Machine Code Generation - 2

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
- Two Simple code generators
- Optimal code generation
  - Sethi-Ullman algorithm
  - Dynamic programming based algorithm
  - Tree pattern matching based algorithm
- Code generation from DAGs
- Peephole optimizations
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)

- return address
- data array A
  - variable x
  - variable y

Activation Record for F2 (76 bytes)

- return address
- parameter 1
- data array B
  - variable m
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 val1, 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

return address need not be stored
### Samples of Generated Code – Static Allocation (with JSR instruction)

<table>
<thead>
<tr>
<th>Code for function F1</th>
<th>Activation record for F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>200: Action code seg 1</td>
<td>//from 600-643</td>
</tr>
<tr>
<td>240: JSR 400 // Call F2</td>
<td>600: //space for array A</td>
</tr>
<tr>
<td>248: Action code seg 2</td>
<td>636: //space for variable x</td>
</tr>
<tr>
<td>268: Halt</td>
<td>640: //space for variable y</td>
</tr>
<tr>
<td>...</td>
<td>//Activation record for F2</td>
</tr>
<tr>
<td>...</td>
<td>//from 644-715</td>
</tr>
<tr>
<td>// Code for function F2</td>
<td>644: //space for array B</td>
</tr>
<tr>
<td>400: Action code seg 3</td>
<td>...</td>
</tr>
<tr>
<td>440: return</td>
<td>712: //space for variable m</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Samples of Generated Code – Dynamic Allocation (no JSR instruction)

Three Address Code

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

// Code for function F2
action code seg 3
  call F1
action code seg 4
  call F2
action code seg 5
  return

Activation Record for F1 (68 bytes)

```
0
return address
4
local data and other information
64
```

Activation Record for F2 (96 bytes)

```
0
return address
4
parameter 1
92
local data and other information
```
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 #258, @SP
246: Move val1, @SP+4
250: Jump 300
258: Sub #96, SP
266: 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: Move val2, @SP+4
420: Jump 300
428: Sub #96, SP
436: Action code seg 5
480: Jump  @SP
Samples of Generated Code – Dynamic Allocation (with JSR instruction)

Three Address Code

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

// Code for function F2
action code seg 3
   call F1
action code seg 4
   call F2
action code seg 5
   return

Activation Record for F1 (64 bytes)

Activation Record for F2 (92 bytes)

local data and other information

parameter 1

local data and other information
# Samples of Generated Code – Dynamic Allocation (with JSR instruction)

<table>
<thead>
<tr>
<th>//Initialization</th>
<th>//Code for F2</th>
</tr>
</thead>
<tbody>
<tr>
<td>100: Move #800, SP</td>
<td>290: Action code seg 3</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>//Code for F1</td>
<td>330: Add #64, SP</td>
</tr>
<tr>
<td>200: Action code seg 1</td>
<td>338: JSR 200</td>
</tr>
<tr>
<td>230: Add #92, SP</td>
<td>346: Sub #64, SP</td>
</tr>
<tr>
<td>238: Move val1, @SP</td>
<td>354: Action code seg 4</td>
</tr>
<tr>
<td>242: JSR 290</td>
<td>382: Add #92, SP</td>
</tr>
<tr>
<td>250: Sub #92, SP</td>
<td>390: Move val2, @SP</td>
</tr>
<tr>
<td>258: Action code seg 2</td>
<td>394: JSR 290</td>
</tr>
<tr>
<td>286: return</td>
<td>402: Sub #92, SP</td>
</tr>
<tr>
<td></td>
<td>410: Action code seg 5</td>
</tr>
<tr>
<td></td>
<td>454: return</td>
</tr>
</tbody>
</table>
A Simple Code Generator – Scheme A

- Treat each quadruple as a ‘macro’
  - Example: The quad $A := B + C$ will result in
    
    - Load $B$, $R_1$ OR Load $B$, $R_1$
    - Load $C$, $R_2$
    - Add $R_2$, $R_1$ OR Add $C$, $R_1$
    - Store $R_1$, $A$

