UNIT V

Optimization & Code Generation

Optimization is the final stage of compiler, though it is optional. This is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed. In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.

A code optimizing process must follow the three rules given below:

1. The output code must not, in any way, change the meaning of the program.
2. Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
3. Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

1. At the beginning, users can change/rearrange the code or use better algorithms to write the code.
2. After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
3. While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Main Types of Code Optimization are -

1. 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 in lining where a function call is replaced by the function body and partial evaluation which employs reordering of a loop, alignment of arrays, padding, layout, and elimination of tail recursion.

Most of the code optimizations performed fall under **intermediate code optimization** which is language independent. This includes:

1. **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.
2. **Constant propagations** – Here, expressions which can be evaluated at compile time are identified and replaced with their values.
3. **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.
4. **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.
5. **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.

6. **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.

Low-level Optimization is highly specific to the type of architecture. This includes the following:

1. **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.
2. **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.
3. **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.
4. **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.
5. **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. *Profile-based optimization* is performed on a compiler which has difficulty in the prediction of likely outcome of branches, sizes of arrays, or most frequently executed loops. They provide the missing information, enabling the compilers to decide when needed.

Optimization can broadly be categorized into two- Machine Independent and Machine dependent.

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.

Machine Independent Optimization

- In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
        do
        {
    item = 10;
  value = value + item;
        } while(value<100);
```

- This code involves repeated assignment of the identifier item, which if we put this way:

```
 Item = 10;
 do
 {
value = value + item;
  } while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

Machine Dependent Optimization

- 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 memory hierarchy.

Machine independence includes two types

i. Function Preserving
ii. Loop optimization

- **Function preserving**

Common Sub Expression Elimination

The expression that produces the same results should be removed out from the code

Example

BO	**AO**
T1 = 4+i	T1 = 4+i
T2 = T2 +T1	T2 = T2 +T1
T3 = 4 * i	T4 = T2 + T1
T4 = T2 + T3	

Constant folding

If expression generates a constant value then instead of performing its calculation again and again we calculate it once and assign it.

Example

BO	**AO**
T1 = 512	T1 = 2.5

- **Copy Propagation**

In this propagation a F value is been send to G and G value is been send to H We can eliminate G variable directly assigning the value of F to H.

BO	**AO**
T1 = X	T2 = T3 + T2
T2 = T3 + T2	T3 = T1
T3 = X	

- **Dead Code Elimination**

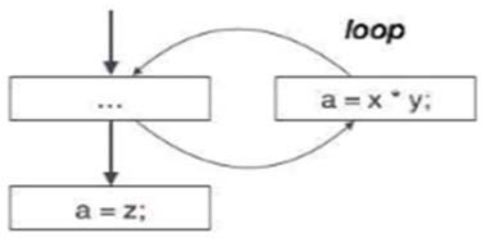
  Dead code is one or more than one code statements, which are:

  - Either never executed or unreachable,
  - Or if executed, their output is never used.

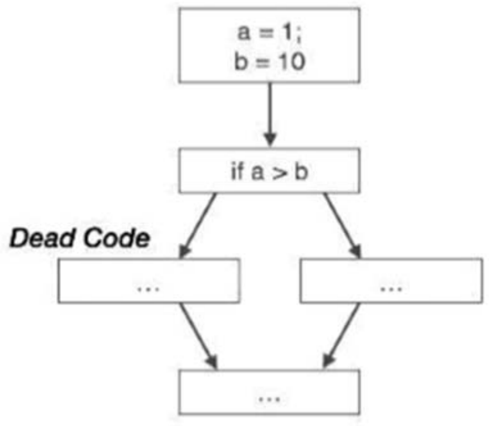
  Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Partially dead code Elimination

- There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not.
- Such codes are known as partially dead-code.



- The above control flow graph depicts a chunk of program where variable ‘a’ is used to assign the output of expression ‘x * y’.
- Let us assume that the value assigned to ‘a’ is never used inside the loop.
- Immediately after the control leaves the loop, ‘a’ is assigned the value of variable ‘z’, which would be used later in the program.
- We conclude here that the assignment code of ‘a’ is never used anywhere, therefore it is eligible to be eliminated.
- Likewise, the picture below depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.



