  = {entry}

dom(n) =
  D = all nodes
  changed = true
  while (changed) {
    changed = false
    for each n≠entry {
      old = D
      D = {n} U ⋂_{p∈pred(n)} dom(p)
      if D ≠ old then changed = true
    }
  }
  return D
```

Computing idom

```
idom(n) =
  D = all nodes s such that s sdom n
  for each x ∈ D {
    for each y ∈ D-{x} {
      if y ∈ sdom(x) then
        D = D = {y}
    }
  }
  return D
```

Better algorithms

* Computing dominators is a classic problem in the study of algorithms
* The idom algorithm presented here runs in $O(e \cdot n^2)$, for a graph with n nodes and e edges
* Lengauer and Tarjan, in 1979, presented algorithms that run in $O(e \cdot \log(n))$ or better

Loop optimizations:

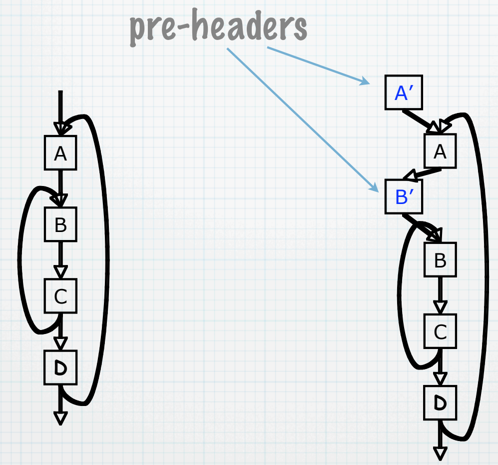
Hoisting of loop-invariant computations

Loop-invariant computations

* A definition
  * $t = x \oplus y$
* in a loop is **loop-invariant** if
  * x and y are constants, or
  * all reaching definitions of x and y are outside the loop, or
  * only one definition reaches x (or y), and that definition is loop-invariant

Hoisting

* In order to "hoist" a loop-invariant computation out of a loop, we need a place to put it
* We could copy it to all immediate predecessors of the loop header...
* ...But we can avoid code duplication by inserting a new block, called the **pre-header**



Hoisting conditions

- For a loop-invariant definition
  - d: t = x ⊕ y
- we can hoist d into the loop's pre-header if
  1. d's block dominates all loop exits at which t is live-out, and
  2. there is only one definition of t in the loop, and
  3. t is not live-out of the pre-header

We need to be careful...

- All hoisting conditions must be satisfied!

```
L0:
  t = 0
L1:
  i = i + 1
  t = a * b
  M[i] = t
  if i<N goto L1
L2:
  x = t
```

OK

```
L0:
  t = 0
L1:
  if i>=N goto L2
  i = i + 1
  t = a * b
  M[i] = t
  goto L1
L2:
  x = t
```

violates 1,3

```
L0:
  t = 0
L1:
  i = i + 1
  t = a * b
  M[i] = t
  t = 0
  M[j] = t
  if i<N goto L1
L2:
```

violates 2

Next time...

- Induction-variable elimination
- Bounds-checking elimination