Code Generation – Part 1

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

NPTEL Course on Compiler Design
Outline of the Lecture

1. Code generation – main issues
2. Samples of generated code
3. Two Simple code generators
4. Optimal code generation
   a) Sethi-Ullman algorithm
   b) Dynamic programming based algorithm
   c) Tree pattern matching based algorithm
5. Code generation from DAGs
6. Peephole optimizations

Topics 4(b), 4(c), 5, and 6 will be covered in part 2 of the lecture
Y.N. Srikant

**Code Generation – Main Issues (1)**

- **Transformation:**
  - Intermediate code $\to$ m/c code (binary or assembly)
  - We assume quadruples and CFG to be available

- **Which instructions to generate?**
  - For the quadruple $A = A+1$, we may generate
    - `Inc A or`
    - `Load A, R1`
    - `Add #1, R1`
    - `Store R1, A`
  - One sequence is faster than the other (cost implication)
Code Generation – Main Issues (2)

- In which order?
  - Some orders may use fewer registers and/or may be faster

- Which registers to use?
  - Optimal assignment of registers to variables is difficult to achieve

- Optimize for memory, time or power?

- Is the code generator easily retargetable to other machines?
  - Can the code generator be produced automatically from specifications of the machine?
Samples of Generated Code

- **B = A[i]**
  - Load \( i, R1 \) // \( R1 = i \)
  - Mult \( R1,4,R1 \) // \( R1 = R1*4 \)
  - // each element of array
  - // A is 4 bytes long
  - Load \( A(R1), R2 \) // \( R2=(A+R1) \)
  - Store \( R2, B \) // \( B = R2 \)

- **X[i] = Y**
  - Load \( Y, R1 \) // \( R1 = Y \)
  - Load \( j, R2 \) // \( R2 = j \)
  - Mult \( R2, 4, R2 \) // \( R2=R2*4 \)
  - Store \( R1, X(R2) \) // \( X(R2)=R1 \)

- **X = *p**
  - Load \( p, R1 \)
  - Load \( 0(R1), R2 \)
  - Store \( R2, X \)

- ***q = Y**
  - Load \( Y, R1 \)
  - Load \( q, R2 \)
  - Store \( R1, 0(R2) \)

- **if X < Y goto L**
  - Load \( X, R1 \)
  - Load \( Y, R2 \)
  - Cmp \( R1, R2 \)
  - Bltz \( L \)
Samples of Generated Code – Static Allocation (no JSR instruction)

Three Address Code

// Code for function F1
action code seg 1
  call F2
action code seg 2
  Halt

// Code for function F2
action code seg 3
  return

Activation Record for F1 (48 bytes)

<table>
<thead>
<tr>
<th>return address</th>
<th>data array</th>
<th>variable x</th>
<th>variable y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>4</td>
<td>40</td>
</tr>
</tbody>
</table>

Activation Record for F2 (76 bytes)

<table>
<thead>
<tr>
<th>return address</th>
<th>parameter 1</th>
<th>data array</th>
<th>variable m</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>72</td>
<td>4044</td>
</tr>
</tbody>
</table>
Samples of Generated Code –
Static Allocation (no JSR instruction)

// Code for function F1
200:   Action code seg 1
// Now store return address
240:   Move #264, 648
252:   Move param1, 652
256:   Jump 400 // Call F2
264:   Action code seg 2
280:   Halt
...
// Code for function F2
400:   Action code seg 3
// Now return to F1
440:   Jump @648
...

