CODE OPTIMIZATION

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. Compilers that apply code-improving transformations are called optimizing compilers. The process of code optimization involves:

- Eliminating the unwanted code lines
- Rearranging the statements of the code

The optimized code has the following **advantages**:

1. Optimized code has faster execution speed.
2. Optimized code utilizes the memory efficiency.
3. Optimized code gives better performance.

A transformation of a program is called **local** if it can be performed by looking only at the statements in a basic block; otherwise, it is called **global**. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

CRITERIA OF CODE OPTIMIZATION

- The optimization should capture most of the potential improvements without an unreasonable amount of efforts.
- The optimization should be such that the meaning of the source program is preserved. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the "safe" approach of

missing an opportunity to apply a transformation rather than risk changing what the program does.

- The optimization should, on average, reduce the time and space expended by the object code. Not every transformation succeeds in improving every program, occasionally an "optimization"may slow down a program slightly.
- The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. "Peephole" transformations of this kind are simple enough and beneficial enough to be included in any compiler.

BASIC OPTIMIZATION TECHNIQUES

Optimizations techniques are classified into two categories. They are

- Machine Independent Optimizations Techniques
- Machine Dependent Optimizations Techniques

1. Machine Independent Optimizations Techniques

Machine independent optimizations techniques are program transformations that improve the target code without taking into consideration any properties of the target machine. This code optimization phase attempts to improve the intermediate code to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.

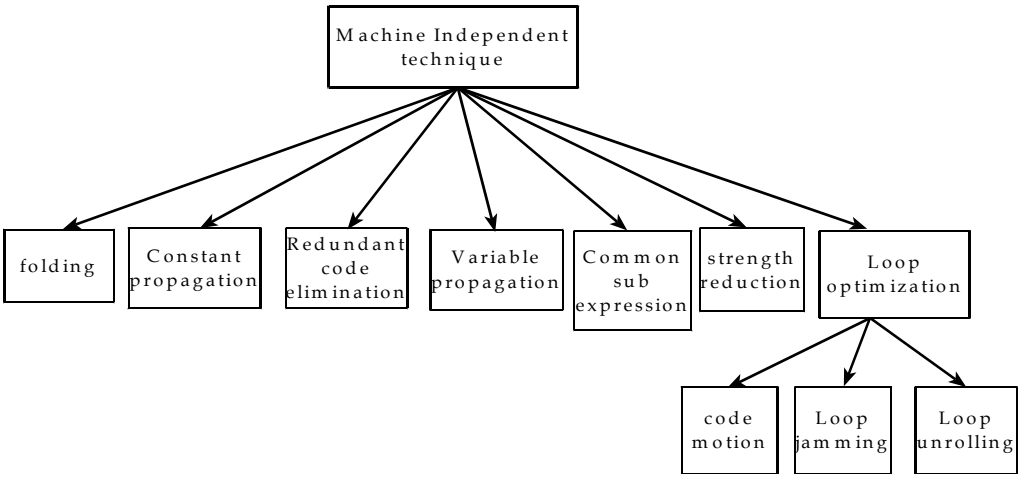

Figure: Machine independent optimization techniques classification

1.1 Constant Folding

In this technique, as the name suggests, it involves folding the constants. The expressions that contain the operands having constant values at compile time are evaluated. Those expressions are then replaced with their respective results.

Expressions with constant operands can be evaluated at compile time, thus improving run-time performance and reducing code size by avoiding evaluation at compile-time.

Example: In the code fragment below, the expression (3 + 5) can be evaluated at compile time and replaced with the constant 8.

```
int f (void)
{
  return 3 + 5;
}
```

Below is the code fragment after constant folding.

```
int f (void)
{
  return 8;
}
```

1.2 Constant Propagation

Constants assigned to a variable can be propagated through the flow graph and substituted at the use of the variable.

Example: In the code fragment below, the value of x can be propagated to the use of x.

```
x = 3;
y = x + 4;
```

Below is the code fragment after constant propagation and constant folding.

```
x = 3;
y = 7;
```

1.3 Redundant Code Elimination

In computer programming, redundant code is source code or compiled code in a computer program that is unnecessary, such as:

- Re-computing a value that has previously been calculated and is still available,
- Code that is never executed (known as unreachable ode),
- Code which is executed but has no external effect (e.g. does not change the output produced by a program; known as dead code).

Example:

x = y + z

a = 2 + y + b + z

That is reduced by,

a = 2 + x + b

1.3 Variable Propagation

Variable propagation means use of one variable instead of another.

Example:

x = r

A = pi x^2

That is reduced by,

A = pi r^2

1.4 Strength Reduction

In this technique, as the name suggests, it involves reducing the strength of expressions. This technique replaces the expensive and costly operators with the simple and cheaper ones.

Example:

Code before optimization	Code after optimization
1. B = A × 2	B = A + A
2. B=A^3	B=A*A*A

Here,

- The expression "A × 2" is replaced with the expression "A +A".
- This is because the cost of multiplication operator is higher than that of addition operator.

1.5 Common Sub-Expression Elimination

The expression that has been already computed before and appears again in the code for computation is called as **Common Sub- Expression.**

In this technique, as the name suggests, it involves eliminating the common sub expressions. The redundant expressions are eliminated to avoid their re-computation. The already computed result is used in the further program when required.

Example: x = (a + b + c) * (a + b)

Code before optimization	Code after optimization
T_1 = a + b	T_1 = a + b
T_2 = T_1 + c	T_2 = T_1 + c
T_3 = a + b // redundant expression	T_3 = T_2 * T_1
T_4 = T_2 * T_3	T_4 = x
T_5 = x	

1.6 Loop Optimization

i) Loop invariant code optimization

Loop invariant code motion is an optimization where operations inside loops are moved outside loops.

