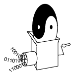


CS 412
Introduction to Compilers

Andrew Myers
Cornell University

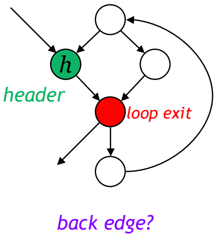
Lecture 30: Loop optimizations
13 Apr 01

Outline

- Loop optimizations
  - – Loop-invariant code motion
  - – Strength reduction
  - – Loop unrolling
  - – Array bounds checks
  - – Loop tiling ...
- Eliminating null checks

Dominators and loops

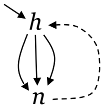
- *A* **dom** *B* if *B* is reachable only by going through *A*
- Defn of loop: set of strongly-connected nodes with single entry point: *loop header* node
- *loop header* dominates all other nodes in loop
- Loop must contain *back edge* w/ respect to dominance relationship: *n*→*h* where *h* **dom** *n*

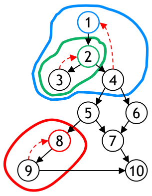


Completing control-flow analysis

- Dominator analysis identifies all *back edges*
- Each back edge *n*→*h* has an associated *natural loop* with *h* as its header: all nodes reachable from *h* that reach *n* without going through *h*
- For each back edge *n*→*h* , find its natural loop:

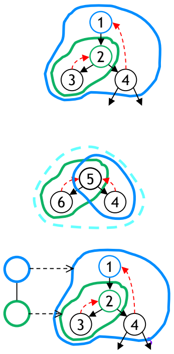
{*n′* | *n* reachable from *n′* in *G*-*h*} ∪ {*h*}





Control tree

- Nest loops based on subset relationship between natural loops
- Exception: natural loops may share same header; merge them into larger loop.
- Build control tree using nesting relationship



Redundant computation

```
for (int i=0; i<a.length; i++) {
   a[i] = a[i]+1;
}
```

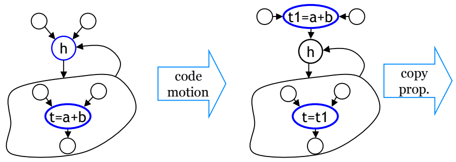
```
      i=0                              Lok1: t3=i*4
L0:   t0=a-4                                 t4=a+t3
      tlen=[t0]                              t0=a-4
      tcmp=i<t0                              tlen=[t0]
      if tcmp goto Lend else L1              tcmp=i<t0
L1:   t1=i*4                                 if tcmp got
      t2=a+t1                          LA2:  abort
      t0=a-4                           Lok2: t5=[t4]
      tlen=[t0]                              t6=t5+1
      tcmp=i<t0                              [t2]=t6
      if tcmp goto Lok1 else LA1             i=i+1
LA1: abort                                   goto L0
```

?

```
      t0=a-4
      tn=[t0]
      t1=a
      t2=a+4
L0:   i=0
      tcmp=i<tn
      if tcmp goto Lend else L1
L1:   t3=[t2]
      [t1]=t3
      t2=t2+4
      t1=t1+4
      i=i+1
      goto L0
```

Loop-invariant hoisting

- *Idea:* move computations that always give the same result out of the loop: only compute once!
- Hoisting `a + b`: `a` and `b` must be *loop-invariant*:
  - constant,
  - only defined outside loop (use *reaching definitions*),
  - or only one definition inside loop whose expression is computed on loop-invariant variables
- Can identify all loop-invariant exprs (& dependencies) in one pass



Induction variables

- *Induction variables* are variables with value $A*i + B$ on the i^{th} iteration of a natural loop, for loop invariants A & B
- Several optimizations can exploit information about induction variables:
  - strength reduction
  - bounds-check elimination
  - loop unrolling

Identifying induction variables

- *Basic induction variables:* only one definition of the form $j = j + K$
- *Derived* (or *dependent*) *induction variables*: value is $j * M + N$ for some b.i.v. j (K, M, N loop invariants)

```
j = 3; z = 0;
for (i = 0; i < n; i++) {
  j = j + 1; z = z + 2;
  k = i*4 + 8;
  m = k*n;
  …
}
```

