Code Generation for Control Flow

Mooly Sagiv

html://www.cs.tau.ac.il/~msagiv/courses/wcc06.html

Chapter 6.4
Outline

• Local flow of control
• Conditionals
• Switch
• Loops
## Machine Code Assumptions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOTO Label</td>
<td>Jump to Label</td>
</tr>
<tr>
<td>GOTO label register</td>
<td>Indirect jump</td>
</tr>
<tr>
<td>IF condition register then GOTO Label</td>
<td>Conditional Jump</td>
</tr>
<tr>
<td>IF not condition register then GOTO Label</td>
<td></td>
</tr>
</tbody>
</table>
Boolean Expressions

• In principle behave like arithmetic expressions
• But are treated specially
  – Different machine instructions
  – Shortcut computations

if (a < b) goto l

Code for a < b yielding a condition value
Conversion condition value into Boolean
Conversion from Boolean in condition value
Jump to l on condition value
Shortcut computations

- Languages such as C define shortcut computation rules for Boolean
- Incorrect translation of e1 && e2

  Code to compute e1 in loc1
  Code to compute e2 in loc2
  Code for && operator on loc1 and loc2
Code for Booleans
(Location Computation)

• Top-Down tree traversal
• Generate code sequences instructions
• Jump to a designated ‘true’ label when the Boolean expression evaluates to 1
• Jump to a designated ‘false’ label when the Boolean expression evaluates to 0
• The true and the false labels are passed as parameters
Example

if ((a==0) && (b > 5))

    x = ((7 * a) + b)
if \( \text{&&} \text{No label Lf} \text{Lt Lf} \text{lf} \text{Lt := Lf} \text{x += b} \)

Cmp_Constant R0, 0
IF NOT EQ THEN GOTO Lf
GOTO Lf

Cmp_Constant R0, 5
IF GT THEN GOTO Lt
GOTO Lf

Load_Local -8(FP), R0
Load_Local -12(FP), R0

Code for :=
PROCEDURE Generate code for Boolean control expression (Node, True label, False label):

SELECT Node .type:
    CASE Comparison type:  // <, >, ==, etc. in C
        Generate code for comparison expression (Node .expr);
        // The comparison result is now in the condition register
        IF True label /= No label:
            Emit ("IF condition register THEN GOTO" True label);
            IF False label /= No label:
                Emit ("GOTO" False label);
        ELSE True label = No label:
            IF False label /= No label:
                Emit ("IF NOT condition register THEN GOTO"
                    False label);
        
        CASE Lazy and type:  // the && in C
            Generate code for Boolean control expression
                (Node .left, No label, False label);
            Generate code for Boolean control expression
                (Node .right, True label, False label);
        CASE ...
    CASE Negation type:  // the ! in C
        Generate code for Boolean control expression
            (Node.arg, False label, True label);
Allocate two new labels Lf, Lend

Generate code for Boolean(left, 0, Lf)

Code for Boolean with jumps to Lf

Code for Boolean with jumps to Lf

Code for Boolean with jumps to Lf

if

GOTO Lend

Lf:

true sequence

Code for true sequence

false sequence

Code for false sequence

Lend:
Code generation for IF (no-else)

Allocate new label Lend

Generate code for Boolean(left, 0, Lend)

Code for Boolean with jumps to Lend

Boolean expression

true sequence

if

Lend:

Code for Boolean with jumps to Lend

Code for true sequence
Coercions into value computations

\[
x : a > b
\]

- Generate new label Lf
- Load_Constant R0, 0;
- Generate code for Boolean(right, 0, Lf)
- Load_Local -8(FP), R1;
- CMP R1, -12(FP);
- IF <= GOTO Lf;
- Load_Constant R0, 1
- Lf: Store_Local R0, -20(FP)
Effects on performance

- Number of executed instructions
- Unconditional vs. conditional branches
- Instruction cache
- Branch prediction
- Target look-ahead
Code for case statements

