CSCI 1260: Compilers and Program Analysis
Steven Reiss
Fall 2017

Lecture 14: Optimization Data Structures

I. Optimization makes use of a variety of specialized data structures
   A. Store information
      1. About each routine
      2. About each basic block
      3. About each temporary
   C. Basic concepts
      1. What information needs to be computed
      2. How we can efficiently store that information
      3. How we can efficiently compute that information

II. Sets
   A. Throughout optimization we will use sets (and maps)
      1. Sets of temporaries
      2. Sets of basic blocks
      3. Sets of temporaries for each basic block
      4. Sets of basic blocks for each temporary
      5. Sets of basic blocks for each blocks
      6. Set of temporaries for each temporary
      7. Sets of registers
      8. Note that all these items can be represented as small integers
   B. Operations on these sets
      1. Frequent: insert, delete, check, iterate
      2. Other: union, intersection, complement, copy
   C. Size of the sets
      1. Up to several thousands (potentially more)
         a. For temporaries and blocks
         b. Generally in the hundreds
      2. Register sets are smaller
      3. Note that set sizes change during optimization
         a. Optimizations can create new temporaries or blocks
         b. Might want to have dynamic sets
c. Alternatively might want to bound the sizes

D. Representation of basic sets

1. Bit sets
   a. Nice and compact (if dense)
   b. How to iterate over a bit set
   c. Is this the only representation?

2. Two array representation
   a. Have a value array and an index array [0..max]
      i. And a counter of the number of elements (num)
   b. Insert
      i. Put the new value at the end of the value array
      ii. Index[value] = its index in the value array
   c. Check
      i. Look at index[value]
      ii. It must be in the range (0..num-1)
      iii. And value[index[value]] == value
   d. Iterate
      i. Just loop over elements in Value
   e. Delete
      i. Set value[index[value]] = value[num-1]
      ii. Index[value[num-1]] = its new index
      iii. Decrement num
   f. Union, intersection, complement are more difficult

3. Tree-based implementations

4. Hash-based implementations

5. Which to use
   a. Bits sets are easiest with some work on iterate
   b. In a real compiler, different representations would be used in different places
      i. Depending on particular operations
      ii. Expected sizes and sparseness
      iii. Efficiency considerations

E. Representation: Sets of sets

1. Arrays of sets
   a. Good if the top index is dense
   b. Easiest to use

2. Maps of sets
a. Useful if sets are less dense
3. Again in a real compiler, you’d choose different ones for different occasions

F. Indexing of sets
1. We’ll provide mechanisms to allow sets to be indexed by temporary or block directly
2. Or using temporary or block index
3. Also provide mapping of index to temporary or block

III. Basic data structures for optimization
A. There are two basic data structures that are used
1. The sequence of instructions inside a basic block
   a. Represented as a list
   b. These can be gone over sequentially in either direction
2. The set of basic blocks
   a. These are linked together via edges
   b. These form a graph
   c. We thus need to worry about graphs

B. Simple versus complex graphs
1. Most programs have reducible graphs
   a. These result from structured programming (no gotos)
   b. These make optimization much easier
      i. Algorithms become linear
      ii. Graphs can be considered in a structured way
2. Reducible graphs do not contain the subgraph
   a. A->B, A->C, B->C, C->B
   b. Either directly or induced
   c. Reducible graphs can be reduced to a single node
      i. Remove singletons
      ii. Remove ifs that coalesce
      iii. Remove self-loops
3. Connected graphs
   a. Every node needs to be reachable from the start node
   b. Every node needs to reach the end node
   c. Strongly connected components
      i. There is a path from any node in the component to any other
      ii. These are useful in some situations

IV. Depth-First search
A. We need to be able to traverse the graphs
   1. To iterate over the blocks
   2. To optimize the blocks
   3. Want to do this in a logical manner
      a. Typically we want to do the predecessors of a block before the block
         i. As much as possible
      b. Alternatively might want to do successors
B. The most logical approach is depth-first search
   1. Basic idea
      a. Start with the root
      b. For each node, visit recursively each child in turn
      c. If a node has already been visited before, ignore it
      d. Keep track of the order the nodes are visited in
         i. This is the depth-first order
   2. What we want to get from the DFS
      a. Reorder the flow blocks to be in depth first order
         i. This provides an ordered way of looking at the blocks
      b. Classify the edges according to how they are used in the DFS
         i. This is just as important as getting a DF ordering
C. Edge types
   1. Tree edges: edges used in the DFS for new nodes
      a. These form a spanning tree
      b. These indicate the natural flow of control
   2. Back edges: edges that go from a node to a node that is in the middle of processing
      a. These indicate loops
   3. Forward edges: edges that go from a node to a node that has been processed in the current DFS
      a. These indicate branch arounds
      b. From ancestor to descendent
   4. Cross edges: edges that go from one subtree to another
      a. These are the problematic ones
      b. Should be rare or nonexistent in a reducible flow graph
D. Why bother using depth first search
   1. We’ll use the depth first order to our advantage in optimization
   2. We’ll use the edge types to find the looping structure of the program
      a. To facilitate global or loop-based optimizations
E. DFS in an optimizer (DECAF)

