UNIT V  CODE OPTIMIZATION AND RUN TIME ENVIRONMENTS


What is code optimization?
The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine. Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

1) List the criteria for code improvement transformations.

Simply stated, the best program transformations are those that yield the most benefit for the least effort.
First, the transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error.
Second, a transformation must, on the average, speed up programs by a measurable amount.
Third, the transformation must be worth the effort.
Some transformations can only be applied after detailed, often time-consuming analysis of the source program, so there is little point in applying them to programs that will be run only a few times.

Code Improvement Criteria
A transformation must preserve meaning of a program (correctness

- A transformation must improve (e.g., speed-up) programs by a measurable amount on average
- A transformation must worth the effort

2) Indicate the places for potential improvements can be made by the user and the compiler.

Programmer
Profiles, change algorithm, transform loops

**Compiler: (on intermediate code)**

Improve loops, procedure calls, various transformations

**Compiler: (on target code)**

Use of registers, instruction selection, peephole optimization

3) **What are the phases of code improvement?**
The code improvement phase consists of control-flow and data-flow analysis followed by the application of transformations.

4) **Discuss in detail the principal sources of optimization.**

   **Principal Sources of Optimization**

   **Function-preserving transformations**
   - Common sub-expression elimination
   - Copy propagation

   ```
   a := x + y
   b := x + y
   c := x + y
   w := x + y
   a := w
   ```
**Dead-code elimination**

- Dead (or useless) code: statements that evaluate values that never get used
- Dead-code may appear after transformations

- **Constant folding**
  - Deducing at compile-time value of an expression is a constant and using the constant instead
- **Loop optimizations (especially inner loops)**
  - Programs tend to spend most of their time in inner loops
  - We may improve running time by *decreasing the number of instructions in an inner loop* even while *increasing the amount of code outside the loop*

**Code motion**

Placing *loop-invariant computation* before the loop

E.g.,

```c
while ( i <= limit-2 ) /* limit is loop invariant */
```

```
t = limit – 2;
while ( i <= t ) /* limit, t are loop-invariant */
```
**Strength reduction**

E.g., The replacement of multiplication by a subtraction or addition might be possible when induction vars are modified in a loop.

5) Discuss optimization of basic blocks.

**Optimizing Basic Blocks**

- **Local transformations that preserve structure**
  - Common sub-expression elimination
  - Dead-code elimination

- **Local transformations that are algebraic**
  - Arithmetic identities; e.g., $x+0 = x$, $x/1 = x$
  - Strength reduction; e.g., $x**2 = x\ast x$, $x/2=x\ast 0.5$
  - Constant folding

6) What is dataflow analysis?

An optimizing compiler needs to

- Collect info about the program as a whole
- Distribute info to each block in the flow graph
- E.g., knowing which vars are live on exit from each block and using it for register allocation

- This info is *dataflow information* and a compiler collects this by *dataflow analysis*
- Info can be collected by setting up and solving systems of equations
- They relate info at various parts in a program
A *dataflow equation* is of the form:

\[ \text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S]) \]

Meaning: “*information at the end of statement (or block) S is either generated within it, or enters at the beginning and is not killed as control flows through it*”

To set up and solve equations, may have to proceed forward or backwards

- i.e., backwards: define \( \text{in}[S] \) in terms of \( \text{out}[S] \)
- Equations are normally set up at basic block level

Need to handle complexities involved in function calls and use of pointers

**Dataflow Analysis: Examples**

- *Reaching Definitions (ud-chains)*
  - Determines which *definitions* of (assignments to) a variable may reach each *use* of it

- *Available Expressions*
  - Determines which expressions are *available* at each point in a program (on every path from the entry to the point, there is an evaluation of the expression and none of the vars in it are assigned values between the last evaluation and the point)

- *Liveness (Live-Variable) Analysis*
  - Determines for a given var and a given point whether there is a use of the var along some path from the point to the exit

- *Upwards Exposed Uses (du-chains)*
  - Determines what *uses* of vars at certain points are reached by particular *definitions* (this is the dual of reaching definitions)

**Example**

*Reaching definitions* and its *dual*:
- *Use* of \( x \) in B5 is reached by the *definitions* in B2, B3
- *Definition* of \( x \) in B2 reaches the *uses* in B4, B5
Dataflow Analysis: Examples

Copy-Propagation Analysis

Determines that on every path from a copy assignment such as \( x := y \), to a use of \( x \), there is no assignment to \( y \).

Constant-Propagation Analysis

Determines that on every path from an assignment of a constant to a variable, \( x := k \), to a use of \( x \), the only assignment to \( x \) assigns the value \( k \).

What is an activation tree? Explain its functions.

Every execution of a procedure is called an activation. We can use a tree, called an activation tree, to depict the way control enters and leaves activations.

