Assignment 11 (last *programming* assignment)

Loop optimizations

Assignment 11
Final (necessary) step in compilation!

- Implement register allocation
- Basic simplify-select only
- No spill/split

Two functions in IRFunction.java
- simplifySelect
- chooseColor
Review of simplifySelect
Uses interference graph

Find node with degree < #regs
Know it can be colored
Remove node from graph
Recurse
Until graph is empty
Choose colors (regs) on way back up,
   based on adjacency colorings

Don’t actually remove nodes
   Too hard to remove nodes and edges
   Too hard to replace nodes and edges

Maintain HashSet of “removed” nodes in
   Ignore nodes in removed set
   Ignore edges connecting to removed nodes

Maintain HashMap for coloring
   Keys are temps, vals are regs

Other function is chooseColor
   Takes a node
   Picks a color unassigned to neighbor
   Must ignore “removed” nodes

Start with all regs
Remove reg for each neighbor
Select valid reg for temp
   IRReg.validForTemp(IRTemp) function
Extra credit option:

Consider move in chooseColor
If choosing color for temp t, search list of moves for move m containing t
If m has register r on other side of move, choose r as color for t
Provided r is not the color of any neighbors of t
Assume moves are ordered according to “payoff” (degree of loop nesting)
Choose first possible match

Loop optimizations

Big efficiency wins for the effort
Programs generally spend large portion of time in loops

Two major categories of optimizations
1. Loop invariant code hoisting
2. Loop induction variable optimizations

Both need support from prior analyses
Control flow: dominators, loops
Data flow: reaching definitions

Defining loop optimizations requires definition of terminology

We first introduce terminology, before studying optimizations

Looking ahead, optimizations based on computing same value only once
Loops: formal definition

Include set of basic blocks or instrs S

Loop has header h in S such that
1. Can reach h from any in S following CFG
2. Can reach any in S from h following CFG
3. No CFG edge leads to S other than h
   unless edge starts from S

1+2 define cycle
3 says says single entrance to cycle

Example

basic loop $S = \{ A, B, C, D, E \}$
header = A

Not all cycles are loops
cfg is reducible iff every cycle is a loop
In other words, loops contain no entry
points into middle of cycles

Cycle B,C is not reducible
NB: goto-less programs always reducible
Node A dominates node B in CFG iff every path from start to B includes A
cannot reach B with executing A
every node dominates itself
A dominates B, C, D
C dominates D
no other dominators

Every node n (except start) has node D s.t.:
\( n \rightarrow^* D \)
D dominates n
D does not dominate any node X that dominates n
(except n and D)
D is the immediate dominator of n
Informative to build dominator tree
Edge to each node from its immediate dominator
For any node \( n \), \( \text{dom}(n) \) denotes the set of dominators of \( n \).

Algorithm to find dominators

Like dataflow, based on equations:

\[
\text{dom}(\text{start}) = \{ \text{start} \}
\]

\[
\text{dom}(n) = \{ n \} \cup \{ \bigcap_{p \in \text{pred}(n)} \text{dom}(p) \}
\]

Algorithm for dominator eqn solns another iterative fixed-point algorithm:

\[
\text{dom}(\text{start}) = \{ \text{start} \}
\]

foreach node \( n \) in \( \text{cfg} \)

if \( n \neq \text{start} \)

\[
\text{dom}(n) = \text{nodes}(\text{cfg})
\]

while changed

foreach node \( n \) in \( \text{cfg} \)

\[
\text{dom}(n) = \{ n \} \cup \{ \bigcap_{p \in \text{pred}(n)} \text{dom}(p) \}
\]

Cheapest computed forward

Back edge is edge in \( \text{cfg} \)

From \( n \) to \( h \) where \( h \) dominates \( n \)

Every back edge defines a loop

\( h \) is called the header of the loop

Natural loop contains every node \( x \)

\( x \) is dominated by \( h \)

\( \text{cfg} \) path from \( x \) to \( n \) not including \( h \)
Loop L1 is nested inside loop L2 if
header of L2 dominates header of L1
(could be same header)

nodes of L1 proper subset of L2

Loop A,B nested inside A,B,C

Optimizing inner loops favored
Not all overlapping loops nest
Some loops with shared header may or may not nest

\[
\begin{align*}
A \quad & \text{A,C nested in A,B?} \\
B & \quad \text{A,B nested in A,C?}
\end{align*}
\]

