CODE OPTIMIZATION & TYPES

Noor Mahammad Sk

Code Optimization

- **Goals of code optimization**:
- Remove redundant code without changing the meaning of program.

- **Objective**:
- Reduce execution speed
- Reduce code size

- Achieved through code transformation while preserving semantics

Code Optimization

- The method that is used for the modification of codes in order to improve the efficiency and quality of the code.

As a result of optimization,

- A program may become lesser in size,
- Consume lesser memory,
- Execute more rapidly, or
- Performs fewer operations (input/output).

- An optimized program should possess exactly the same outputs and side effects as that of its non-optimized program.

Code Optimization

- Optimization can also be referred to as a *program transformation technique*, performed either by optimizers or programmers.
- An optimizer is a specialized software or an inbuilt unit of a compiler which is also called as an optimizing compiler.
- Modern processors are also used to optimize the order of execution of code instructions.
- Code optimization is generally performed at the end of the developmental stage as it reduces the readability and adds code in order to increase the performance.

Types of Code Optimization

- High-level optimizations
- Intermediate level optimizations, and
- Low-level optimizations

- ***High-level optimization*** is a language dependent type of optimization that operates at a level in the close vicinity of the source code.
- High-level optimizations include inlining where a function call is replaced by the function body and partial evaluation which employs reordering of a loop, alignment of arrays, padding, layout.

Intermediate Code Optimization

- Language independent

- **The elimination of common sub Expressions**
- This type of compiler optimization probes for the instances of identical expressions by evaluating to the same value and researches whether it is valuable to replace them with a single variable which holds the computed value.

Example

Code before Optimization
S1 = 4 x i S2 = a[S1] S3 = 4 x j S4 = 4 x i **// Redundant Expression** S5 = n S6 = b[S4] + S5

Code after Optimization
S1 = 4 x i S2 = a[S1] S3 = 4 x j S5 = n S6 = b[S1] + S5

Example

- A = 2*(22.0/7.0)*r
- A = 6.2857 * r

- x = 12.4
- y = x/2.3
- Evaluate x/2.3 as 12.4/2.3

Example

Original Code

```
a = (b + c) * m;
x = b + c;
y = (b + c) * z;
```

Optimized Code

```
T1 = b + c;
a = T1 * m;
x = T1;
y = T1 * z;
```

Intermediate Code Optimization

- **Constant propagations**
- Here, expressions which can be evaluated at compile time are identified and replaced with their values.

Example

- pi = 3.14
- radius = 10
- Area of circle = pi x radius x radius

- Here, this technique will substitute the value of the variables ‘pi’ and ‘radius’ at the compile time
- It will evaluate the expression 3.14 x 10 x 10 at the compile time which will save the time during the program execution.

Intermediate Code Optimization

- **Jump threading**
- This involves an optimization of jump directly into a second one.
- The second condition is eliminated if it is an inverse or a subset of the first which can be done effortlessly in a single pass through the program.
- Acyclic chained jumps are followed till the compiler arrives at a fixed point.

Example

```
10.       a = SomeNumber();
20.     IF a > 10 GOTO 50
...
50. IF a > 0 GOTO 100
...
```

- The jump on line 50 will always be taken if the jump on line 20 is taken.
- Therefore the jump on line 20 may safely be modified to jump directly to line 100.

Intermediate Code Optimization

- **Loop invariant code motion**
- This is also known as hoisting or scalar promotion.
- A loop invariant contains expressions that can be taken outside the body of a loop without any impact on the semantics of the program.
- The above-mentioned movement is performed automatically by loop invariant code motion.

Example

Code before Optimization

```
for ( int j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 x j;
}
```

Code after Optimization

```
x = y + z ;
for ( int j = 0 ; j < n ; j ++)
{
a[j] = 6 x j;
}
```

Example

```
a = 200;
 while(a>0)
 {
    b = x + y;
    if (a % b == 0}
    printf(“%d”, a);
  }
```

```
//This code optimized as
a = 200;
b = x + y;
while(a>0)
 {
    if (a % b == 0}
    printf(“%d”, a);
  }
```

Intermediate Code Optimization

- **Dead code elimination**
- Here, as the name indicates, the codes that do not affect the program results are eliminated.
- It has a lot of benefits including reduction of program size and running time.
- It also simplifies the program structure.
- Dead code elimination is also known as DCE, dead code removal, dead code stripping, or dead code strip.

Example

c = a * b

x = a

till

d = a * b + 4

//After elimination :

c = a * b

till

d = a * b + 4

Example

Code before Optimization

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

Code after Optimization
i = 0 ;

Intermediate Code Optimization

- **Strength reduction**
- This compiler optimization replaces expensive operations with equivalent and more efficient ones, but less expensive.
- For example, replace a multiplication within a loop with an addition.

Example

```
i = 1
While(i<10)
{
        y = i * 4;
        i++;
}
```

```
//After Reduction
i = 1
t = 4

while(t<40)
{
        y = t;
        t = t + 4;
}
```

Example

Code before Optimization
B = A x 2

Code after Optimization
B = A + A

Low-level optimization

- Highly specific to the type of architecture
- **Register allocation** – Here, a big number of target program variables are assigned to a small number of CPU registers.
- This can happen over a local register allocation or a global register allocation or an inter-procedural register allocation.
- **Instruction Scheduling** – This is used to improve an instruction level parallelism that in turn improves the performance of machines with instruction pipelines.
- It will not change the meaning of the code but rearranges the order of instructions to avoid pipeline stalls.
- Semantically ambiguous operations are also avoided.

Low-level optimization

- **Floating-point units utilization** – Floating point units are designed specifically to carry out operations of floating point numbers like addition, subtraction, etc.
- The features of these units are utilized in low-level optimizations which are highly specific to the type of architecture.
- **Branch prediction** – Branch prediction techniques help to guess in which way a branch functions even though it is not known definitively which will be of great help for the betterment of results.
- **Peephole and profile-based optimization** – *Peephole optimization* technique is carried out over small code sections at a time to transform them by replacing with shorter or faster sets of instructions.
- This set is called as a peephole.

Machine-independent optimization and machine-dependent optimization

- ***Machine-independent optimization*** phase tries to improve the intermediate code to obtain a better output.
- The optimized intermediate code does not involve any absolute memory locations or CPU registers.
- ***Machine-dependent optimization*** is done after generation of the target code which is transformed according to target machine architecture.
- This involves CPU registers and may have absolute memory references.
- To conclude, code optimization is certainly required as it provides a cleaner code base, higher consistency, faster sites, better readability and much more.

Unroll loop bodies into equivalent sequential code:

Original code

```
for i in 1..10 loop
        a[i][i] = 2*i;
end loop
```

Optimized code

```
A[1][1] := 2;
A[2][2] := 4;
...
```

Loop unrolling:

Original Code

```
for i:=1 to 20 do
begin
        for j:= 1 to 2 do
                write (x[i, j])
end
```

Optimized Code

```
for i:=1 to 20 do
begin
        write (x[i, j],  x[i,2])
end
```

If Optimization

```
void f (int *p)
{
        if (p)
        {
                g(1);
                if (p)
                g(2);
                g(3);
        }
return;
}
```

```
void f (int *p)
{
        if (p)
        {
                g(1);
                g(2);
                g(3);
        }
return;
}
```

Example

```
void f (int *p)
{
        if (p) g(1);
        if (p) g(2);
        return;
}
```

```
void f (int *p)
{
        if (p)
        {
                g(1);
                g(2);
        }
return;
}
```