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 data flow 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 ⊆
  n = ∩
  T = {all n}
  F[x](x) = x ∪ {n}

Completing control-flow analysis
- Dominator analysis gives 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, can find its natural loop:
  {n' | n' reachable from h} ∩ {n' | n reachable from n' in G-h}

Dominator analysis gives 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, can find its natural loop:
{n' | n' reachable from h} ∩ {n' | n reachable from n' in G-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;
}

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

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
  \[ t1 = t1 + 4 \]
- Additional optimizations facilitated: copy/constant propagation, dead variable elimination, dead code elimination

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

Loop unrolling

- Loop unrolling: creates K copies of loop in sequence
  \[ \text{Useless unrolling: (K=2)} \]
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

Array bounds checks

- Iota+: On every expression \(a[i]\), must ensure \(i < length\ a, i \geq 0\) \((i < length\ a)\)
- 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

Example

\[
u = p.x + p.y
\]

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

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 \(k < a.length\)
- 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

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

\[
\begin{align*}
x_i & \quad \text{if } x_i \\
\ldots, \text{true, } \ldots & \quad \text{if } \ldots
\end{align*}
\]
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
  ty = M[ay]
  u = ax + 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