15-745 Lecture 4b

Classical Loop Optimizations
Based on slides by Peter Lee

Common loop optimizations

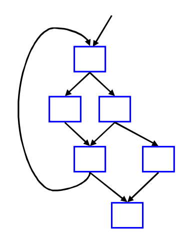
- Hoisting of loop-invariant computations
  - pre-compute before entering the loop
- Elimination of induction variables
  - change p=i*w+b to p=b,p+=w, when w,b invariant
- Loop unrolling
  - to reduce number of control transfers
- Loop permutation
  - to improve cache memory performance
- Elimination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

Finding Loops

- To optimize loops, we need to find them!
- Could use source language loop information in the abstract syntax tree…
- BUT:
  - There are multiple source loop constructs: for, while, do-while, even goto in C
  - Want IR to support different languages
  - Ideally, we want a single concept of a loop so all have same analysis, same optimizations
  - Solution: dismantle source-level constructs, then re-find loops from fundamentals

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

- Many languages have goto and other complex control, so loops can be hard to find in general
- Determining the control structure of a program is called control-flow analysis
- Based on the notion of dominators

Dominators

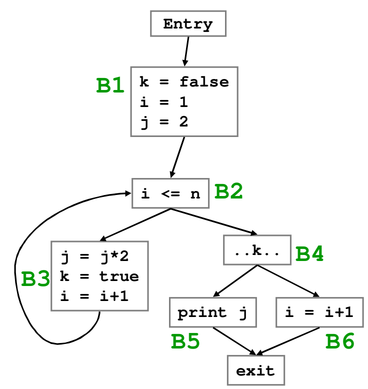
- a dom b
  - node a dominates b if every possible execution path from entry to b includes a
- a sdom b
  - a strictly dominates b if a dom b and a != b
- a idom b
  - a immediately dominates b if a sdom b, AND there is no c such that a sdom c and c sdom b

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 B3 to B2 is a back edge if B2 dom B3
- Furthermore, in that case node B2 is a loop header

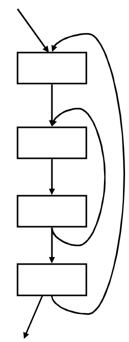


Natural loop

- 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

Examples

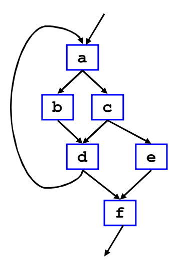
Simple example:



(often it's more complicated, since a source code FOR loop might need an if/then guard)

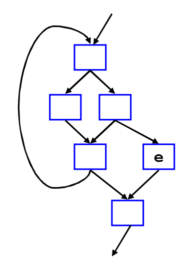
Examples

Try this:



Examples

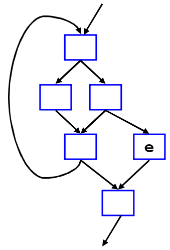
```
for (..) {
  if {
  …
  } else {
    …
    if (x) {
        e;
        break;
    )
  }
}
```



Examples

```
for (..) {
  if {
  …
  } else {
    …
    if (x) {
      e;
      break;
    )
  }
}
```

lexically, in loop, but not in natural loop

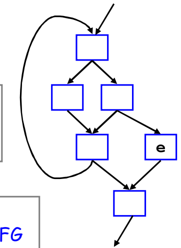


Examples

```
for (..) {
  if {
  …
  } else {
    …
    if (x) {
      e;
      break;
    )
  }
}
```

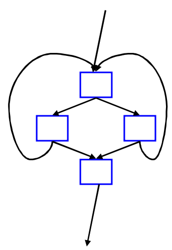
lexically, in loop, but not in natural loop

and another reason why CFG analysis is preferred over source/AST loops



Examples

- Yes it can happen in C



More later...

- We've already covered the straightforward dataflow computation of the dom relation.
- We'll have more to say about dominators, including how to compute them efficiently, in the future
  - Hint: they are part of computing SSA efficiently..

Loop optimizations: Hoisting of loop-invariant computations

Loop-invariant computations

- A definition

  t = x op y

  in a loop is (conservatively) 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
    - so keep marking iteratively

Loop-invariant computations

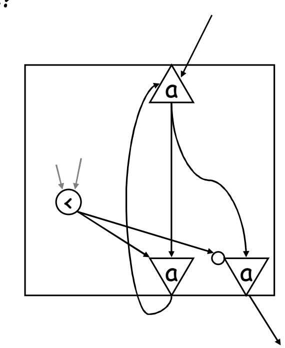
- Be careful:

```
t = expr;
for () {
     s = t * 2;
     t = loop_invariant_expr;
     x = t + 2;
     …
}
```

- Even though t's two reaching expressions are each invariant, s is not invariant…

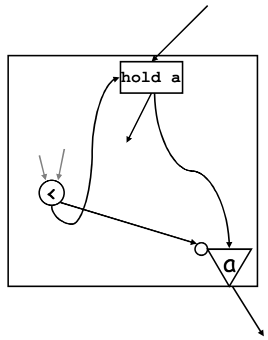
Loop-invariant computations

- In Pegasus! What does a basic loop-invariant variable look like?