//Activation record for F1
//from 600-647
600:   //return address
604:   //space for array A
640:   //space for variable x
644:   //space for variable y
//Activation record for F2
//from 648-723
648:   //return address
652:   // parameter 1
656:   //space for array B
...
720:   //space for variable m
Samples of Generated Code – Static Allocation (with JSR instruction)

Three Adress Code

// Code for function F1
action code seg 1
   call F2
action code seg 2
   Halt

// Code for function F2
action code seg 3
   return

Activation Record for F1 (44 bytes)

0
   data array A
36
   variable x
40
   variable y

Activation Record for F2 (72 bytes)

0
   data array B
68
   variable m
Samples of Generated Code – Static Allocation (with JSR instruction)

// Code for function F1
200: Action code seg 1
// Now jump to F2, return addr
// is stored on hardware stack
240: JSR 400 // Call F2
248: Action code seg 2
268: Halt
...
// Code for function F2
400: Action code seg 3
// Now return to F1 (addr 248)
440: return
...

//Activation record for F1
//from 600-643
600: //space for array A
636: //space for variable x
640: //space for variable y

//Activation record for F2
//from 644-715
644: //space for array B
...
712: //space for variable m
Samples of Generated Code – Dynamic Allocation (no JSR instruction)

### Three Address Code

<table>
<thead>
<tr>
<th>Code for function F1</th>
<th>Code for function F2</th>
</tr>
</thead>
<tbody>
<tr>
<td>action code seg 1</td>
<td>action code seg 3</td>
</tr>
<tr>
<td>call F2</td>
<td>call F1</td>
</tr>
<tr>
<td>action code seg 2</td>
<td>action code seg 4</td>
</tr>
<tr>
<td>return</td>
<td>call F2</td>
</tr>
<tr>
<td></td>
<td>action code seg 5</td>
</tr>
<tr>
<td></td>
<td>return</td>
</tr>
</tbody>
</table>

### Activation Record

- **F1 (68 bytes)**
  - return address: 0
  - local data and other information: 4

- **F2 (96 bytes)**
  - return address: 0
  - local data and other information: 92
Samples of Generated Code – Dynamic Allocation (no JSR instruction)

//Initialization
100: Move #800, SP
...

//Code for F1
200: Action code seg 1
230: Add #96, SP
238: Move #254, @SP
246: Jump 300
254: Sub #96, SP
262: Action code seg 2
296: Jump @SP

//Code for F2
300: Action code seg 3
340: Add #68, SP
348: Move #364, @SP
356: Jump 200
364: Sub #68, SP
372: Action code seg 4
400: Add #96, SP
408: Move #424, @SP
416: Jump 300
424: Sub #96, SP
432: Action code seg 5
480: Jump @SP
Samples of Generated Code – Dynamic Allocation (with JSR instruction)

<table>
<thead>
<tr>
<th>Three Address Code</th>
<th>Activation Record for F1 (64 bytes)</th>
<th>Activation Record for F2 (92 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Code for function F1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>action code seg 1</td>
<td>36</td>
<td>parameter 1</td>
</tr>
<tr>
<td>call F2</td>
<td></td>
<td>local data and other information</td>
</tr>
<tr>
<td>action code seg 2</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td></td>
<td>local data and other information</td>
</tr>
<tr>
<td>// Code for function F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>action code seg 3</td>
<td></td>
<td>88</td>
</tr>
<tr>
<td>call F1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>action code seg 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>action code seg 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Samples of Generated Code – Dynamic Allocation (with JSR instruction)

//Initialization
100: Move #800, SP
...

//Code for F1
200: Action code seg 1
230: Add #92, SP
238: Move param1, @SP
242: JSR 290
250: Sub #92, SP
258: Action code seg 2
286: return

//Code for F2
290: Action code seg 3
330: Add #64, SP
338: JSR 200
346: Sub #64, SP
354: Action code seg 4
382: Add #92, SP
390: JSR 290
398: Sub #92, SP
406: Action code seg 5
454: return
A Simple Code Generator – Scheme A

- Treat each quadruple as a ‘macro’
  - Example: The quad $A := B + C$ will result in
    - Load $B$, $R1$   OR   Load $B$, $R1$
    - Load $C$, $R2$
    - Add $R2$, $R1$           Add $C$, $R1$
    - Store $R1$, $A$           Store $R1$, $A$
  - Results in inefficient code
    - Repeated load/store of registers
  - Very simple to implement
A Simple Code Generator – Scheme B

