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)

- A code optimizer is often organized as follows:
  - Control-Flow Analysis --- divide the IR into basic blocks, build the control-flow graph (CFG)
  - Data-Flow Analysis --- gather data-flow information (e.g., the set of live variables).
  - Code Transformations --- the actual optimizations

Examples: Source Code

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

```c
void quicksort(int n);

int i, j, v, x;
if (n <= m) return;

i = m-1; j = n; v = a[n];
while (1) {
  do i = i+1; while ( a[i] < v);
  do j = j-1; while ( a[j] > v);
  if (i >= j) break;
  x = a[i]; a[i] = a[j]; a[j] = x;
}

x = a[i]; a[i] = a[n]; a[n] = x;
quicksort(m,j); quicksort(i+1,n);
```
Example: Intermediate Code

- Intermediate code for the shaded fragments of previous example:

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(01)</td>
<td>i := n - 1</td>
<td>(06)</td>
<td>t2 := 4 * i</td>
<td>(16)</td>
<td>t7 := 4 * i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(02)</td>
<td>j := n</td>
<td>(07)</td>
<td>t3 := a[c3]</td>
<td>(17)</td>
<td>t8 := 4 * j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(03)</td>
<td>t1 := 4 * n</td>
<td>(08)</td>
<td>if t3 &lt; v goto (5)</td>
<td>(18)</td>
<td>t9 := a[t8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(05)</td>
<td>i := i + 1</td>
<td>(10)</td>
<td>t4 := 4 * j</td>
<td>(20)</td>
<td>t10 := 4 * j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(06)</td>
<td>t2 := 4 * i</td>
<td>(11)</td>
<td>t5 := a[t4]</td>
<td>(21)</td>
<td>a[c10] := x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(07)</td>
<td>c3 := a[c2]</td>
<td>(12)</td>
<td>if t5 &gt; v goto (9)</td>
<td>(22)</td>
<td>goto (5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(08)</td>
<td>if t3 &lt; v goto (5)</td>
<td>(13)</td>
<td>if i &gt;= j goto (23)</td>
<td>(23)</td>
<td>t11 := 4 * i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(09)</td>
<td>j := j - 1</td>
<td>(14)</td>
<td>c6 := 4 * i</td>
<td>(24)</td>
<td>x := a[t11]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(10)</td>
<td>t4 := 4 * j</td>
<td>(15)</td>
<td>x := a[c6]</td>
<td>(25)</td>
<td>t12 := 4 * i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(11)</td>
<td>t5 := a[t4]</td>
<td>(16)</td>
<td>t7 := 4 * i</td>
<td>(26)</td>
<td>t13 := 4 * n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(12)</td>
<td>if t5 &gt; v goto (9)</td>
<td>(17)</td>
<td>t8 := 4 * j</td>
<td>(27)</td>
<td>t14 := a[t13]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(13)</td>
<td>if i &gt;= j goto (23)</td>
<td>(18)</td>
<td>a[c12] := t14</td>
<td>(28)</td>
<td>a[c12] := t14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(14)</td>
<td>t6 := 4 * i</td>
<td>(19)</td>
<td>t15 := 4 * n</td>
<td>(29)</td>
<td>t16 := 4 * n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 \( p_1 \) to \( p_n \) is a sequence of points \( p_1, ..., p_n \) such that \( p_i \) and \( p_{i+1} \) are “adjacent” for all \( i = 1, ..., n-1 \).

- For each statement \( S \), we associate it with four sets:
  - \( in[S] \): the set of data-flow info. associated with the point before \( S \)
  - \( out[S] \): the set of data-flow info. associated with the point after \( S \)
  - \( gen[S] \): the set of data-flow info. generated by \( S \)
  - \( kill[S] \): 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 \( in[S_2] = out[S_1] \).

- We can define these four sets for each basic block \( B \) in the same way. The \( gen \) and \( kill \) 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 */
change := true;
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] \union in[B] \ subtract 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 ϕ-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 ϕ-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 ϕ-function is introduced to represent yet another new name of \( v \).

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

• **Code Size**: the f-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.