The lifetime of an activation of procedure \( P \) is the sequence of steps between the first and last steps of \( P \)'s body, including any procedures called while \( P \) is running.

Normally, when control flows from one activation to another, it must (eventually) return to the same activation.

When activations are thusly nested, we can represent control flow with activation trees.
1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for a is the parent of the node b if and only if control flows from activation a to b.
4. The node for 'a' is to the left of the node for 'b' if and only if the lifetime of 'a' occurs before the lifetime of 'b'.

8) **What are control stacks?**
The flow of control in a program corresponds to a depth first traversal of the activation tree that starts at the root, visits a node before its children, and recursively visits children at each node in a left-to-right order.

We can use a stack, called a control stack to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack and to pop the node when the activation ends.

When node 'n' is at the top of the control stack, the stack contains the nodes along the path from 'n' to the root.

9) **What are the storage allocation strategies used at Run time?**

**Storage Allocation**

**Static allocation**
Storage allocation was fixed during the entire execution of a program.

**Stack allocation**
Space is pushed and popped on a run-time stack during program execution such as procedure calls and returns.

**Heap allocation**
Allow space to be allocated and freed at any time.

10) Explain with a diagram the run time storage organization.
11) What is static allocation?

Static Allocation

Bindings between names and storage are fixed.

The values of local names are retained across activations of a procedure.

Addressing is efficient.

Limitations:
- No recursive procedures
- No dynamic data structures

12) What is stack allocation?

Stack Allocation

- Recursive procedures require stack allocation.
- Activation records (AR) are pushed and popped as activations begin and end.
- The offset of each local data relative to the beginning of AR is stored in the symbol table.

```c
float f(int k)
{
    float c[10], b;
    b = c[k] * 3.14;
    return b;
}
```

<table>
<thead>
<tr>
<th>Return value</th>
<th>offset = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter k</td>
<td>offset = 4</td>
</tr>
<tr>
<td>Local c[10]</td>
<td>offset = 8</td>
</tr>
</tbody>
</table>
13) What are calling sequences?
Procedure calls are implemented by generating what are known as calling sequences in the target code. A call sequence allocates an activation record and enters information into its fields. A return sequence restores the state of the machine so the calling procedure can continue execution.

Calling Sequences

A call sequence allocates an activation record and enters information into its fields

parameters, return address, old stack top, saving registers, local data initialization

A return sequence restores the state of the machine

return value, restore registers, restore old stack top, branch to return address

The code in calling sequence is often divided between the caller and the callee.

14) What are return sequences?

Possible return sequence

- Callee places a return value next to the AR of the caller.
- Callee restores top-sp and other registers
- Callee branches to the return address
- Caller can copy return value into its own AR

15) What are non-local names?
In a language with nested procedures (or blocks) and static scope (lexical scope), some names are neither local nor global, they are non-local names.
procedure A
real a;

procedure B
real b;
reference a; ← non-local
end B
end A;

Example: Non-local names in C

```
main () {
    int a = 0, b=0; {
        int b = 1; {
            int a = 2;
            print(a,b); }
        { 
            int b = 3;
            print(a,b); }
        print(a,b); }
    print(a,b); }  
```

16) Explain in detail parameter passing.

Parameter Passing

Names that appear in the declaration of a procedure are formal parameters.

Variables and expressions that are passed to a procedure are actual parameters (or arguments)
Parameter passing modes
   Call by value
   Call by reference
   Copy-restore
   Call by name

Call-by-Value

The actual parameters are evaluated and their $r$-values are passed to the called procedure.

A procedure called by value can affect its caller either through nonlocal names or through pointers.

Parameters in C are always passed by value. Array is unusual, what is passed by value is a pointer.

Pascal uses pass by value by default, but $var$ parameters are passed by reference.

Call-by-Reference

Also known as call-by-address or call-by-location. The caller passes to the called procedure the $l$-value of the parameter.

If the parameter is an expression, then the expression is evaluated in a new location, and the address of the new location is passed.

Parameters in Fortran are passed by reference
   an old implementation bug in Fortran
   func(a,b) { a = b};
   call func(3,4); print(3);

Copy-Restore

A hybrid between call-by-value and call-by-reference.

The actual parameters are evaluated and their $r$-values are passed as in call-by-value. In addition, $l$-values are determined before the call.

When control returns, the current $r$-values of the formal parameters are copied back into the $l$-values of the actual parameters.

Call-by-Name

The actual parameters literally substituted for the formals. This is like a macro-
expansion or in-line expansion

Call-by-name is not used in practice. However, the conceptually related technique of in-line expansion is commonly used.

In-lining may be one of the most effective optimization transformations if they are guided by execution profiles.