Code Optimizations

The intermediate code (e.g., IR tree) generated by the front-end is often not efficient.

The code optimizer reads IR, emits better IR; almost all optimizations done here are machine-independent. Machine-dependent optimizations are done in the back-end.

Main techniques used: graph algorithms, control- and data-flow analysis

Code Optimizations (cont’d)

Optimizations that are restricted to one basic block are called local-optimizations; otherwise, they are called global optimizations

Here are a partial list of well-known compiler optimizations:

algebraic optimizations (strength reduction, constant folding)
common-subexpression eliminations

Examples: Source Code

C code for quicksort (also in ASU page 588):

```c
1 void quicksort(m, n);
2 int n, n;
3 {
4   int i, j, v, x;
5   if (n <= n) return;
6   i = m-1; j = n; v = a[n];
7   while (1) {
8     do i = i+1; while ( a[i] < v);
9     do j = j-1; while ( a[j] > v);
10    if (i >= j) break;
11    x = a[i]; a[i] = a[j]; a[j] = x;
12  }
13  x = a[i]; a[i] = a[n]; a[n] = x;
14  quicksort(m, j); quicksort(i+1, n);
15 }
```
Example: Intermediate Code

- Intermediate code for the shaded fragments of previous example:

  (01) \[ i := m - 1 \]
  (02) \[ j := n \]
  (03) \[ c_1 := 4 \ast n \]
  (04) \[ v := a[c_1] \]
  (05) \[ i := i + 1 \]
  (06) \[ t_2 := 4 \ast i \]
  (07) \[ c_2 := a[c_2] \]
  (08) \[ \text{if } t_3 < v \text{ goto (5)} \]
  (09) \[ j := j - 1 \]
  (10) \[ t_4 := 4 \ast j \]
  (11) \[ t_5 := a[t_4] \]
  (12) \[ \text{if } t_5 > v \text{ goto (9)} \]
  (13) \[ \text{if } i \geq j \text{ goto (23)} \]
  (14) \[ t_6 := 4 \ast i \]
  (15) \[ x := a[t_6] \]
  (16) \[ t_7 := 4 \ast i \]
  (17) \[ t_8 := 4 \ast j \]
  (18) \[ t_9 := a[t_8] \]
  (19) \[ a[c_7] := t_9 \]
  (20) \[ t_{10} := 4 \ast j \]
  (21) \[ a[t_{10}] := x \]
  (22) \[ \text{goto (5)} \]
  (23) \[ t_{11} := 4 \ast i \]
  (24) \[ x := a[t_{11}] \]
  (25) \[ t_{12} := 4 \ast i \]
  (26) \[ t_{13} := 4 \ast n \]
  (27) \[ t_{14} := a[t_{13}] \]
  (28) \[ a[t_{12}] := t_{14} \]
  (29) \[ t_{15} := 4 \ast n \]
  (30) \[ a[t_{15}] := x \]

Control-Flow Analysis

- How to build the Control-Flow Graph (CFG)?
  - each basic block as node, each jump statement as edge.
  - there is always a root --- the “initial” node or the entry point.
- How to identify loops? and how to identify nested loops?
  1. build the dominator tree from the CFG
  2. find all the back edges; each back edge defines a natural loop
  3. keep finding the innermost loop and reduce it to a single node.
- Given a CFG G with the initial node (root) r, we say node d dominates node n, if every path from root r to n goes through d.
- Dominator tree is used to characterize the “dominate” relation: r as the root, the parent of a node is its immediate dominator. (see ASU page 602--608 for more details)

Data-Flow Analysis

- Data-Flow Analysis refers to a process in which the optimizer collects data-flow information at all the program points.
- Examples of interesting data-flow information:
  - reaching definitions: the set of definitions reaching a program point
  - available expressions: the set of expressions available at a point
  - live variables: the set of variables that are live at a point
- Program points: with each basic block, the point between two adjacent statements, or the point before the first statement and after the last. A path from point \( p_i \) to \( p_n \) is a sequence of points \( p_1, ..., p_n \) such that \( p_i \) and \( p_{i+1} \) are “adjacent” for all \( i = 1, ..., n-1 \).

Data-Flow Analysis (cont’d)

- For each statement \( S \), we associate it with four sets:
  - \( \text{in}[S] \): the set of data-flow info. associated with the point before \( S \)
  - \( \text{out}[S] \): the set of data-flow info. associated with the point after \( S \)
  - \( \text{gen}[S] \): the set of data-flow info. generated by \( S \)
  - \( \text{kill}[S] \): the set of data-flow info. destroyed by \( S \)
  - Naturally, if \( S_1 \) and \( S_2 \) are two “adjacent” statements within a basic block, say, \( S_2 \) immediately follows \( S_1 \) then \( \text{in}[S_2] = \text{out}[S_1] \)
  - We can define these four sets for each basic block \( B \) in the same way. The \( \text{gen} \) and \( \text{kill} \) sets of a basic block can be calculated from the corresponding values for each statement of that basic block.
- Forward-DataFlowProblem: the data-flow info. is calculated along the direction of control flow.
  - Backward-DataFlowProblem: the data-flow info. is calculated opposite to the direction of control flow.
Example: Reaching Definitions

- A definition \( d \) reaches a point \( p \) if there is a path from the point immediately following \( d \) to \( p \), such that \( d \) is not "killed" along that path.

