Code Generation – Part 2

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 1,2,3, and 4(a) were covered in part 1 of the lecture
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 \( \text{op} R, R \) or \( \text{op} M, R \)
- 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
  - \textit{if} \( n \) is the leftmost child of its parent \texttt{then}
    
    \[
    \text{label}(n) := 1 \texttt{else label}(n) := 0
    \]

- For internal nodes
  - \( \text{label}(n) = \max (l_1, l_2), \texttt{if} \ l_1 \neq l_2 \)
    
    \[
    = l_1 + 1, \texttt{if} \ l_1 = l_2
    \]
Labelling - Example
Code Generation Phase –
Procedure GENCODE(n)

- RSTACK – stack of registers, $R_0, ..., R_{(r-1)}$
- TSTACK – stack of temporaries, $T_0, T_1, ...$

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));
    }

leaf node
/* case 2 */

\textit{else if} \(((1 < \text{label}(n1) < \text{label}(n2))\) \\
\text{and(} \text{label}(n1) < r)\) \\
\textit{then} \{ \\
\text{swap}(\text{RSTACK}); \text{gencode}(n2); \\\n\text{R := pop}(\text{RSTACK}); \text{gencode}(n1); \\\n/* R holds the result of n2 */ \\\n\text{print}(\text{OP R, top}(\text{RSTACK})); \\\n\text{push}(\text{RSTACK}, \text{R}); \\\n\text{swap}(\text{RSTACK}); \\\n\}

The swap() function ensures that a node is evaluated into the same register as its left child
/* case 3 */

```c
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 $\geq r$ */
else {
    gencode(n2); T:= pop(TSTACK);
    print(LOAD top(RSTACK), T);
    gencode(n1);
    print(OP T, top(RSTACK));
    push(TSTACK, T);
}
}
No. of registers $= r = 2$

$n5 \rightarrow n3 \rightarrow n1 \rightarrow a \rightarrow \text{Load } a, R0$
$\rightarrow \text{op}_{n1} b, R0$
$\rightarrow n2 \rightarrow c \rightarrow \text{Load } c, R1$
$\rightarrow \text{op}_{n2} d, R1$
$\rightarrow \text{op}_{n3} R1, R0$
$\rightarrow n4 \rightarrow e \rightarrow \text{Load } e, R1$
$\rightarrow \text{op}_{n4} f, R1$
$\rightarrow \text{op}_{n5} R1, R0$
No. of registers = \( r = 1 \).
Here we choose \( rst \) first so that \( lst \) can be computed into \( R0 \) later (case 4)

\[
\begin{align*}
n5 & \rightarrow n4 \rightarrow e \rightarrow \text{Load } e, \text{ R0} \\
& \rightarrow \text{op}_{n4} f, \text{ R0} \\
& \rightarrow \text{Load } R0, \text{ T0} \{\text{release R0}\} \\
& \rightarrow n3 \rightarrow n2 \rightarrow c \rightarrow \text{Load } c, \text{ R0} \\
& \rightarrow \text{op}_{n2} d, \text{ R0} \\
& \rightarrow \text{Load } R0, \text{ T1} \{\text{release R0}\} \\
& \rightarrow n1 \rightarrow a \rightarrow \text{Load } a, \text{ R0} \\
& \rightarrow \text{op}_{n1} b, \text{ R0} \\
& \rightarrow \text{op}_{n3} \text{ T1, R0} \{\text{release T1}\} \\
& \rightarrow \text{op}_{n5} \text{ T0, R0} \{\text{release T0}\}
\end{align*}
\]
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 $T_1$, $T_2$, $op$, in that order, OR,
  - Rest of $T_2$, $T_1$, $op$, in that order
- Part of $T_1$, part of $T_2$, part of $T_1$ 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, $C$

- $C[i] = \min$ cost of computing the complete subtree rooted at $n$, assuming $i$ 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
\( R_i = R_i - R_j \) (\( i,j = 0,1 \))

\[ C_2[1] = C_4[1] + C_5[0] + 1 \]
\[ = 1 + 0 + 1 = 2 \]

\[ C_4[2] + C_5[0] + 1, \]
\[ C_4[1] + C_5[2] + 1, \]
\[ C_4[1] + C_5[1] + 1, \]
\[ C_4[1] + C_5[0] + 1 \}\]
\[ = \text{Min}\{1 + 1 + 1 + 1 + 0 + 1, 1 + 1 + 1 + 1 + 1, \]
\[ 1 + 1 + 1 + 1 + 0 + 1 \}\]
\[ = \text{Min}\{3,2,3,2,3\} = 2 \]

\[ C_2[0] = 1 + C_2[2] = 1 + 2 = 3 \]
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>min value</td>
<td>4</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: \( R0 := R0 \times R1 \)
Compute RST(7) with 2 regs into R1
Compute LST(6) into R0
For node 2, instruction: \( R1 := R1 - b \)
Compute RST(5) into memory (available already)
Compute LST(4) into R1
For node 4, instruction: \( R1 := a \)
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: \( R0 := c \)
Code Generation by Tree Rewriting

- Caters to complex instruction sets and very general machine models
- Can produce locally optimal code (basic block level)
- Non-contiguous evaluation orders are possible without sacrificing optimality
- Easily retargetable to different machines
- Automatic generation from specifications is possible
Example

Tree intermediate code for $a[i] = b+1$, $a$ and $i$ are local, and $b$ is global.
Match #1

Pattern
reg_i \leftarrow \text{const}_a

Code
Load \#a, R0

\text{Code so far:}
Load \#a, R0
Match #2

Pattern
\[ \text{reg}_i \leftarrow + \left( \text{reg}_i, \text{reg}_j \right) \]

Code
Add SP, R0

Code so far:
Load #a, R0
Add SP, R0
Match #3

Pattern
\[ \text{reg}_i \leftarrow \text{ind} \left( + (\text{const}_c, \text{reg}_j) \right) \]
OR
\[ \text{reg}_i \leftarrow + (\text{reg}_i, \text{ind} \left( + (\text{const}_c, \text{reg}_j) \right)) \]

Code for 2nd alternative (chosen)
Add #i(SP), R0

Code so far:
Load #a, R0
Add SP, R0
Add #i(SP), R0
Match #4

Code so far:
Load #a, R0
Add SP, R0
Add #i(SP), R0
Load b, R1

Pattern
\( \text{reg}_i \leftarrow \text{mem}_a \)

Code
Load b, R1
**Match #5**

Pattern
\[ \text{reg}_i \leftarrow + (\text{reg}_i, \text{const}_1) \]

Code so far:
- Load \#a, R0
- Add SP, R0
- Add \#i(SP), R0
- Load b, R1
- Inc R1

Code
- Inc R1
Match #6

Pattern

\[ \text{mem} \leftarrow \text{:=} (\text{ind} (\text{reg}_i), \text{reg}_j) \]

Code

\[
\text{Load } R1, *R0
\]

**Code so far:**
- Load #a, R0
- Add SP, R0
- Add #i(SP), R0
- Load b, R1
- Inc R1
- Load R1, *R0
Code Generator Generators (CGG)

- Based on tree pattern matching and dynamic programming
- Accept tree patterns, associated costs, and semantic actions (for register allocation and object code emission)
- Produce tree matchers that produce a cover of minimum cost
- Make two passes
  - First pass is a bottom-up pass and finds a set of patterns that cover the tree with minimum cost
  - Second pass executes the semantic actions associated with the minimum cost patterns at the nodes they matched
- BEG, Twig, BURG, and IBURG are such CGGs
Code Generator Generators (2)

- **BEG and IBURG**
  - Produce similar matchers
  - Use dynamic programming (DP) at compile time
  - Costs can involve arbitrary computations
  - The matcher is hard coded

- **TWIG**
  - Uses a table-driven tree pattern matcher based on Aho-Corasick string pattern matcher
  - High overheads, could take $O(n^2)$ time, $n$ being the number of nodes in the subject tree
  - Uses DP at compile time
  - Costs can involve arbitrary computations

- **BURG**
  - Uses BURS (bottom-up rewrite system) theory to move DP to compile-compile time (matcher generation time)
  - Table-driven, more complex, but generates optimal code in $O(n)$ time
  - Costs must be constants
EBNF Grammar for *iburg* Specifications
(Adapted From Fraser [ACM LOPLAS, Sep 1992])