Example:

```
while(i<5000)
{
        x = i * sin(A) * cos(A);
}
```

This can be optimized as

```
t = sin(A) * cos(A);
while(i<5000)
{
        x = i * t;
}
```

ii) Loop jamming / Loop Fusion

In loop fusion method several loops are merged to one loop.

Example:

```
for i = 1 to n do
        for j = 1 to m do
                a[i, j] = 10
```

It can be written as,

```
for i = 1 to n*m do
        a[i] = 10
```

iii) Loop Unrolling

The number of jumps and tests can be reduced by writing the code two times.

Example:

```
int i=1;
while(i<=100)
{
        a[i]=b[i];
        i++;
}
```

It can be written as,

```
int i=1;
while(i<=100)
{
      a[i]=b[i];
      i++;
      a[i]=b[i];
      i++;
}
```

2. Machine Dependent Optimization Techniques

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of the memory hierarchy.

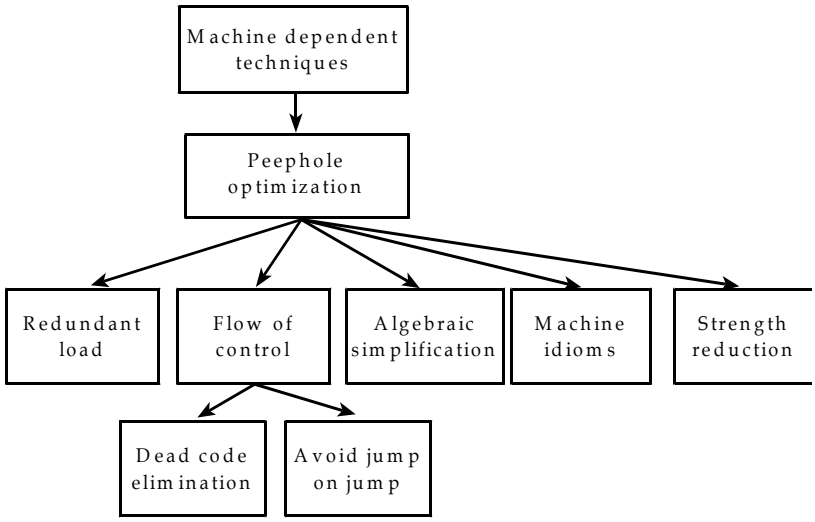

Figure: Machine dependent optimization techniques classification

2.1 Peephole Optimization

Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions (called the peephole) and replace these instructions replacing by shorter or faster sequence whenever possible. Peephole is a small, moving window on the target program.

Objectives of Peephole Optimization

The objective of peephole optimization is:

1. To improve performance
2. To reduce memory footprint
3. To reduce code size

Peephole Optimization Techniques

1. **Redundant Load and Store Elimination**

   In this technique the redundancy is eliminated.

   Example:

   Initial code

   ```
   y = x + 5;
   i = y;
   z = i;
   w = z * 3;
   ```

   Optimized code

y = x + 5;
i = y;
w = y * 3;

2. The flow of Control Optimization

i) Dead Code Elimination

In this technique, as the name suggests, it involves eliminating the dead code. The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

Example:

Code before optimization	Code after optimization

i = 0;
if (i==1)
{
a=x + 5;
}
	i = 0;

ii) Avoid jump on jump

The unnecessary jumps can eliminate in either the intermediate code or the target code by the following types of peephole optimizations.

We can replace the jump sequence.

Goto L1

. . . . . .

L1: goto L2

By the sequence,

Goto L2

. . . . . .

L1: goto L2

If there are no jumps to L1 then it may be possible to eliminate the statement L1: goto L2 provided it preceded by an unconditional jump.

3. Algebraic Simplification

Peephole optimization is an effective technique for algebraic simplification. The statements such as x = x + 0 or x = x * 1 can eliminated by peephole optimization.

4. Machine idioms

The target instruction shave equivalent machine instructions for performing some have operations.

Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.

Example:

Some machines have auto-increment or auto-decrement addressing modes. Decrement these modes can use in a code for a statement like i = i + 1.

5. Strength reduction

In this technique, as the name suggests, it involves reducing the strength of expressions. This technique replaces the expensive and costly operators with the simple and cheaper ones.

Example:

Code before optimization	Code after optimization
B = A × 2	B = A + A

Here,

- The expression "A × 2" is replaced with the expression "A +A".
- This is because the cost of multiplication operator is higher than that of addition operator.

EXERCISE

Short Answer Questions

1. What do you mean by code optimization?
2. What is the purpose of code optimization?
3. Write the advantages of optimized code.
4. Define Peephole optimization?
5. What are the objectives of Peephole Optimization?
6. Explain the criteria of code optimization.
7. Differentiate between Machine independent and machine dependent optimizations techniques.
8. The classification of code optimization. Show graphically.
9. Define renaming of temporary variables with example.
10. Explain briefly about the principal source of optimization.

Long Answer Question

1. Discuss the machine independent optimizations techniques.
2. Explain the various techniques used for loop optimization.
3. Discuss in details the peephole optimization techniques.
4. Discuss in detail the role of dead code elimination and strength reduction during code optimization of a compiler.
5. Explain in detail about optimization of basic blocks.
6. Why optimization is often required in the code generated by simple code generator? Explain the strength reduction optimization techniques.
7. What do you mean by machine independent optimization techniques? Explain the variable propagation optimization techniques.
8. Explain the flow of control optimization and machine idioms with suitable example.
9. Explain in detail the constant folding and constant propagation with example.
10. Discuss in detail about redundant code elimination and common sub-expression elimination with example.

❑❑❑