3. Loop Optimization

We are going to perform optimization on loops.

- Code Motion

  It specifies on a condition if we perform some operations to be carried out and then compare for a condition.

  Instead of that perform the calculation outside the loop and assign a value in the calculation.

BO	**AO**

While(i < = limit-2)
{
…..
……
…
}
	t1 = limit – 2
	While (i< =t1)
	{
	…………………
	…………
	…………..
	}

- **Strength Reduction**

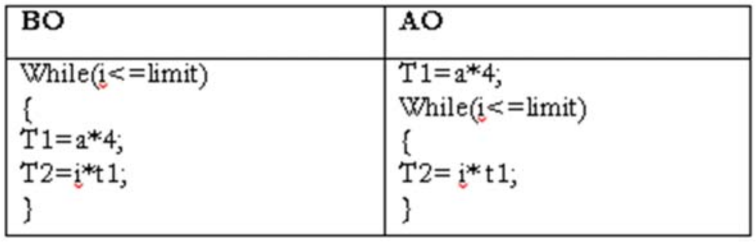
  It specifies the operators such as multiplication and division can be replaced by a addition and subtraction respectively.

  The multiplication operator can be easily replaced by left shift operator a<<1 Division can be replaced by a a>>1 operator.

BO	**AO**
T1 = a * 2	a<<1
T1 = a / 2	a >> 1

- **Frequency Reduction**

In this case if a expression inside a loop is not dynamically affected by a loop we calculate it outside the loop and use the value inside the loop.



BO	AO

While(i<=limit)
{
T1=a*4;
T2=i*t1;
}
	T1=a*4;
	While(i<=limit)
	{
	T2= i*t1;
	}

- **Loop Distribution**

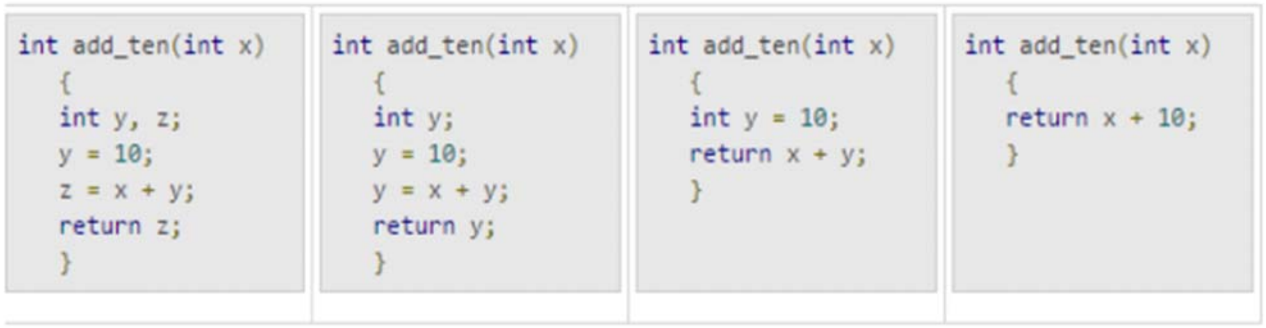
It specifies the values in a particular loop to be assigned to a array keeps of varing i.e the array location in which a loop need to be work again and again. We can use two different loops which allows loop distribution

Peephole Optimization

- This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand.
- These methods can be applied on intermediate codes as well as on target codes.
- A bunch of statements is analyzed and are checked for the following possible optimization:

1. Redundant instruction elimination

- At source code level, the following can be done by the user:



```
int add_ten(int x)
   {
   int y, z;
   y = 10;
   z = x + y;
   return z;
   }
```

```
int add_ten(int x)
   {
   int y;
   y = 10;
   y = x + y;
   return y;
   }
```

```
int add_ten(int x)
   {
   int y = 10;
   return x + y;
   }
```

```
int add_ten(int x)
   {
   return x + 10;
   }
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

- MOV x, R0
- MOV R0, R1

We can delete the first instruction and re-write the sentence as:

MOV x, R1

2. Unreachable code

- Unreachable code is a part of the program code that is never accessed because of programming constructs.
- Programmers may have accidently written a piece of code that can never be reached.

Example:

