Code Generation and Code Optimization

code generation is part of the process chain of a compiler and converts intermediate representation of source code into a form (e.g., machine code) that can be readily executed by the target system.

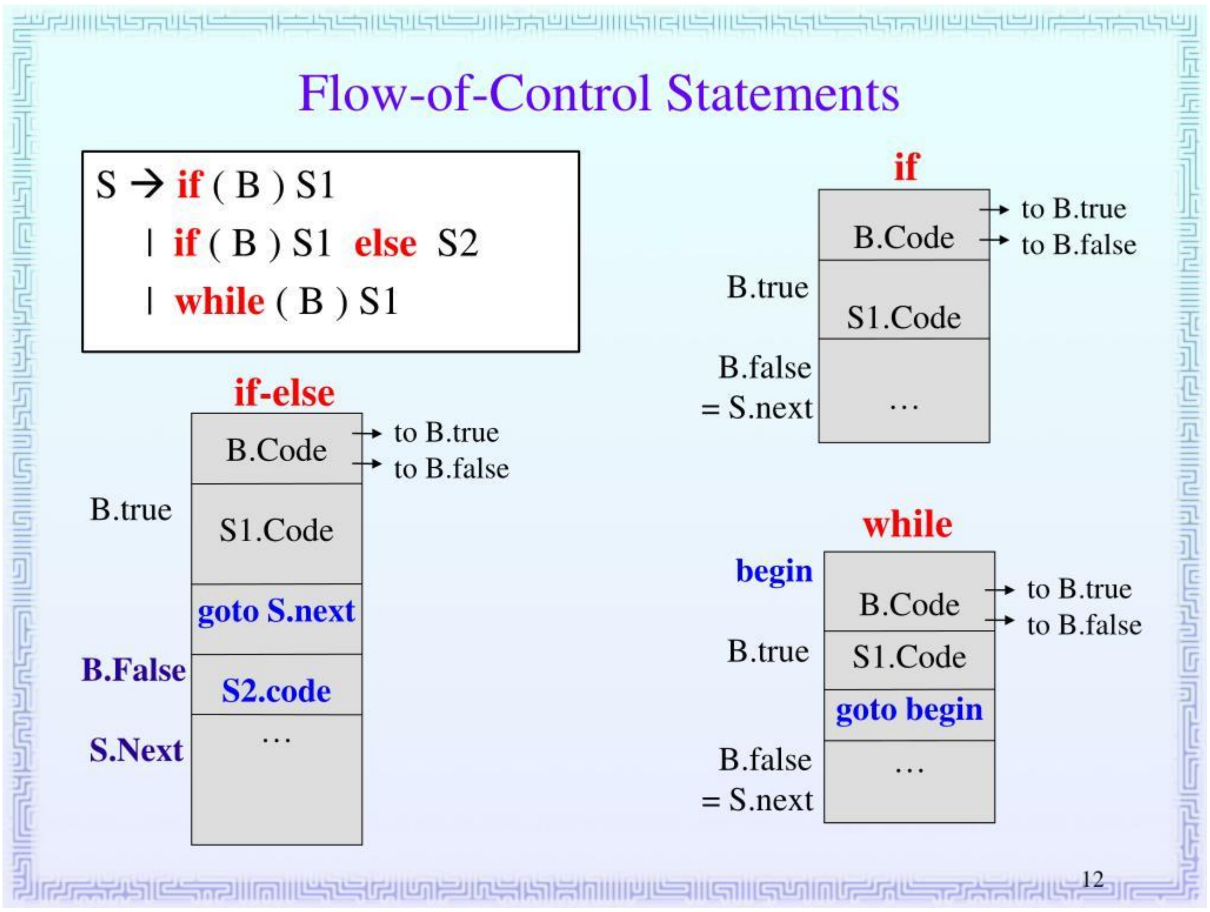
Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

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:

