Machine Code Generation - 3

Y. N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
Outline of the Lecture

- Mach. code generation – main issues (in part 1)
- Samples of generated code (in part 2)
- Two Simple code generators (in part 2)
- Optimal code generation
  - Sethi-Ullman algorithm
  - Dynamic programming based algorithm
  - Tree pattern matching based algorithm
- Code generation from DAGs
- Peephole optimizations
Optimal Code Generation
- The Sethi-Ullman Algorithm

- Generates the shortest sequence of instructions
  - Provably optimal algorithm (w.r.t. length of the sequence)

- Suitable for expression trees (basic block level)

- Machine model
  - All computations are carried out in registers
  - Instructions are of the form $op \, R,R$ or $op \, M,R$

- **Always computes the left subtree into a register and reuses it immediately**

- Two phases
  - Labelling phase
  - Code generation phase
The Labelling Algorithm

- Labels each node of the tree with an integer:
  - fewest no. of registers required to evaluate the tree with no intermediate stores to memory
  - Consider binary trees
- For leaf nodes
  - if \( n \) is the leftmost child of its parent then
    \[
    \text{label}(n) := 1 \quad \text{else} \quad \text{label}(n) := 0
    \]
- For internal nodes
  - \( \text{label}(n) = \max (l_1, l_2), \text{ if } l_1 <> l_2 \)
    \[
    = l_1 + 1, \text{ if } l_1 = l_2
    \]
Labelling - Example
Code Generation Phase –
Procedure GENCODE(n)

- RSTACK – stack of registers, $R_0, \ldots, R_{r-1}$
- TSTACK – stack of temporaries, $T_0, T_1, \ldots$

A call to Gencode(n) generates code to evaluate a tree T, rooted at node n, into the register top (RSTACK) , and

- the rest of RSTACK remains in the same state as the one before the call

A swap of the top two registers of RSTACK is needed at some points in the algorithm to ensure that a node is evaluated into the same register as its left child.
The Code Generation Algorithm (1)

Procedure gencode(n);
{ /* case 0 */
  if
    n is a leaf representing operand N and is the leftmost child of its parent
  then
    print(LOAD N, top(RSTACK))
}
/* case 1 */

else if
    n is an interior node with operator OP, left child n1, and right child n2
then
    if label(n2) == 0 then {
        let N be the operand for n2;
        gencode(n1);
        print(OP N, top(RSTACK));
    }

/* case 2 */

else if ((1 ≤ label(n1) < label(n2))
          and (label(n1) < r))
then {
    swap(RSTACK); gencode(n2);
    R := pop(RSTACK); gencode(n1);
    /* R holds the result of n2 */
    print(OP R, top(RSTACK));
    push (RSTACK, R);
    swap(RSTACK);
}
The Code Generation Algorithm (4)

/* case 3 */

`else if ((1 \leq label(n2) \leq label(n1))
and (label(n2) < r))`

`then {`

gencode(n1);
R := pop(RSTACK); gencode(n2);
/* R holds the result of n1 */
print(OP top(RSTACK), R);
push (RSTACK, R);
}

```plaintext
def gencode(n):
    R := pop(RSTACK); gencode(n2);
    R := top(RSTACK);
    print(OP R
    push (RSTACK, R);
```
/* case 4, both labels are $\geq r$ */
else {
    gencode(n2); T := pop(TSTACK);
    print(LOAD top(RSTACK), T);
    gencode(n1);
    print(OP T, top(RSTACK));
    push(TSTACK, T);
}
}
Code Generation Phase – Example 1

No. of registers = \( r = 2 \)

\[ n_5 \rightarrow n_3 \rightarrow n_1 \rightarrow a \rightarrow \text{Load } a, R0 \]
\[ \rightarrow \text{op}_{n_1} b, R0 \]
\[ \rightarrow n_2 \rightarrow c \rightarrow \text{Load } c, R1 \]
\[ \rightarrow \text{op}_{n_2} d, R1 \]
\[ \rightarrow \text{op}_{n_3} R1, R0 \]
\[ \rightarrow n_4 \rightarrow e \rightarrow \text{Load } e, R1 \]
\[ \rightarrow \text{op}_{n_4} f, R1 \]
\[ \rightarrow \text{op}_{n_5} R1, R0 \]
Code Generation Phase – Example 2

