Code Generation

Chapter 9
Issues in Code Generation

Target language

- Relocatable machine code
  
  Commercial compilers produce this

- Absolute machine code
  
  OS code must start at a particular memory address

- Assembly language
  
  - Easier, assembler does the linking
  
  - Commercial compilers don’t do this normally

- High-level language
  
  E.g., cfront
Issues in Code Generation

- Instruction selection

  \[
  a = a + 1;
  \]

  LDR R0, a  
  ADD R0, 1  
  STR R0, a

  (If a memory increment instruction exists)

- Registers
  - Maximize register hit ratios
  - NP-complete problem, heuristics are helpful
Code optimization

- By analyzing various aspects of the code it may be possible to transform the program into an equivalent but more efficient form.

- Example, code motion:

```c
while (a < b) {
    c = 10 * d;
    a = a + c;
}
```
Basic Blocks

- A sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any branching
- A basic block always ends at a branch or at end of code

```plaintext
a = 0
b = c + b
M: if (c < a ) goto L
c = c + b
a = a * 2
goto M
L: print c
```
Dividing Code into Basic Blocks

1
a = 0
b = c + b

2
M: if (c < a ) goto L

3
c = c + b
a = a * 2
goto M

4
L: print c
Variables and Basic Blocks

- **Variable definition**
  A statement that assigns a value to a variable

- **Variable use**
  A statement that accesses the value of a variable

Example:

\[ a = b + c \]

- \( a \) is defined
- \( b \) is used
Flow graph

- A directed graph that shows the flow of control among basic blocks

- There will be a directed edge from $B_1$ to $B_2$ if either of the following two conditions are satisfied:
  1. there is a conditional or unconditional jump from $B_1$ to $B_2$
  2. $B_2$ immediately follows $B_1$ and $B_1$ does not have an unconditional jump
Basic Blocks and Flow Graphs

- Within a basic block we can compute uses of variables.
- Basic blocks allow us to do local optimizations.
- Flow graphs allow us to global optimizations.
Partitioning Algorithm

Program $\rightarrow$ basic blocks

- Input: A sequence of 3-address statements
- Output: A list of basic blocks
Partitioning Algorithm

1. Determine the set of leaders, the first statement in a basic block
   - The first statement in a program is a leader
   - Any statement that is the target of a conditional or unconditional jump is a leader (can’t branch into the middle of a basic block)
   - Any statement that immediately follows a conditional or unconditional jump is a leader

2. For each leader, its basic block consists of the leader and all statements up to, but not including, the next leader or the end of the program
Now we can

- construct flow graphs for global optimization
- do local optimizations on basic blocks

Here *local* and *global* mean relative to basic blocks.
Potential Optimizations on Basic Blocks

Potential optimizations include

- Common subexpression elimination
- Dead code elimination
- Algebraic optimizations
Common Subexpression Elimination

\[
\begin{align*}
a &= b + c \\
b &= a - d \\
c &= b + c \\
d &= a - d
\end{align*}
\]
Dead Code Elimination

a = b + c
b = d + e
e = b + f
a = e + x
Algebraic Optimizations

Use algebraic identities to eliminate code

\[
\begin{align*}
x &= x + 0; \\
y &= 1 \times y;
\end{align*}
\]

Can happen more often than you think

/* In C/C++ */
#define OFFSET 0
#define FACTOR 1
.
.
.
x = x + OFFSET;
y = FACTOR \times y;
Next Use Information

Allows us to determine if a variable is “live”

- A live variable is used subsequently in the basic block
- Applies only locally
- Need to perform data flow analysis on the flow graph for global optimization
Let the statements within a basic block be numbered sequentially from 1 to \( n \);
for ( \( i \leftarrow n; i > 0; i \leftarrow i - 1 \) ) {
    \textit{Scan backwards}
    if ( Statement \( i \) has the form \( x = y \ op \ z \) ) {
        Attach to Statement \( i \) the information found in the symbol table
        about the next use and liveness of \( x \), \( y \), and \( z \);
        In the symbol table, set \( x \) to “not live” and “no next use”
        In the symbol table, set \( y \) and \( z \) to “live” and their next use to \( i \);
    }
}
Example

1. \( t_1 = a \times b \)
2. \( t_2 = a \times a \)
3. \( t_3 = 2 \times t_2 \)
4. \( t_4 = t_1 + t_3 \)
5. \( t_5 = b \times b \)
6. \( t_6 = t_4 + t_5 \)