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: \( n \rightarrow h \) where \( h \ dom n \)

Completing control-flow analysis

- Dominator analysis identifies 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 \( n \rightarrow h \), find its natural loop:
  \( \{ n' | n' \text{ reachable from } n' \text{ in } G-h \} \cup \{ h \} \)

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: i=0
L1: i=1
L2: i=2
L3: i=3
L4: i=4
L5: i=5
L6: i=6
L7: i=7
L8: i=8
L9: i=9
Goto L0
Lok1: i=10
Lok2: i=11
Lok3: i=12
Lok4: i=13
Lok5: i=14
Lok6: i=15
Lok7: i=16
Lok8: i=17
Lok9: i=18
Goto L0

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

i=0
L0: i=0
L1: i=1
L2: i=2
L3: i=3
L4: i=4
L5: i=5
L6: i=6
L7: i=7
L8: i=8
L9: i=9
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

Example:
```plaintext
ht = a + b
ht1 = a + b
t = t1
```

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)

```plaintext
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 $A$
- For all distinct $(A, B)$ pairs:
  - insert before loop header: $k = A * i + B$
  - insert after assignment to $i$: $k = k + (A * i_A)$
  - Replace definition of any $k'$ whose formula is also $A * i + B$ by $k' = k$
- Effect: multiplication(s) replaced by single addition

```plaintext
t1 = a + i*4
M = k*n
```

Loop unrolling

- Loop unrolling: creates $N$ copies of loop in sequence

Useless unrolling: ($N=2$)

```plaintext
Useless unrolling:
```

Useful unrolling

```plaintext
Useful unrolling:
```

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
Array bounds checks

- Iota*: On every expression a[i], must ensure i < length a, i ≥ 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!

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

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 < length a
- 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

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

Example

\[
\begin{align*}
    u &= p.x + p.y \\
    t1 &= p \neq 0 \\
    t1 &= p \neq 0 \\
    \text{if } t1 \text{ goto L1 else L2} & \text{ if } t1 \text{ goto L1 else L2} \\
    \text{L2: abort} & \text{ abort} \\
    \text{L1: ax = p + 4} & \text{ ax = p + 4} \\
    tx &= M[ax] & tx &= M[ax] \\
    t2 &= p \neq 0 & \text{CSE: } t2 = t1 & t2 &= t1 \\
    \text{if } t2 \text{ goto L3 else L4} & \text{ goto L4} \\
    \text{L3: abort} & \text{ abort} \\
    \text{L4: ay = p + 8} & \text{ ay = p + 8} \\
    ty &= M[ay] & ty &= M[ay] \\
    u &= tx + ty & u &= tx + ty
\end{align*}
\]