```
grammar → { dcl } %%% { rule }
dcl → %START nonterm
   | %TERM { identifier = integer }
rule → nonterm : tree = integer [ cost ] ;
cost → ( integer )
tree → term ( tree , tree )
   | term ( tree )
   | term
   | nonterm
```
IBURG Specifications (2) (Adapted from Fraser [ACM LOPLAS, Sep 1992])

1. %term ADDI=309  ADDRLP=295  ASGNI=53
2. %term CNSTI=21  CVCI=85  I0I=661  INDIRC=67
3. %%
4. stmt:    ASGNI (disp,reg) = 4 (1);
5. stmt:    reg = 5;
6. reg:     ADDI (reg,rc) = 6 (1);
7. reg:     CVCI (INDIRC (disp)) = 7 (1);
8. reg:     I0I = 8;
9. reg:     disp = 9 (1);
10. disp:   ADDI (reg,con) = 10;
11. disp:   ADDRLP = 11;
12. rc:     con = 12;
13. rc:     reg = 13;
14. con:    CNSTI = 14;
15. con:    I0I = 15;
IBURG Tree Matcher

- Produces two functions, *label* and *reduce*
- User calls these routines
- *label(p)* makes a bottom-up, left-to-right pass over the subject tree *p* and computes the minimum cost cover, if there is one
- Each node is labeled with \((M,C)\) (or \([M,C]\) for chain rules) to indicate that *the pattern associated with rule* \(M\) *matches the node with cost* \(C\)
- Nodes are annotated with \((M,C)\) (or \([M,C]\)) only if \(C\) is min cost for nonterminal of rule \(M\) (considering all rules that match as well)
  - Example: For ADDI node, rule 10 matches, and the chain rules 9, 5, and 13 also match
  - But, cost of this match for rules 9, 5, and 13 is not less than the cost during previous matches for the same nonterminals *reg*, *stmt*, and *rc* on the LHS of rules 9, 5, and 13 resp.
Example of Labeling \{\text{int } i; \text{ char } c; i = c + 4;\}

(Adapted From Fraser [ACM LOPLAS, Sep 1992])
IBURG Tree Matcher (2)

- Once labeled, the *reducer* traverses the subject tree, in a top-down manner.
- During a visit to each node, user-supplied code that implements semantic side effects such as register allocation and emission of code, is executed.
Code Generation from DAGs

- Optimal code generation from DAGs is NP-Complete
- DAGs are divided into trees and then processed
- We may replicate shared trees
  - Code size increases drastically
- We may store result of a tree (root) into memory and use it in all places where the tree is used
  - May result in sub-optimal code
DAG example: Duplicate shared trees
DAG example: Compute shared trees once and share results

[Diagram of a Directed Acyclic Graph (DAG) with nodes labeled 1 to 11 and edges connecting them to form shared trees.]