Lecture 17: Loop Optimizations

I. Loops
   A. Loop basics
      1. Why bother with loops as a special case
         a. What might we want to do to speed up a loop
         b. What would you do?
      2. Loop definitions
         a. A loop is a set of blocks, L, such that if B0, B1 are in L then there is a path in L from B0 to B1 and a path in L from B1 to B0.
         b. Where have we seen a definition like this?
         c. A block B in L is an entry block if B has a predecessor that is not in L.
         d. A block B in L is an exit block if B has a successor that is not in L
      3. Loop detection
         a. Done using information from depth-first search
         b. A block B can be in a loop only if it is on a path that includes a back edge to itself or a prior node in some depth first search walk of the graph
            i. If it has a back edge, it is an entry block
         c. We find loops by walking the loop backward when we find a back edge
      4. We want to compute the set of loops and their relationships
         a. These form a tree (why)
         b. Leaves of the tree are blocks in the flow graph
         c. Loops are the interior nodes of the tree
         d. The root is a special node representing the whole loop tree
   B. Compute the loop tree
      1. Basic Idea:
         a. Find loop associated with each block
            i. Using backedges to find potential loops
            ii. Tracing through the loops to find nodes in the loop
               ➢ Essentially strongly-connected components
Using DFS ordering to build a tree of nested loops

b. Result
   i. For each block B
      ➢ loop_entry[B] = entry block
      ➢ loop_contains[B] = Set of blocks in the loop
      ➢ loop_root = Root of the tree
      ➢ loop_parent[B] = entry for loop containing B, null if entry

c. Specific code (below) is not needed in lecture

2. Top level code
   procedure computeLoopTree
      foreach block B do
         loop_entry[B] = B
         loop_contains[B] = { B }
      next
      for each Block B in a post-order DFS traversal do
         findLoop(B)
      next
      BlockSet bset = { exit_block }
      loop_root = findLoopBody(bset,start_block)
   end procedure

   procedure findLoop(Block B)
      BlockSet loop = emptyset
      foreach edge E going into B do
         if E is a backedge then
            Block P = from block of E
            if P != B and P is not in loop
               add P to loop
            fi
         fi
      next
      findLoopBody(loop,B)
   end procedure

3. Finding a loop body given head and generators
   procedure findLoopBody(BlockSet gen,Block head)
      BlockSet loop = empty
      BlockList queue = empty
      foreach block B in gen do
         Block L = loopAncestor(B)
         if L is not in loop then
            add L to loop
            push L onto queue
         fi
      next
      while queue is not empty do
         pop Block B off of queue
         foreach predecessor P of loop_entry[B] to
            if P != head then
               add P to loop
            fi
         next
Block \( L = \text{loopAncestor}(P) \)
If \( L \) is not in the loop then
   Add \( L \) to loop
   push \( L \) onto queue
endif
endif
next
next

Add head to loop
create a new loop block \( X \)
\( \text{loop}_{-}\text{contains}[X] = \text{loop} \)
\( \text{loop}_{-}\text{entry}[X] = \text{head} \)
\( \text{loop}_{-}\text{parent}[X] = \text{null} \)
foreach block \( B \) in loop to
   \( \text{loop}_{-}\text{parent}[B] = X \)
next

4. Find the loop ancestor
   procedure \text{loopAncestor}(\text{Block} \ B)
      while \( \text{loop}_{-}\text{parent}[B] \) != null do
         \( B = \text{loop}_{-}\text{parent}[B] \)
      next
   return \( B \)

5. Example (from handout)
   a. Draw flow graph and use it to find loops

C. What should we be looking at in terms of optimizing loops
   1. Loop invariants – things computed in a loop that don’t change
   2. Induction variables – things that change by a predictable amount
   3. More aggressive optimizations in general

D. Want to identify for each loop, a loop prefix block.
   1. This is a unique block that precedes the entry to a loop
      a. Predecessor from a block not in the loop or an inner loop
   2. Might need to be created explicitly

II. Loop Optimizations
   A. Loop invariants
      1. A temporary \( T \) is \textit{loop invariant} in a loop \( L \) if it is either not computed in
         the loop or if all its operands are loop invariants.
      2. Compute \( \text{variant}[T] = \text{innermost loops in which} \ T \ \text{is not loop invariant} \)
         a. This is based on the instruction computing \( T \)
            i. Note this is unique because of SSA
         b. Why should it differ by instruction?
            i. Location of the instruction: is this sufficient
            ii. Operator of the instruction: why might this matter
c. Normal operators: variant is innermost loop where T is computed

d. LOAD: variant is innermost loop of any corresponding store

e. $\Phi$: what should we do here
    i. Things get somewhat messy (need to know input block)
    ii. Just assume things change here (variant = current loop)