• Three possibilities
  – Sequence of IFs
    • $O(n)$ comparisons
  – Jump table
    • $O(1)$ comparisons
  – Balanced binary tree
    • $O(\log n)$ comparisons

• Performance depends on $n$
• Need to handle runtime errors
tmp_case_value := case expression;
IF tmp_case_value = l_1 THEN GOTO label_1;
...
IF tmp_case_value = l_n THEN GOTO label_n;
GOTO label_else; // or insert the code at label_else

label_1:
code for statement sequence_1
GOTO label_next;
...

label_n:
code for statement sequence_n
GOTO label_next;
label_else:
code for else-statement sequence
label_next:
Jump Table

• Generate a table of $L_{\text{high}} - L_{\text{low}} + 1$ entries
  – Filled at ?time

• Each entry contains the start location of the corresponding case or a special label

• Generated code

  tmp_case_value := case expression;
  if tmp_case_value < $L_{\text{low}}$ GOTO label_else;
  if tmp_case_value > $L_{\text{high}}$ GOTO label_else;
  GOTO table[tmp_case_value – $L_{\text{low}}$];
Balanced trees

- The jump table may be inefficient
  - Space consumption
  - Cache performance
- Organize the case labels in a balanced tree
  - Left subtrees smaller labels
  - Right subtrees larger labels
- Code generated for node_k

```plaintext
label_k: IF tmp_case_value < l_k THEN
        GOTO label of left branch;
    IF tmp_case_value > l_k THEN
        GOTO label of right branch;
    code for statement sequence;
    GOTO label_next;
```
Repetition Statements (loops)

• Similar to IFs
• Preserve language semantics
• Performance can be affected by different instruction orderings
• Some work can be shifted to compile-time
  – Loop invariant
  – Strength reduction
  – Loop unrolling
while statements

Generate new labels test_label, L_{end}

test_label:

while

Generate code for \text{Boolean}(\text{left}, 0, L_{end})

GOTO test_label;

L_{end}:

statement

Sequence

Code for

statement sequence

Boolean expression

Code for Boolean with jumps to L_{end}
while statements (2)

Generate labels test_label, Ls

GOTO test_label: while
Ls:

Code for Boolean with jumps to $L_S$

Generate code for Boolean(left, Ls, 0)

Code for statement sequence
For-Statements

• Special case of while
• Tricky semantics
  – Number of executions
  – Effect on induction variables
  – Overflow
Simple-minded translation

FOR i in lower bound .. upper bound DO
    statement sequence
END for

↓

i := lower_bound;
tmp_ub := upper_bound;
WHILE I <= tmp_ub DO
    code for statement sequence
    i := i + 1;
END WHILE
Correct Translation

FOR i in lower bound .. upper bound DO
  statement sequence
END for

↓

i := lower_bound;

tmp_ub := upper_bound;
IF i > tmp_ub THEN GOTO end_label;

loop_label:
  code for statement sequence
  if (i==tmp_ub) GOTO end_label;
  i := i + 1;
  GOTO loop_label;
end_label:
Tricky question

```plaintext
for (expr1; expr2; expr3) {
    body;
}
```

```plaintext
expr1;
while (expr2) {
    body;
    expr3;
}
```
Loop unrolling

FOR i := 1 to n DO
    sum := sum + a[i];
END FOR;

FOR i := 1 TO n-3 STEP 4 DO
    // The first loop takes care of the indices 1 .. (n div 4) * 4
    sum := sum + a[i];
    sum := sum + a[i+1];
    sum := sum + a[i+2];
    sum := sum + a[i+3];
END FOR;

FOR i := (n div 4) * 4 + 1 TO n DO
    // This loop takes care of the remaining indices
    sum := sum + a[i];
END FOR;
Summary

• Handling control flow statements is usually simple

• Complicated aspects
  – Routine invocation
  – Non local gotos
  – Runtime errors

• Runtime profiling can help