I. Handout 5
II. Normalizing the Flow Graph
   A. The first step in optimization is to ensure the flow graph is proper
      1. To allow us to perform the later optimizations
      2. We've already done some of this by reordering the graph in DFS order
   B. Optimization will edit the code
      - Creating new blocks between blocks
      - Inserting new instructions in blocks
      - Moving instructions between predecessor and successor
      - Inserting new instructions on edges
        i. At least logically. They then have to be placed in blocks
        ii. Need to ensure that edges are associated with blocks
   2. What block is associated with an edge
      - If the source of the edge has only one successor, then computations can be inserted at the end of the source block
      - If the target of the edge has only one processor, then computations can be inserted at the start of the target block
      - What do we do in other cases
   3. Critical edges
      - Consider BLOCK P -> X, P -> S, Y -> S
        i. How do we insert code on the edge P->S
        ii. This is called a CRITICAL EDGE
      - What can we do here?
        i. Insert a new (empty) block between P and S
   C. Other normalizations
      1. Remove any unreachable blocks
         - Blocks with no predecessors
         - Blocks whose only predecessors are NONE arcs
         - Do not remove the exit block
2. Ensure all blocks can reach exit
   • If a block has no outgoing edges, and NONE edge to exit block
   • Might have to handle loops with no exit as well
3. Identify abnormal edges
   • Branches where the compiler cannot detect the destination of a branch
   • This can occur for various reasons
     i setjmp/longjmp
     ii Exceptions (Throw statements)
     iii Subroutine calls with non-local branching (PL/1)
   • Have the compiler add all possible edges in these cases
     i Label the edges appropriately
D. Local Information
1. Consider what we did for local optimization
   • Redundant subexpression elimination
   • Constant folding
   • Dead code elimination
2. Recall how we did this
   • Going thru instructions in execution order
     i Checking for constants
     ii Checking for duplicate value numbers
   • Going thru instructions in reverse execution order
     i Tracking what was actually needed
3. Suppose we wanted to extend this to the whole routine
   • What information did we need to have
     i What value (value number) a temporary had at any point
     ii Whether a value was constant or not
     iii Whether a temporary was used after it was computed
   • We need to consider temporaries over the whole routine
4. The compiler needs to understand how temporaries are used
   • Where they are created
   • Where they are used
   • Where they have a valid value (available)
   • Where they may not hold a valid value (not available)
5. These all need to be computed
   • Flow analysis
   • Flow equations
• Flow computations
  i Define the changes that occur in the block
  ii Compute the actual values at the start and end of block
     ➢ Incrementally based on previous computed values
  iii Iterate until nothing changes (fixed point algorithm)

E. Lifetime Analysis
1. A temporary $T$ is live at a point $p$ in the flow graph (program) if there is a path from $p$ to a use of $T$ as an operand that does not contain an instruction evaluating $T$ and a path from an evaluation of $T$ to $p$.
   • Why is this needed for extending local optimizations
2. $\text{LIVEUSE}(B)$ = set of temporaries $T$ such that $T$ is used in block $B$ and whose use is not preceded by an evaluation of $T$
   • $\text{LIVEUSE}(T)$ = set of blocks that contain a use of the temporary $T$ that is not preceded by an evaluation of $T$
   • This may be used later on, but not needed right now
3. $\text{LIVEKILL}(B)$ = the set of temporaries $T$ for which there is an evaluation of $T$ in the block $B$
4. $\text{LIVEIN}(B)$ = the set of temporaries that are live at the start of block $B$
   • $\text{LIVEIN}(T)$ = set of blocks $B$ where the temporary is live at the start
5. $\text{LIVEOUT}(B)$ = the set of temporaries live at the end of block $B$
6. $\text{GLOBAL}$ = set of temporaries that are live at the beginning of any block
   • Union of $\text{LIVEIN}(B)$ over all $B$
7. Flow equations
   • $\text{LIVEIN}(B)$ = $\text{LIVEUSE}(B)$ + ($\text{LIVEOUT}(B)$ – $\text{LIVEKILL}(B)$)
   • $\text{LIVEOUT}(B)$ = UNION over successors $S$ of $B$ of ($\text{LIVEIN}(S)$)
   • Why does this work?
8. How would you compute the basic sets
   • $\text{LIVEUSE}(B)$, $\text{LIVEKILL}(B)$
   • Single pass over the instructions in the block in execution order
   • If $T$ is an operand and not in KILL, add to USE
   • If $T$ is the target, add to KILL
   • Example: Block9 in Handout5
     
     LDC #4,T11
     LDC #4,T11
     aADD T1,T11,T12
     iMUL T7,T11,T13
     aADD T12,T13,T14
     LDC #4,T11
     LDC #4,T11
• LIVEUSE = \{1,7,2\}
• LIVEKILL = \{11,12,13,14,15,16,17,18,19\}

9. Basic code

Compute LIVEUSE, LIVEKILL locally
Foreach Block B
  Set LIVEIN(B) = LIVEOUT(B) = emptyset
  Repeat until nothing changes
    Foreach Block B
      LIVEIN(B) = LIVEUSE(B) union (LIVEOUT(B) – LIVEKILL(B))
    next
    Foreach Block B
      LIVEOUT(B) = empty
      Foreach Successor S of B
        LIVEOUT(B) = LIVEOUT(B) union LIVEIN(S)
      next
    next

10. Example (from handout 5)
  • Justify the use of a worklist algorithm here

11. This is the general idea
  • How would you actually compute this
  • Can take account of the fact that each T is computed in and used in a limited set of blocks
  • Working partly by temporaries can be more efficient
  • But the code is a lot more complex

12. Even with the basic code, you don’t do everything for every block
  • Work list algorithm
  • Basic idea
    i. Keep a list of blocks to consider
    ii. When a LIVEIN for a block changes, add its predecessors to the list
    iii. When the list is empty you are done
    iv. Don’t add a block if it is already on the list
13. Other forms of data flow problems

- Intersection problems
  - Where one of the items is the intersection rather than the union
    - Done much the same way
    - Except start with All rather than the empty set

- Forward flow problems
  - Work on successors rather than predecessors

F. Anticipation Analysis

1. A lot of the information we need to know about the program to do optimization (or program analysis) can be defined and computed in a similar fashion.