No. of registers = \( r = 1 \).
Here we choose \( rst \) first so that \( lst \) can be computed into \( R0 \) later (case 4)

\[
\begin{align*}
n5 & \rightarrow n4 \rightarrow e \rightarrow \text{Load } e, R0 \\
   & \rightarrow \text{op}_{n4} f, R0 \\
   & \rightarrow \text{Load } R0, T0 \{\text{release } R0\} \\
   n3 & \rightarrow n2 \rightarrow c \rightarrow \text{Load } c, R0 \\
   & \rightarrow \text{op}_{n2} d, R0 \\
   & \rightarrow \text{Load } R0, T1 \{\text{release } R0\} \\
   n1 & \rightarrow a \rightarrow \text{Load } a, R0 \\
   & \rightarrow \text{op}_{n1} b, R0 \\
   & \rightarrow \text{op}_{n3} T1, R0 \{\text{release } T1\} \\
   & \rightarrow \text{op}_{n5} T0, R0 \{\text{release } T0\}
\end{align*}
\]
Dynamic Programming based Optimal Code Generation for Trees

- Broad class of register machines
  - $r$ interchangeable registers, $R_0,\ldots,R_{r-1}$
  - Instructions of the form $R_i := E$
    - If $E$ involves registers, $R_i$ must be one of them
    - $R_i := M_j, R_i := R_i \text{ op } R_j, R_i := R_i \text{ op } M_j, R_i := R_j, M_i := R_j$

- Based on principle of contiguous evaluation

- Produces optimal code for trees (basic block level)

- Can be extended to include a different cost for each instruction
Contiguous Evaluation

- First evaluate subtrees of $T$ that need to be evaluated into memory. Then,
  - Rest of $T1$, $T2$, $op$, in that order, OR,
  - Rest of $T2$, $T1$, $op$, in that order
- Part of $T1$, part of $T2$, part of $T1$ again, etc., is not contiguous evaluation
- Contiguous evaluation is optimal!
  - No higher cost and no more registers than optimal evaluation
1. Compute in a bottom-up manner, for each node $n$ of $T$, an array of costs, $C$

   $C[i] = \text{min cost of computing the complete subtree rooted at } n, \text{ assuming } i \text{ registers to be available}$

   - Consider each machine instruction that matches at $n$ and consider all possible contiguous evaluation orders (using dynamic programming)
   - Add the cost of the instruction that matched at node $n$
The Algorithm (2)

- Using $C$, determine the subtrees that must be computed into memory (based on cost)
- Traverse $T$, and emit code
  - memory computations first
  - rest later, in the order needed to obtain optimal cost
- Cost of computing a tree into memory = cost of computing the tree using all registers + 1 (store cost)
An Example

Max no. of registers = 2

Node 2: matching instructions

\[ R_i = R_i - M \ (i = 0,1) \] and
\[ R_i = R_i - R_j \ (i,j = 0,1) \]

\[ C2[1] = C4[1] + C5[0] + 1 = 1+0+1 = 2 \]

\[ C4[2] + C5[0] + 1, \]
\[ C4[1] + C5[1] + 1, \]
\[ C4[1] + C5[0] + 1 \} \]
\[ = \text{Min}\{1+1+1,1+0+1,1+1+1, \]
\[ 1+1+1,1+0+1\} \]
\[ = \text{Min}\{3,2,3,3,2\} = 2 \]

\[ C2[0] = 1 + C2[2] = 1+2 = 3 \]

Generated sequence of instructions

\[ R0 = c \]
\[ R1 = d \]
\[ R1 = R1 \div e \]
\[ R0 = R0 \times R1 \]
\[ R1 = a \]
\[ R1 = R1 - b \]
\[ R0 = R1 + R0 \]
Example – continued
Cost of computing node 3 with 2 registers

<table>
<thead>
<tr>
<th>#regs for node 6</th>
<th>#regs for node 7</th>
<th>cost for node 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>1+3+1 = 5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1+2+1 = 4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1+3+1 = 5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1+2+1 = 4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1+2+1 = 4</td>
</tr>
<tr>
<td></td>
<td><strong>min value</strong></td>
<td><strong>4</strong></td>
</tr>
</tbody>
</table>

Cost of computing with 1 register = 5 (row 4, red)
Cost of computing into memory = 4 + 1 = 5

Triple = (5,5,4)
Example – continued

Traversal and Generating Code

Min cost for node 1=7, Instruction: R0 := R1+R0
  Compute RST(3) with 2 regs into R0
  Compute LST(2) into R1
For node 3, instruction: R0 := R0 * R1
  Compute RST(7) with 2 regs into R1
  Compute LST(6) into R0
For node 7, instruction: R1 := R1 / e
  Compute RST(9) into memory (already available)
  Compute LST(8) into R1
For node 8, instruction: R1 := d
For node 6, instruction: R0 := c
For node 2, instruction: R1 := R1 – b
  Compute RST(5) into memory (available already)
  Compute LST(4) into R1
For node 4, instruction: R1 := a