  - 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, location> 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
  - A variable is **live** at a point, if it is used later, possibly in other blocks – obtained by dataflow analysis
  - 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
    - \textit{ADD} R2,R1 (cost = 1, result in R1)
      - legal only if B is \textit{not live} after the statement
  - If R1 contains B, but C is in memory
    - \textit{ADD} C,R1 (cost = 2, result in R1) \textbf{or}
    - \textit{LOAD} C, R2
      - \textit{ADD} R2,R1 (cost = 3, result in R1)
      - legal only if B is \textit{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 $A$)
  
  (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

<table>
<thead>
<tr>
<th>No.</th>
<th>Statement</th>
<th>Live Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>( T_1 := 4 \times I )</td>
<td>( T_1 \rightarrow (nlv, lu 0, nu 5), I \rightarrow (lv, lu 3, nu 10) )</td>
</tr>
<tr>
<td>4</td>
<td>( T_2 := \text{addr}(A) - 4 )</td>
<td>( T_2 \rightarrow (nlv, lu 0, nu 5) )</td>
</tr>
<tr>
<td>5</td>
<td>( T_3 := T_2[T_1] )</td>
<td>( T_3 \rightarrow (nlv, lu 0, nu 8), T_2 \rightarrow (nlv, lu 5, nnu), T_1 \rightarrow (nlv, lu 5, nu 7) )</td>
</tr>
<tr>
<td>6</td>
<td>( T_4 := \text{addr}(B) - 4 )</td>
<td>( T_4 \rightarrow (nlv, lu 0, nu 7) )</td>
</tr>
<tr>
<td>7</td>
<td>( T_5 := T_4[T_1] )</td>
<td>( T_5 \rightarrow (nlv, lu 0, nu 8), T_4 \rightarrow (nlv, lu 7, nnu), T_1 \rightarrow (nlv, lu 7, nnu) )</td>
</tr>
<tr>
<td>8</td>
<td>( T_6 := T_3 \times T_5 )</td>
<td>( T_6 \rightarrow (nlv, lu 0, nu 9), T_3 \rightarrow (nlv, lu 8, nnu), T_5 \rightarrow (nlv, lu 8, nnu) )</td>
</tr>
<tr>
<td>9</td>
<td>( \text{PROD} := \text{PROD} + T_6 )</td>
<td>( \text{PROD} \rightarrow (lv, lu 9, nnu), T_6 \rightarrow (nlv, lu 9, nnu) )</td>
</tr>
<tr>
<td>10</td>
<td>( I := I + 1 )</td>
<td>( I \rightarrow (lv, lu 10, nu 11) )</td>
</tr>
<tr>
<td>11</td>
<td>if ( I \leq 20 ) goto 3</td>
<td>( I \rightarrow (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 $A := B \text{ op } C$ \\
1. If $B$ is in a register (say $R$), $R$ holds no other names, and \\
   $B$ 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) \\
   - If $A$ has a next use in the block, OR \\
     if $B \text{ op } C$ needs a register (\textit{e.g.}, \textit{op} is an indexing operator) \\
     - Use a \textit{heuristic} to find an occupied register \\
       - a register whose contents are referenced farthest in future, or \\
       - the number of next uses is smallest etc. \\
     - \textit{Spill} it by generating an instruction, \textit{MOV R, mem} \\
       - \textit{mem} is the memory location for the variable in $R$ \\
       - That variable is not already in \textit{mem} \\
   - Update Register and Address descriptors \\
4. If $A$ is not used in the block, or no suitable register can be found \\
   - 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>Store R0, W</td>
<td></td>
<td>W in memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(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 \( op \ R, R \) 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
    $$\text{label}(n) := 1 \text{ else label}(n) := 0$$

- For internal nodes
  - $\text{label}(n) = \max(l_1, l_2)$, if $l_1 \neq l_2$
    $$= l_1 + 1, \text{ if } l_1 = l_2$$
Labelling - Example