Strength reduction

- Derived induction variable k can be written as $A*i + B$, i some basic induction variable stepping by A_i
- For all distinct (A, B) pairs:
  - insert before loop header: $k = A*i + B$
  - insert after assignment to i: $k = k + (A*A_i)$
  - Replace definition of any k' whose formula is also $A*i + B$ by $k' = k$
- Effect: multiplication(s) replaced by single addition

$$t1=a+i*4 \Rightarrow t1=t1+A_i*4$$
$$M = k*n \Rightarrow M = M + t_M \quad (t_M = A_k*n)$$

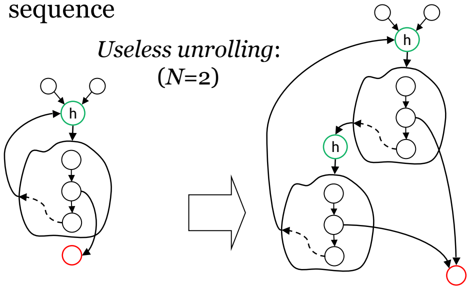
- Other optimizations facilitated: constant propagation, algebraic simplification, copy propagation, dead variable elimination, dead code elimination

---

Loop unrolling

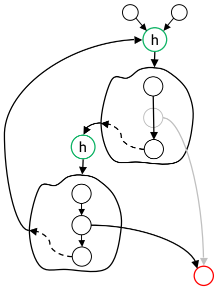
- Loop unrolling: creates N copies of loop in sequence

Useless unrolling:
(N=2)



Using induction variables

- **Idea:** use one loop test to ensure that entire unrolled loop (N copies) will succeed
- Loop test must depend on induction variable: *e.g.*, `i < n`
- `i+K*(N-1) < n` : no interior loop tests needed
- Additional loop needed to "finish up" `0..N-1` iterations
- Best if loop is small, straight-line code



Useful unrolling

Array bounds checks

- Iota+: On every expression `a[i]` , must ensure `i < length a, i ≥ 0` (`i <u length a`)
- Checking array bounds is expensive
- Array indices are often induction variables -- can use induction variable information to avoid the bounds check entirely!

```
for (int i=0; i<a.length; i++) {
      a[i] = a[i]+1;
}
```

two unnecessary bounds checks

Eliminating checks

- Given reference `a[k]` where `k` is an induction variable with value $a*i + b$: find a conditional test on some induction variable `j`
  - test terminates the loop
  - test dominates the reference to `a[k]`
  - test is against a loop-invariant expression that is ensures `k <u a.length`
- When to perform optimization?
  - **AST?** Need domination analysis, other optimizations not done.
  - **Quadruples?** Hard to recognize array length, array accesses, checks. **Solution:** propagate annotations

---

Null checks

- Java, Iota+ : need null checks on every
  - field access or assignment (except on `this`)
  - method invocation (except on `this`)
  - array element access
  - string operation
- *Idea:* Once we've checked for null, shouldn't need to check again

---

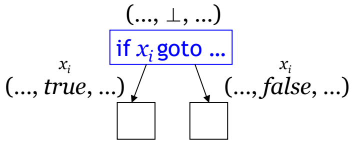
Example

u = p.x + p.y

```
⇒    t1 = p != 0                               t1 = p != 0
     if t1 goto L1 else L2                     if t1 goto L1 else L2
     L2: abort                                 L2: abort
     L1: ax = p + 4                            L1: ax = p+4
     tx = M[ax]                                tx = M[ax]
     t2 = p != 0          CSE: t2 = t1         t2 = t1
     if t2 goto L3 else L4                     goto L4      Copy: if t1 goto ...
                                                             Bool: t1 = true
     L3: abort                                 L3: abort
     L4: ay = p + 8                            L4: ay = p + 8
     ty = M[ay]                                ty = M[ay]
     u = tx + ty                               u = tx + ty
```

Boolean propagation

- Augment constant propagation with special propagation of booleans
- *Almost* fits into standard dataflow analysis model
- Different information leaves on out-edges of if quadruples



Finishing optimization

```
t1 = p != 0                 t1 = p != 0
if t1 goto L1 else L2       if t1 goto L1 else L2
L2: abort                   L2: abort
L1: ax = p+4                L1: ax = p+4
tx = M[ax]                  tx = M[ax]
t2 = t1
goto L4
L3: abort
L4: ay = p + 8               ay = p + 8
ty = M[ay]                  ty = M[ay]
u = tx + ty                 u = tx + ty
```

u = p.x + p.y