- Track values in registers and reuse them
  - If any operand is already in a register, take advantage of it
  - Register descriptors
    - Tracks <register, variable name> pairs
    - A single register can contain values of multiple names, if they are all copies
  - Address descriptors
    - Tracks <variable name, locations> pairs
    - A single name may have its value in multiple locations, such as, memory, register, and stack
A Simple Code Generator – Scheme B

- Leave computed result in a register as long as possible
- Store only at the end of a basic block or when that register is needed for another computation
  - On exit from a basic block, store only live variables which are not in their memory locations already (use address descriptors to determine the latter)
  - If liveness information is not known, assume that all variables are live at all times
Example

- **A := B + C**
  - If B and C are in registers R1 and R2, then generate
    - `ADD R2, R1` (cost = 1, result in R1)
      - legal only if B is *not live* after the statement
  - If R1 contains B, but C is in memory
    - `ADD C, R1` (cost = 2, result in R1) or
    - `LOAD C, R2
      `ADD R2, R1` (cost = 3, result in R1)
      - legal only if B is *not live* after the statement
      - attractive if the value of C is subsequently used (it can be taken from R2)
Next Use Information

- Next use info is used in code generation and register allocation
- Next use of $A$ in quad $i$ is $j$ if
  Quad $i : A = ...$ (assignment to $A$)
  \(\downarrow\) (control flows from $i$ to $j$ with no assignments to $A$)
  Quad $j : A \text{ op } B$ (usage of $A$)
- In computing next use, we assume that on exit from the basic block
  - All temporaries are considered non-live
  - All programmer defined variables (and non-temps) are live
- Each procedure/function call is assumed to start a basic block
- Next use is computed on a backward scan on the quads in a basic block, starting from the end
- Next use information is stored in the symbol table
### Example of computing Next Use

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$T_1 := 4 \times I$</td>
<td>$T_1$ – (nlv, lu 0, nu 5), $I$ – (lv, lu 3, nu 10)</td>
</tr>
<tr>
<td>4</td>
<td>$T_2 := \text{addr}(A) - 4$</td>
<td>$T_2$ – (nlv, lu 0, nu 5)</td>
</tr>
<tr>
<td>5</td>
<td>$T_3 := T_2[T_1]$</td>
<td>$T_3$ – (nlv, lu 0, nu 8), $T_2$ – (nlv, lu 5, nnu), $T_1$ – (nlv, lu 5, nu 7)</td>
</tr>
<tr>
<td>6</td>
<td>$T_4 := \text{addr}(B) - 4$</td>
<td>$T_4$ – (nlv, lu 0, nu 7)</td>
</tr>
<tr>
<td>7</td>
<td>$T_5 := T_4[T_1]$</td>
<td>$T_5$ – (nlv, lu 0, nu 8), $T_4$ – (nlv, lu 7, nnu), $T_1$ – (nlv, lu 7, nnu)</td>
</tr>
<tr>
<td>8</td>
<td>$T_6 := T_3 \times T_5$</td>
<td>$T_6$ – (nlv, lu 0, nu 9), $T_3$ – (nlv, lu 8, nnu), $T_5$ – (nlv, lu 8, nnu)</td>
</tr>
<tr>
<td>9</td>
<td>$\text{PROD} := \text{PROD} + T_6$</td>
<td>$\text{PROD}$ – (lv, lu 9, nnu), $T_6$ – (nlv, lu 9, nnu)</td>
</tr>
<tr>
<td>10</td>
<td>$I := I + 1$</td>
<td>$I$ – (lv, lu 10, nu 11)</td>
</tr>
<tr>
<td>11</td>
<td>if $I \leq 20$ goto 3</td>
<td>$I$ – (lv, lu 11, nnu)</td>
</tr>
</tbody>
</table>
Scheme B – The algorithm