1. Done early in the optimization (might be repeated as things change)
2. Purpose
   a. Compute the depth-first ordering
   b. Assign a DfsType to each edge
   c. Reorders the blocks in the routine to be in DF order
   d. Fixes up all branches
      i. Removes BRANCH instructions to next blocks
      ii. Adds BRANCH instructions if blocks are not next

F. DFS Algorithm

```c
int pre [#BLOCKS] = 0
int post [#BLOCKS] = 0
int precount = 0;
int postcount = #BLOCKS + 1

function dfs(Block B)
    pre[B] = precount++
    foreach Edge E in Successors(B)
        Block S = E.getToBlock();
        if (pre[S] == 0)
            e is a TREE edge
dfs(S)
        else if (post[S] == 0)
            e is a BACK edge
dfs(S)
        else if pre[B] < pre[S]
            e is a FORWARD edge
        else
            e is a CROSS EDGE
    next
    post[B] = postcount--
```

G. Example: from handout 4

V. Domination

A. Background

1. DFS alone is not sufficient for optimization
2. What we want to do is ensure that if block A precedes block B, then all executions of go through block A before they go through block B
   a. This can be viewed in either the forward or the backward direction
3. But such an ordering is impossible (WHY)
   a. However, we can keep track of the partial ordering that is so induced
   b. And use the partial ordering for optimization
c. This is done by maintaining DOMINANCE RELATIONS

4. What is a partial ordering?
   a. Reflexive A <= A
   b. Antisymmetric A <= B and B <= A implies A == B
   c. Transitive A <= B and B <= C implies A <= C

5. Can also be non-strict partial order
   a. Irreflexive: NOT A < A
   b. Asymmetric: A < B implies NOT B < A
   c. Transitive A < B and B < C implies A < C

6. What is a total order

7. What is a lattice

B. Domination
   1. A block B1 dominates a block B2 iff every path from Entry to B2 contains B1
      a. B1 DOM B2
      b. Domination is reflexive (B1 DOM B1)
      c. Domination is transitive (B1 DOM B2 & B2 DOM B3 => B1 DOM B3)
      d. If B1 DOM B2 and B2 DOM B1, then B1 == B2
      e. Hence DOM is a (strict) partial order
   2. The dominators of a block can be ordered
      a. If B2 DOM B and B1 DOM B then either
         i. B1 == B2
         ii. B1 DOM B2
         iii. B2 DOM B1
      b. Thus there is a unique IMMEDIATE DOMINATOR for a block
      c. Denote this as IDOM(B)
      d. Immediate dominance forms a tree
      e. Children(B) are the nodes X such that IDOM(X) = B

C. Example (from handout 4)
   static public void sort(int [] A) {
      int i = 0;
      while (i < A.length) {
         int j = i+1;
         while (j < A.length) {
            if (A[j] < A[i]) {
               swap(A,i,j);
            }
         }
         j = j+1;
      }
D. Computing Dominance

1. Basic algorithm (fixed point excluding)
   Block R = startBlock
   BlockSet dset = emptyset
   BlockSet tset = emptyset
   BlockSet all = set of all blocks
   BlockSetByBlock domin = emptyset

   domin(R) = { R }
   foreach block B
     if ( B != R) then domin(B) = all
   next

   boolean change = true;
   while (change) {
     change = false;
     foreach block N
       if ( N != R ) then
         tset = all
         foreach predecessor of N, P
           tset = tset INTERSECT domin(P)
         tset = tset UNION { N }
         if ( tset != domin(N) ) then
           change = true
           domin(N) = tset
         fi
       fi
     next
   next

2. Computing immediate dominance
   Block R = startBlock
   BlockSetByBlock idom = emptyset
   BlockToBlockMap idommap

   foreach block N
     idom(N) = domin(N) - { N }
   next

   foreach block N
     bset = emptyset
if ( N != R ) then
    foreach block S in idom(N)
        foreach block T in idom(S)
            if ( T != S && T in idom(S) ) then
                bset = bset UNION { T }
            fi
        next
    fi
    idom(N) = idom(N) - bset
next

foreach block N
    if ( N != R ) then
        idommap[N] = any element of idom(N)
    fi
next

return idommap

3. Computing set of children
BlockSetByBlock children

foreach block B
    Block C = idom(B)
    if (C != null) children(C) += B
next

return children

4. Example
E. Dominance Frontier
1. We need to know the edges or limits of the dominance relation
2. The **DOMINANCE FRONTIER** of a block B, **DF**(B), is the set of all blocks C such that B dominates a processor of C but either B equals C or B does not dominate C.
   a. Represents the set of all block just outside the domination of B
3. Computing the dominance frontier
   BlockSetByBlock domfnt = emptyset
call blockFrontier(startBlock,domfnt)

function blockFrontier(Block B,
BlockSetByBlock domfnt)
  foreach Block C in children(B)
    call blockFrontier(C,domfnt)
  next
  foreach Block X in successors(B)
    if (idom(X) != B)
      domfnt[B] = domfnt[B] UNION {X}
    next
  self
  foreach block C in children(B)
    foreach block X in domfnt(C)
      if ( idom(X) != B )
        domfnt(B)= domfnt(B) UNION {X}
      fi
    next
  next
return