Forms unnatural loop

Conditional branching generates unnatural loops

\[
\begin{align*}
& \text{while (i < 10)} \\
& \quad \{ \\
& \quad \quad \text{i++;} \\
& \quad \quad \text{if (P)} \\
& \quad \quad \quad \{ \text{LOOP1; continue;} \} \\
& \quad \quad \text{LOOP2;} \\
& \quad \}
\end{align*}
\]

Reaching definitions
Additional dataflow analysis
Similar to liveness analysis

What defs *must* reach a usage?
Definition is pair
- Storage location (variable/reg)
- Program location (stmt/instr)
Same def must reach
Regardless of path taken
Only consider defs from all preds
New def kills any incoming defs
Kill set
   Kills any previous definition of variable in def
New data flow equations to solve:
\[ \text{in}(n) = \bigcap_{p \in \text{pred}(n)} \text{out}(p) \]
\[ \text{out}(n) = \text{defs}(n) \cup (\text{in}(n) - \text{kill}(n)) \]
Use iterative fixed-point algorithm
Looking for greatest solution

Basic block analysis easier
   Reaching def trivial within basic block

Reaching def = previous assignment within block

No reaching def if no assignment
Why are reaching defs relevant?

Loop invariant
   Every reaching def outside loop
Common sub-expression elim
   need same reaching defs
Induction variables
   (like ++ in for loop)
   need simple reaching def within loop

Loop optimization
Two important cases
   Invariant hoisting
      Induction variable optimization

Both reduces work in loop
   Moves to before loop.
   And/or converts to cheaper work

First up: invariant hoisting

Loop-invariant conditions

loc1 = loc2 is loop invariant if:
   - loc2 is constant
   - loc2 never assigned to inside loop
   - loc2 has 1 reaching def from loop and
     source of that is invariant
loc1 = loc2 op loc3 is loop-invariant if
op is side-effect free and:
   - loc2,loc3 each are constant
   - loc2, loc3 are never assigned to in loop
   - loc2, loc3 have 1 reaching invariant def
When control flow recognizes loop
adds preheader to support optimization
New empty block dominates head

Put loop setup code in preheader

hoist invariants : loop to pre-header

\begin{verbatim}
L1:
    i++;  
    t = a + b;
    x[i] = t;
    if (i < N) goto L1
L2:
    y = t;
\end{verbatim}

c invariant: assigned constant
a invariant: assigned from c
b invariant: never assigned to in loop
t invariant: + side-effect-free, a, b invariant
\[c = t;\]
\[L1:\]
\[\text{i++;}\]
\[a = c;\]
\[t = a + b;\]
\[c = 0;\]
\[\text{if (i < N) goto L2}\]
\[L2:\]

- **c invariant:** assigned constant
- **a NOT invariant:** 2 reaching defs of c
- **b invariant:** never assigned to in loop
- **t NOT invariant:** a not invariant

**Other considerations:**

**Cannot hoist if value is used after loop and might not be set in loop**

\[t = 0\]
\[L1:\]
\[\text{if (i >= N) goto L2}\]
\[\text{i++;}\]
\[t = a + b;\]
\[x[i] = t;\]
\[\text{goto L1}\]
\[L2:\]
\[y = t;\]

Hoisting t may change y at L2

**If value reassigned in loop, cannot hoist safely**

\[t = 0\]
\[L1:\]
\[\text{i++;}\]
\[t = a + b;\]
\[x[i] = t;\]
\[t = 0;\]
\[\text{if (i < N) goto L2}\]
\[L2:\]
\[y = t;\]

Hoisting t changes \(x[i]\)
If used in loop before def, cannot hoist

L1:
\[ t = 0 \]
\[ i = i + 1 \]
\[ x[i] = t \]
\[ t = a + b \]
if \( i < N \) goto L2
L2:
\[ y = t \]

First element in x will be wrong if t hoisted

Precise invariant hoisting requirements
Can hoist stmt/instr s assigning t if assignment is loop-invariant and:

1) \( s \) dominates all loop exits for which \( t \) is live-out
2) only one def of \( t \) in loop
3) \( t \) is not live-out of pre-header

1) \( s \) dominates all loop exits for which \( t \) is live-out

Ensures that all uses (before kill) of \( t \) outside of loop flow through \( s \)
2) only one def of t in loop

Ensures that hoisting won’t change assignment results via reordering

3) t is not live-out of pre-header

Ensures that hoisting won’t impose spurious redefinition
In case loop is not executed