- We deal with one basic block at a time
- We assume that there is no global register allocation
- For each quad $A := B \text{ op } C$ do the following
  - Find a location $L$ to perform $B \text{ op } C$
    - Usually a register returned by $\text{GETREG()}$ (could be a mem loc)
  - Where is $B$?
    - $B'$, found using address descriptor for $B$
    - Prefer register for $B'$, if it is available in memory and register
    - Generate $\text{Load } B'$, $L$ (if $B'$ is not in $L$)
  - Where is $C$?
    - $C'$, found using address descriptor for $C$
    - Generate $\text{op } C'$, $L$
  - Update descriptors for $L$ and $A$
  - If $B/C$ have no next uses, update descriptors to reflect this information
Function \textit{GETREG}() \\

Finds $L$ for computing $A := B \, op \, C$ \\
1. If $B$ is in a register (say $R$), $R$ holds no other names, and \\
   $\quad$ \textit{B has no next use, and B is not live after the block, then return $R$} \\
2. Failing (1), return an empty register, if available \\
3. Failing (2) \\
   $\quad$ If $A$ has a next use in the block, OR \\
   $\quad$ if $B \, op \, C$ needs a register (e.g., $op$ is an indexing operator) \\
   $\quad$ \textbf{Use a heuristic to find an occupied register} \\
   $\quad$ a register whose contents are referenced farthest in future, or \\
   $\quad$ the number of next uses is smallest etc. \\
   $\quad$ \textbf{Spill it by generating an instruction, MOV R,mem} \\
   $\quad$ \textit{mem} is the memory location for the variable in $R$ \\
   $\quad$ That variable is not already in \textit{mem} \\
   $\quad$ \textbf{Update Register and Address descriptors} \\
4. If $A$ is not used in the block, or no suitable register can be found \\
   $\quad$ \textbf{Return a memory location for $L$}
Example

T, U, and V are temporaries - **not live** at the end of the block
W is a non-temporary - **live** at the end of the block, 2 registers

<table>
<thead>
<tr>
<th>Statements</th>
<th>Code Generated</th>
<th>Register Descriptor</th>
<th>Address Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>T := A * B</td>
<td>Load A, R0</td>
<td>R0 contains T</td>
<td>T in R0</td>
</tr>
<tr>
<td></td>
<td>Mult B, R0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U := A + C</td>
<td>Load A, R1</td>
<td>R0 contains T</td>
<td>T in R0</td>
</tr>
<tr>
<td></td>
<td>Add C, R1</td>
<td>R1 contains U</td>
<td>U in R1</td>
</tr>
<tr>
<td>V := T - U</td>
<td>Sub R1, R0</td>
<td>R0 contains V</td>
<td>U in R1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R1 contains U</td>
<td>V in R0</td>
</tr>
<tr>
<td>W := V * U</td>
<td>Mult R1, R0</td>
<td>R0 contains W</td>
<td>W in R0</td>
</tr>
<tr>
<td></td>
<td>Load R0, W</td>
<td></td>
<td>W in memory (restored)</td>
</tr>
</tbody>
</table>
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 \textit{op R,R} or \textit{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 \textit{then}
    \[
    \text{label}(n) := 1 \quad \text{else} \quad \text{label}(n) := 0
    \]

- For internal nodes
  - \( \text{label}(n) = \max(l_1, l_2), \quad \text{if} \quad l_1 \neq l_2 \)
    - \( = l_1 + 1, \quad \text{if} \quad l_1 = l_2 \)
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))
The Code Generation Algorithm (2)

/* 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));
}

leaf node
The Code Generation Algorithm (3)

/* 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);
}
/* case 3 */

else if ((1 ≤ label(n2) ≤ 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);
}
/* case 4, both labels are ≥ r */

else {
    gencode(n2); T := pop(TSTACK);
    print(LOAD top(RSTACK), T);
    gencode(n1);
    print(OP T, top(RSTACK));
    push(TSTACK, T);
}
}