- A definition of a variable \( v \) is "killed" between two points if there is a read of \( v \) or an assignment to \( v \) in between.

- Goal: given a program point \( p \), find out the set of definitions that might reach point \( p \). This is a forward data-flow problem:

```c
/* initialize out[B] assuming in[B] = Ø for all B */
while change do begin
  change := false;
  for each block B do begin
    in[B] := union of out[P] for all predecessor P of B;
    oldout := out[B];
    out[B] := gen[B] \ Ø in[B] - kill[B];
    if out[B] <> oldout then change := true
  end
end
```

Other Data-Flow Problems

- Use-Definition Chains: for each use of a variable \( v \), find out all the definitions that reach that use. (directly from reaching definitions info.)

- Available Expressions: an expression \( x + y \) is available at a point \( p \) if every path from the initial node to \( p \) evaluates \( x + y \), and after the last such evaluation prior to reaching \( p \), there are no subsequent assignments to \( x \) or \( y \). (this is a forward data-flow problem)

- Live-Variable Analysis: a variable \( x \) is live at point \( p \) if the value of \( x \) at \( p \) may be used along some path starting at \( p \). (this is a backward data-flow problem)

- Definition-Use Chains: for each program point \( p \), compute the set of uses \( s \) of a variable \( x \) such that there is a path from \( p \) to \( s \) that does not redefine \( x \). (backward data-flow problem)

Using Data-Flow Info.

- Common Subexpression Eliminations: a flow graph with available expression information. (ASU page 634)

For every statement \( s \) of the form \( x := y + z \) such that \( y+z \) is available at the beginning of \( s \)'s block, neither \( y \) nor \( z \) is defined prior to \( s \) in that block.

1. discover all the last evaluations of \( y+z \) that reach \( s \)'s block
2. create a new variable \( u \).
3. replace each statement \( w := y+z \) found in (1) by
   \[ u := y + z \]
   \[ w := u \]
4. replace statement \( s \) by \( x := u \)

Using Data-Flow Info. (cont’d)

- Copy Propagations: a flow graph plus the ud-chains and du-chains information, and also some copy-statement info. (see ASU page 638)

for each copy \( s : x := y \), determine all the uses of \( x \) that reached by this definition of \( x \), then for each use of \( x \), determine \( s \) is the only definitions that reaches this use, if so, replace the use of \( x \) with \( y \).

- Loop Invariants: a flow graph plus the ud-chains information

  a statement is a loop invariant if its operands are all constants, or its reaching definitions are loop invariants or from outside the loop.

  - For more examples, see the ASU section 10.7.
  - Challenges: what if there are procedure calls, pointer dereferencing ...? also, how to make these algorithms more efficient?
Static-Single Assignment

• **Motivation**: how to make data-flow analysis more efficient & powerful?

• **Static-Single Assignment (SSA) form*** --- an extension of CFG:

  v := 4  
  z := v + 5  
  v := 6  
  y := v + 7  

  if P then v := 4  
  else v := 6  
  u = v + y

  SSA transformation

  v1 := 4  
  z := v1 + 5  
  v2 := 6  
  y := v2 + 7

  if P then v3 := 4  
  else v4 := 6  
  v5 = \phi(v3,v4)  
  u = v5 + y

• **Main idea #1**: each assignment to a variable is given a unique name, and all of the uses reached by that assignment are renamed to match the assignment’s new name.

Static-Single Assignment (cont’d)

• **Main idea #2**: after each branch-join node, a special form of assignment called a \( \phi \)-function is inserted. \( \phi(v_1,v_2,...,v_n) \) means that if the runtime execution comes from the \( i \)-th predecessor, then the above \( \phi \)-function returns the value of \( v_i \).

• **Why SSA is good**? SSA significantly simplifies the representation of many kinds of dataflow information; data flow algorithms built on def-use chains, etc. gain asymptotic efficiency.

  In SSA, each use is reached by a unique def, so the size of def-use chains is linear to the number of edges in the CFG.

  In non-SSA, the def-use chains are much bigger.

SSA Construction [Cytron91]

• Turn every “preserving” def into a “killing” def, by copying potentially unmodified values (at subscripted defs, call sites, aliased defs, etc.)

• Every ordinary definition of \( v \) defines a new name.

• At each node in the flow graph where multiple definitions of \( v \) meets, a \( \phi \)-function is introduced to represent yet another new name of \( v \).

• Uses are renamed by their dominating definitions (where uses at a \( \phi \)-function are regarded as belonging to the appropriate predecessor node of the \( \phi \)-function).

• **Code Size**: the \( \phi \)-function inserted in SSA can increase the code size, but only linearly; in practice, the ratio of SSA over OLD is 0.6 - 2.4.