```
 void add_ten(int x)
{
return x + 10;
printf("value of x is %d", x);
}
```

- In this code segment, the printf statement will never be executed as the program control returns back before it can execute, hence printf can be removed.

• In this code segment, the printf statement will never be executed as the program control returns back before it can execute, hence printf can be removed.

3. Flow of control optimization

- There are instances in a code where the program control jumps back and forth without performing any significant task.
- These jumps can be removed. Consider the following chunk of code:

```
...
MOV R1, R2


GOTO L1
 ...
 L1:   GOTO L2
L2:   INC R1
```

- In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
          ...
          MOV R1, R2
          GOTO L2
          ...
           L2:   INC R1
```

4. Algebraic expression simplification

- There are occasions where algebraic expressions can be made simple. For example, the expression a = a + 0 can be replaced by a itself and the expression a = a + 1 can simply be replaced by INC a.

5. Strength reduction

- There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result.
- For example, x * 2 can be replaced by x << 1, which involves only one left shift. Though the output of a * a and a2 is same, a2 is much more efficient to implement.

6. Accessing machine instructions

- The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently.
- If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient results.

CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

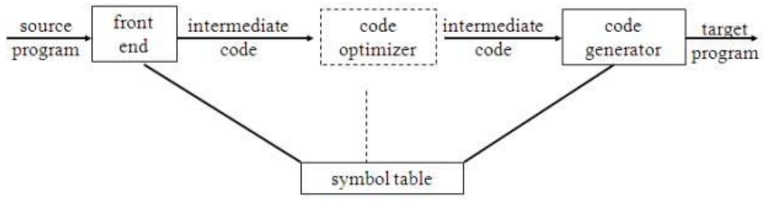


Fig. 4.1 Position of code generator

ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase:

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

The input to the code generation consists of the intermediate representation of the source program produced by front end, together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate representation can be :

a. Linear representation such as postfix notation
b. Three address representation such as quadruples
c. Virtual machine representation such as stack machine code
d. Graphical representations such as syntax trees and DAGs

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

The output of the code generator is the target program. The output may be :

a. Absolute machine language

- It can be placed in a fixed memory location and can be executed immediately. b. Relocatable machine language
- It allows subprograms to be compiled separately.

c. Assembly language

- Code generation is made easier.

3. Memory management:

Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.

It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.

Labels in three-address statements have to be converted to addresses of instructions. For example,

j:gotoigenerates jump instruction as follows:

* if i < j, a backward jump instruction with target address equal to location of code for quadruple i is generated.

* if i > j, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to i are filled.

4. Instruction selection:

The instructions of target machine should be complete and uniform.

Instruction speeds and machine idioms are important factors when efficiency of target program is considered.

The quality of the generated code is determined by its speed and size.

The former statement can be translated into the latter statement as shown below:

```
a:=b+c
d:=a+e
(a)
    MOV b,R0
    ADD c,R0
    MOV R0,a
(b)
    MOV a,R0
    ADD e,R0
    MOV R0,d
```

5. Register allocation

Instructions involving register operands are shorter and faster than those involving operands in memory. The use of registers is subdivided into two sub problems :

1. Register allocation - the set of variables that will reside in registers at a point in the program is selected.
2. Register assignment - the specific register that a value picked•
3. Certain machine requires even-odd register pairs for some operands and results. For example, consider the division instruction of the form :D x, y

where, x - dividend even register in even/odd register pair y-divisor

even register holds the remainder
odd register holds the quotient

6. Evaluation order

The order in which the computations are performed can affect the efficiency of the target code.

Some computation orders require fewer registers to hold intermediate results than others.

Questions:

1. Discuss the Necessity of optimization in compilation?
2. Specify the necessary and sufficient conditions for –
   a) Constant Propagation
   b) Dead code elimination
   c) Loop optimization
3. Explain the role of code optimization in Compiler Designing ? Explain Peephole optimization along with an example.
4. Discuss the issues in Code Generation phase.