CS 412/413

Introduction to
Compilers and Translators

Andrew Myers
Cornell University

Lecture 29: More optimizations
10 April 00

Outline

- Loop optimizations
  - –Loop-invariant code motion
  - –Strength reduction
  - –Loop unrolling
  - –Array bounds checks
- Eliminating null checks
- Alias analysis
- Incremental dataflow analysis

Dominator data-flow analysis

- **A** dom **B** if **B** is reachable only by going through **A**.
- Forward analysis; *out*[*n*] is set of nodes dominating n
- "**A** dom **B** only if **A** dominates *all* predecessors of **B**"

L = sets of nodes ordered by $\subseteq$

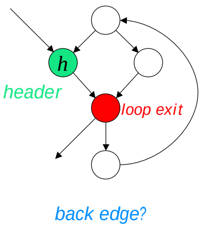
$\sqcap = \cap$

$\top = \{\text{all } n\}$

$F_n(x) = x \cup \{n\}$

Dominators and loops

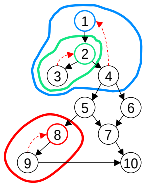
- Defn of loop: set of strongly-connected nodes with single entry point: *loop header* node
- *loop header* dominates all other nodes in loop
- Loop must contain *back edge* w/ respect to dominance relationship: $n \rightarrow h$ where h dom n



Completing control-flow analysis

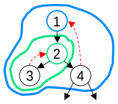
- Dominator analysis gives all *back edges*
- Each back edge $n \rightarrow h$ has an associated *natural loop* with h as its header: all nodes reachable from h that reach n without going through h
- For each back edge, can find its natural loop:

$\{n' \mid n' \text{ reachable from } h\} \cap$
$\{n' \mid n \text{ reachable from } n' \text{ in } G\text{-}h\}$

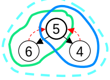


Control tree

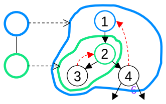
- Nest loops based on subset relationship between natural loops



- 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;
}
```

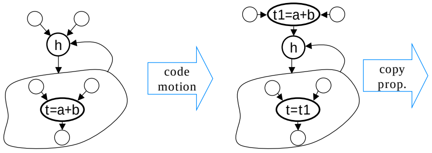
```
     i=0
L0:  t0=a-4
     tlen=[t0]
     tcmp=i<t0
     if tcmp goto Lend else L1
L1:  t1=i*4
     t2=a+t1
     t0=a-4
     tlen=[t0]
     tcmp=i<t0
     if tcmp goto Lok1 else LA1
LA1: abort
```

```
Lok1:t3=i*4
     t4=a+t3
     t0=a-4
     tlen=[t0]
     tcmp=i<t0
     if tcmp go
LA2: abort
Lok2: t5=[t4]
     t6=t5+1
     [t2]=t6
     i=i+1
     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 or b must be
  - constant,
  - only defined outside loop (use *reaching definitions*),
  - or only one definition inside loop whose expression is loop-invariant
- Can identify all loop-invariant exprs in one pass



Induction variables

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

Identifying induction variables

- *Basic induction variables:* only one definition of the form i = i + K
- *Derived induction variables***:** value is i * M + N for some b.i.v. i

```
j = 3;
for (i = 0; i < n; i++) {
  j = j + 1;
  k = i*4 + 8;
  m = k*12 + j*2;
  …
}
```

Strength reduction

- Every derived induction variable k can be written as $a*i + b$, a and b constants, i some basic induction variable
- For all distinct (a, b) pairs:
  - insert before loop header $k' = b$
  - insert after loop header $k' = k' + a$
  - Replace definition of any k whose formula is $a*i + b$ with $k = k'$
- Result: multiplication(s) replaced by single addition

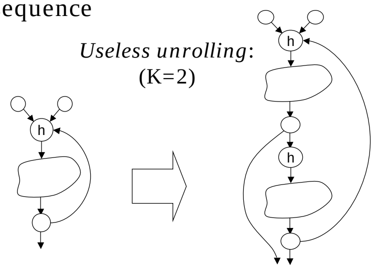
  t1=i*4 ⇒ t1=t1+4

- Additional optimizations facilitated: copy/constant propagation, dead variable elimination, dead code elimination

Loop unrolling

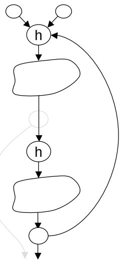
- Loop unrolling: creates K copies of loop in sequence

Useless unrolling:
(K=2)



Using induction variables

- When loop test expression depends on induction variable (e.g. i < n), can use one loop test to ensure that entire unrolled loop will succeed (i+K-1 < n): remove all interior loop tests
- Additional loop is needed to "finish up" 0..K-1 iterations



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!
- Can eliminate the bounds check if we can prove at compile time that it will always succeed

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

two unnecessary bounds checks

Rules

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

Null checks

- Another costly operation: checking for null pointers
- Java, Iota+ : needed 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

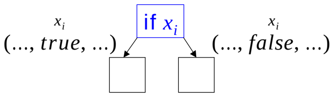
Example

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     Copy: if t1 goto …
    if t2 goto L3 else L4                    goto L4     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 propagation of booleans
- *Almost* fits into standard dataflow analysis model
  - different information leaves on different out-edges of if statements



Finishing optimization

```
t1 = p != 0               t1 = p != 0               CJUMP p != 0, L1
if t1 goto L1 else L2     if t1 goto L1 else L2     ABORT
L2: abort                 L2: abort                 L1: MOVE(u, M[p+4]
L1: ax = p+4              L1: ax = p+4                         + M[p+8])
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

Memory accesses

- Memory operations are expensive, confuse optimizer
  - want to use CSE to eliminate extra reads whenever possible
  - converting [t] to temporary makes many optimizations more effective
- Problem: *kill*[*n*] for statement [a] = b:

	gen	*kill*
[a]=b	b	[t] **(for all t that might be equal to a)**

Aliasing

- Problem: don’t know when two memory operands might refer to same location (*alias* one another)
- Flow-insensitive alias analysis: “*x* may alias *y*”
- Flow-sensitive alias analysis: “*x* may alias *y* at program point (flowgraph edge) *p*”
- Key: exploit high-level language knowledge
  - stack and heap locations cannot be aliases
  - objects of unrelated types cannot be aliases
- Alias analysis: for each node, variable x, determine which things [x] might alias