2. Definitions

   - A temporary $T$ is **locally anticipated** in block $B$ if there is an instruction if $B$ that evaluates $T$ and there is no instruction preceding that instruction in $B$ that might kill $T$.
   - A temporary $T$ is **locally available** in block $B$ if there is an instruction in $B$ that evaluates $T$ and there is no instruction following that instruction in $B$ that might kill $T$.
   - A temporary $T$ is **transparent** in block $B$ if there is no instruction in $B$ that might kill $T$. If $T$ is not transparent in $B$ then $T$ is **killed** in $B$.

3. Anticipation can be used to move code around safely

   - If $T$ is anticipated in $B$, then the evaluation instruction for it can be moved to a prior block

4. What does it mean for a temporary to be killed (1st usage, not second)?

   - It is the target of some instruction (reevaluation)
   - It represents an expression and one of the subexpressions has its value changed by an instruction
     - Note that recomputing does not change the value, no need to change the value of an underlying variable
   - It represents a variable and an instruction does or has the potential to change that variable
   - If the variable is stored in memory, then any store into memory can change it
     - Restrict this as much as possible
       - By data type if the language is strongly typed
       - By interprocedural and alias analysis for calls
5. Killed is reflected in a precomputed set \texttt{modifies}(T)
   - The set of temporaries (and memory locations) that are killed by an instruction that has T as a target
   - Empty for expression temporaries
   - Defined for variable temporaries
   - We’ll assume it exists, put off definition for later

6. Example: Handout 5
   - Modifies:
   - Locally Anticipated:
   - Locally Available
   - Killed

7. Computing these locally for a block
   \begin{verbatim}
   computeAvailable(Block B)
   TemporarySet killed = empty
   TemporarySet anticipated = empty
   TemporarySet available = empty

   foreach Instruction I in execution order
     Temporary T = the target of I
     If ( T is defined )
       If ( !killed(T) && !anticipated(T) )
         anticipated = anticipated UNION \{ T \}
         ANTLOC(T) = ANTLOC(T) UNION \{ B \}
         ANTLOC(B) = ANTLOC(B) UNION \{ T \}
       fi
     available = available - modifies(T)
     available = available UNION \{ T \}
     killed = killed UNION modifies(T)
     fi
   next

   foreach Temporary T in available
     AVLOC(T) = AVLOC(T) UNION \{ B \}
   next

   KILL(B) = killed
   TemporarySet trans = SETCOMPLETMENT( killed )
   TRANSF(B) = TRANSF(B) UNION trans
   \end{verbatim}

8. Global Definitions
   - A temporary T is \texttt{anticipated} at a point P in the flow graph iff every path from P to Exit contains an instruction that evaluates T that is not preceded on that path by an instruction that might kill T
• ANTIN(B) : the set of temporaries that are anticipated at the start of a block B
• ANTOUT(B) : the set of temporaries that are anticipated at the end of a block B

9. Flow Equations
• ANTIN(B) = ANTLOC(B) union (ANTOUT(B) – KILL(B))
• ANTOUT(B) = empty if B is an exit
  i Intersection over successors S of B of ANTIN(S)
• Why do these work

10. These equations (and flow equations in general) can have multiple solutions
• Multiple solutions arise from loops
• We want to compute a fixed point (stable) solution
  i Actually we want the maximum fixed point
  ii With the sets as small as possible
• Some algorithms want a minimum fixed point

11. This can be computed using an iterative algorithm
• Basic idea
  i Start by assuming all the sets are full
  ii Iterate through each set, removing those elements that shouldn’t be there
  iii Repeat this until nothing changes
• How does this differ from what we did before

12. Basic code (using worklist)
  TempSetByBlock ANTIN
  TempSetByBlock ANTOUT
  BlockList worklist
  Foreach Block B
    ANTOUT(B) = all temps if B != exit, else emptyset
    ANTIN(B) = ANTLOC(B) union (ANTOUT(B) – KILL(B))
    Add B to worklist
  next
  While worklist not empty
    Take Block B off the worklist
    Compute ANTOUT(B) = intersection of ANTIN(S) for all succ(B)
    Compute ANTIN(B) = ANTLOC(B) union (ANTOUT(B) – KILL(B))
    If ANTIN(B) has changed then
Foreach P in pred(B) : Add P to worklist

fi
next

13. Can also be done using repeat until nothing changes

G. Reaching Definitions
1. REACHIN(B) = the set of definitions that reach the start of B
2. REACHOUT(B) = the set of definitions that reach the end of B
3. Local information
   • The set of temporaries used before they are killed
   • The set of definitions that are killed in a block
   • The set of definitions that reach the end of a block
4. Data flow equations
   • REACHIN(B) = Union { P in pred(B)} of REACHOUT(P)
   • REACHOUT(B) = GEN(B) union (REACHIN(B) – KILLS(B))
   • GEN(B) = definitions generated in B
   • KILL(B) = definitions killed by an instruction in B
5. Akin to reverse liveness
6. What could this be used for?
   • Think about doing the local optimizations we did over the whole routine
   • We need to have this information plus liveness information
7. This can get messy
   • So we will try to avoid computing it.