3. Compute this by looking at the blocks
   a. In preorder walk of dominator tree

4. Then loop invariant computations can be moved out of the loop into the
   loop prefix
   a. We need to add a block at the entry to a loop
   b. More complex with multiple-entry loops
      i. These are often ignored by compilers
   c. These can be generalized however and handled as a subcase of a
      more general case

B. Induction Variables

1. Temporaries that are incremented by a predictable amount each time
   through the loop are called *induction temporaries*.

2. Limited operations can be applied
   a. $T = T_i + T_j$ where $T_i$ is induction Temporary and $T_j$ is invariant
   b. $T = T_i$ where $T_i$ is induction temporary
   c. $T = -T_i$ where $T_i$ is induction temporary
   d. $T = T_j$ where $T_j$ is invariant
   e. $T = \Phi(T_1, \ldots T_m)$ where all are induction or invariant

3. Candidates can be found by assuming everything is induction and then
   pruning down
   a. Worklist algorithm
   b. use $\text{use} \_\text{set}[T], \text{def} \_\text{set}[T]$ computed during SSA conversion

4. Candidates are more general than we actually want
   a. May not be predictable from one iteration to the next
   b. These are pruned by looking for relationships among the temporaries
      in strongly connected regions of the flow graph

5. Once we find induction temporaries, we replace their computation with
   simple additions in the loop along with an initial setting in the loop
   prefix
   a. We can combine this with loop invariant computations
   b. Note that loop invariants are also induction variables

C. Reshaping expressions
1. Detecting actual induction variables and loop invariants can be tricky
   a. Want to take into account associativity, distributivity
   b. Expressions may be split over loops
      i. Partially invariant in a nested loop
      ii. Partially invariant in an outer loop
      iii. Think of A[i,j]

2. Rewrite expressions for loop invariants
   a. $E = E' + (LC1 + (LC2 + (LC3 + ... + LCn)))$
   b. $E'$ is not loop invariant in the innermost loop
   c. LC1 is loop invariant in the innermost loop
   d. LC2 is loop invariant in the next-outer loop
   e. And so forth up the loop tree

3. Rewrite expressions for induction variables
   a. $E = E' + FD1*I1 + FD2*I2 + ... + FDm*Im$
   b. FD1 is loop-invariant and I1 is an induction variable in the inner most loop
   c. FDj, Ij are invariant and induction variables in the jth loop

4. Doing these rewrites lets you find more loop invariants and induction variables that can be optimized

D. Strength reduction
1. Take the expression $E$ and compute it before entering the loop
2. Rather than recomputing it each time through the loop, just update it
   a. The loop invariant parts are constant
   b. The induction variables provide increments
   c. Remove multiplications
3. Especially useful for array bounds A[i,j]

E. Loop rolling and unrolling
1. Unrolling: expand the loop to do multiple iterations at once
   a. Why might this be helpful?
      i. Instruction scheduling
      ii. Eliminates condition check
      iii. Allows more code overlap
      iv. Allows more local optimization (combines blocks)
      v. Allows more common subexpression elimination
2. Rolling: combining two loops into one
   a. Why might this be useful?

F. Summary
1. Find the innermost loop where \( T \) does not vary for each \( T \)
2. This lets you identify what is loop invariant for each loop
3. And what is a potential induction variable
4. Rewrite expressions taking these into account
5. Then move the computations around
   a. Loop invariants are moved into loop prefix block
   b. Induction variables are split into an initial computation \( E' \), and then increments in each of the loops

III. Other dominator-based optimizations
   A. Coalescing temporaries
      1. We want to have a minimum number of temporaries
      2. Form equivalent classes based on copy instructions
      3. Rename into equivalence classes
      4. This eliminates move instructions and simplifies later register allocation
      5. This will be done again as part of register allocation
   B. General dead code elimination
      1. Take branches used into account
      2. Remove unused blocks as well as unused instructions
   C. Better value numbering
      1. We did value numbering statically
         a. Conservative approximation with \( \Phi \) operators
      2. Taking branch conditions into account
      3. A better approximation can be achieved by doing it iteratively

IV. Other optimizations
   A. Move loads and store
      1. Move loads as early as possible to allow memory fetch to overlap execution
      2. Move stores as late as possible (outside of loops)