Loop-invariant computations

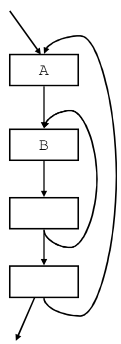
- In Pegasus! What does a basic loop-invariant variable look like?



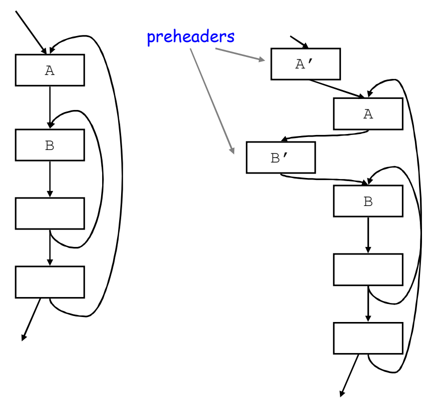
Hoisting

- 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 (except along the back-edge) of the loop header...
- ...But we can avoid code duplication by inserting a new block, called the pre-header

Hoisting



Hoisting



Hoisting conditions

- For a loop-invariant definition

  d: t = x op y

- we can hoist d into the loop's pre-header only if
  1. d's block dominates all loop exits at which t is live-out, and
  2. d is only the only 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

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
```

this def reaches

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:
```

this def reaches

violates 2

Announcements

- Tuesday's lecture is about efficient creation of minimal SSA form. There is a paper to read on the schedule page.
- If you get an error with CVS update....

Loop optimizations: Induction-variable Strength reduction

The basic idea of IVE

- Suppose we have a loop variable
  - i initially 0; each iteration i = i + 1
- and a variable that linearly depends on it:
  x = i * c1 + c2
- In such cases, we can try to
  - initialize x = i_o * c1 + c2 (execute once)
  - increment x by c1 each iteration

Is it faster?

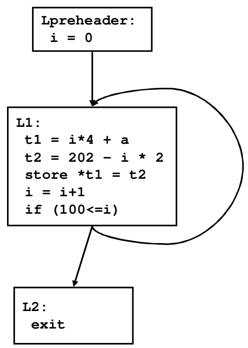
- On some hardware, adds are much faster than multiplies
- Furthermore, one fewer value is computed,
  - thus potentially saving a register
  - and decreasing the possibility of spilling

An example

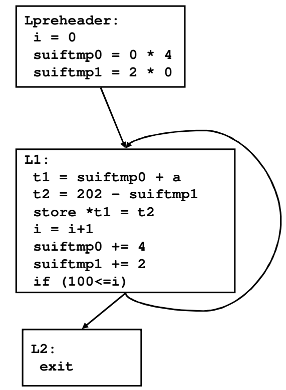
```
void p()
{
  int *a;
  int i;

  a = alloc(100,int);
  for (i=0; i<100; i=i+1)
    a[i] = 202 - 2 * i;
}
```

An example



An example



Loop preparation

- Before attempting IVE, it is best to perform first:
  - constant propagation & constant folding
  - copy propagation
  - loop-invariant hoisting

How to do it, step 1

- First, find the basic IVs
  - scan loop body for defs of the form
    x = x + c, where c is loop-invariant
  - record these basic IVs as
    x = (x, 1, c)
  - this represents the IV: x = x * 1 + c

How to do it, step 2

- Scan for derived IVs of the form
  k = i * c1 + c2
  - where i is a basic IV and this is the only def of k in the loop
- We say k is in the family of i
- Record as k = (i, c1, c2)

How to do it, step 3

- Iterate, looking for derived IVs of the form
  k = j * c1 + c2
  - where IV j =(i, a, b), and
  - this is the only def of k in the loop, and
  - there is no def of i between the def of j and the def of k
- Record as k = (i, a*c1, b*c1+c2)

How to do it, step 4

- For an induction variable k = (i, c1, c2)
  - initialize k = i * c1 + c2 in the preheader
  - replace k's def in the loop by
    k = k + c1
  - make sure to do this after i's def

Notes

- Are the c1, c2 constant, or just invariant?
  - if constant, then you can keep folding them: they're always a costant even for derived IVs
  - otherwise,they can be expressions of loop-invariant variables
- But if constant, can find IVs of the type
  x = i/b
  and know that it's legal, if b evenly divides the stride…

Is it faster? (2)

- On some hardware, adds are much faster than multiplies
- But…not always a win!
  - Constant multiplies might otherwise be reduced to shifts/adds that result in even better code than IVE
  - Scaling of addresses (i*4) might come for free on your processor's address modes
- So maybe: only convert **`i*c1+c2`** when c1 is loop invariant but not a constant