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

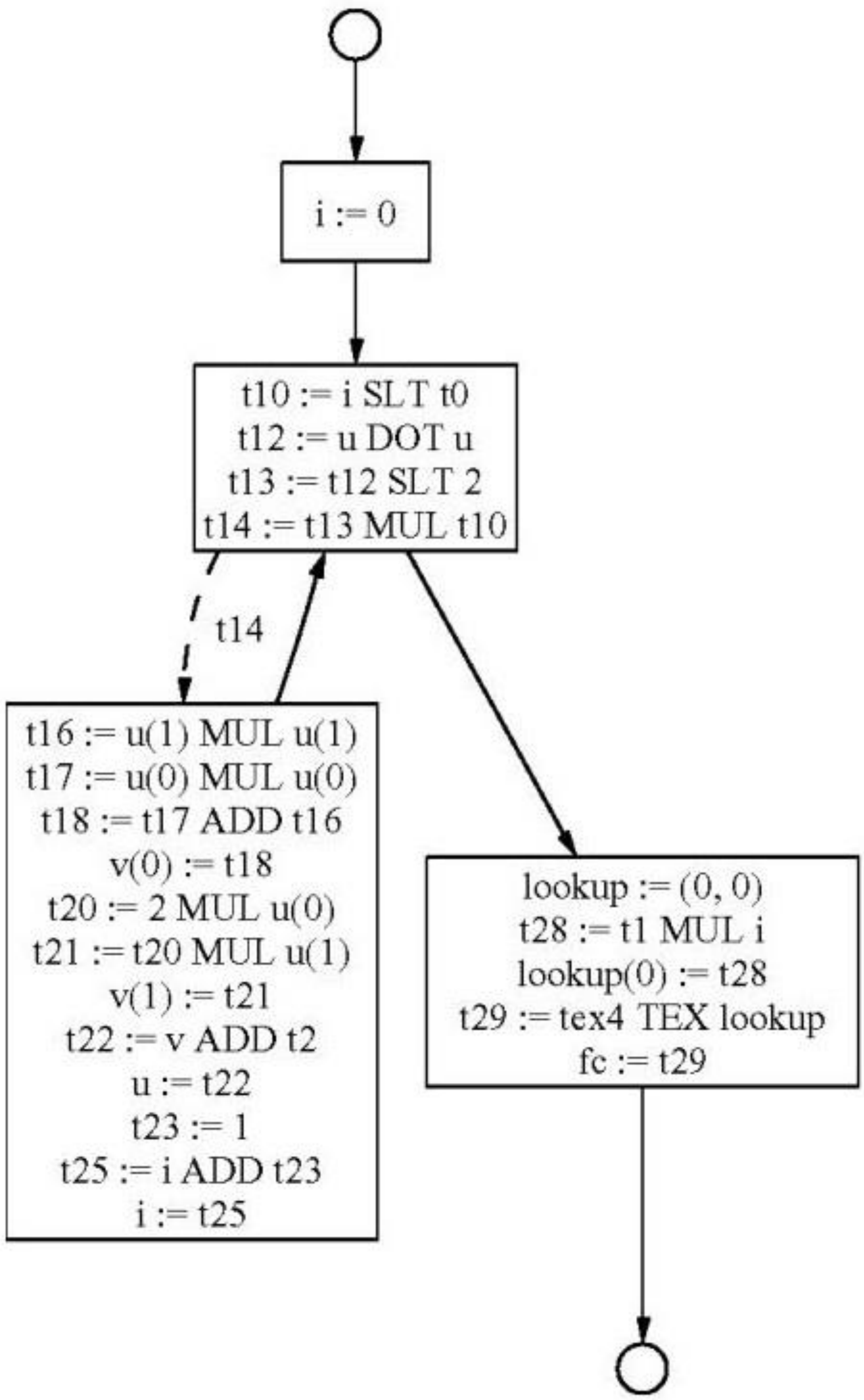
Control-flow

Control flow in compiler design refers to the representation and translation of statements that affect the order of execution of instructions. Control flow can be represented by a flow graph, which shows the connections among basic blocks of intermediate code. A basic block is a sequence of instructions that has only one entry point and one exit point[2]. Control flow can be implemented by inheriting a label next that marks the first instruction after the code for a statement.



Flow-of-Control Statements

$$
\begin{aligned}
S \rightarrow\ & \textbf{if}\ (\ B\ )\ S1 \\
\mid\ & \textbf{if}\ (\ B\ )\ S1\ \ \textbf{else}\ \ S2 \\
\mid\ & \textbf{while}\ (\ B\ )\ S1
\end{aligned}
$$



```
i := 0
```

```
t10 := i SLT t0
t12 := u DOT u
t13 := t12 SLT 2
t14 := t13 MUL t10
```

```
t16 := u(1) MUL u(1)
t17 := u(0) MUL u(0)
t18 := t17 ADD t16
v(0) := t18
t20 := 2 MUL u(0)
t21 := t20 MUL u(1)
v(1) := t21
t22 := v ADD t2
u := t22
t23 := 1
t25 := i ADD t23
i := t25
```

```
lookup := (0, 0)
t28 := t1 MUL i
lookup(0) := t28
t29 := tex4 TEX lookup
fc := t29
```

B1

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

B2

```
y = x;
x++;
```

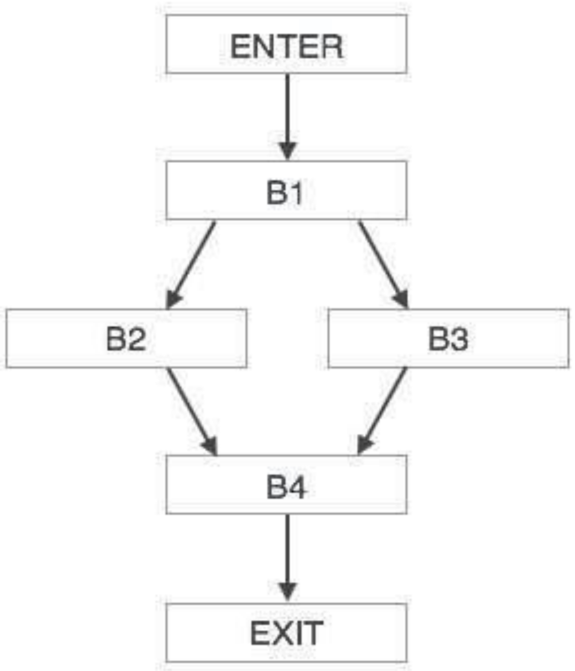
B3

```
y = z;
z++;
```

B4

```
w = x + z;
```

Basic Blocks



Flow Graph

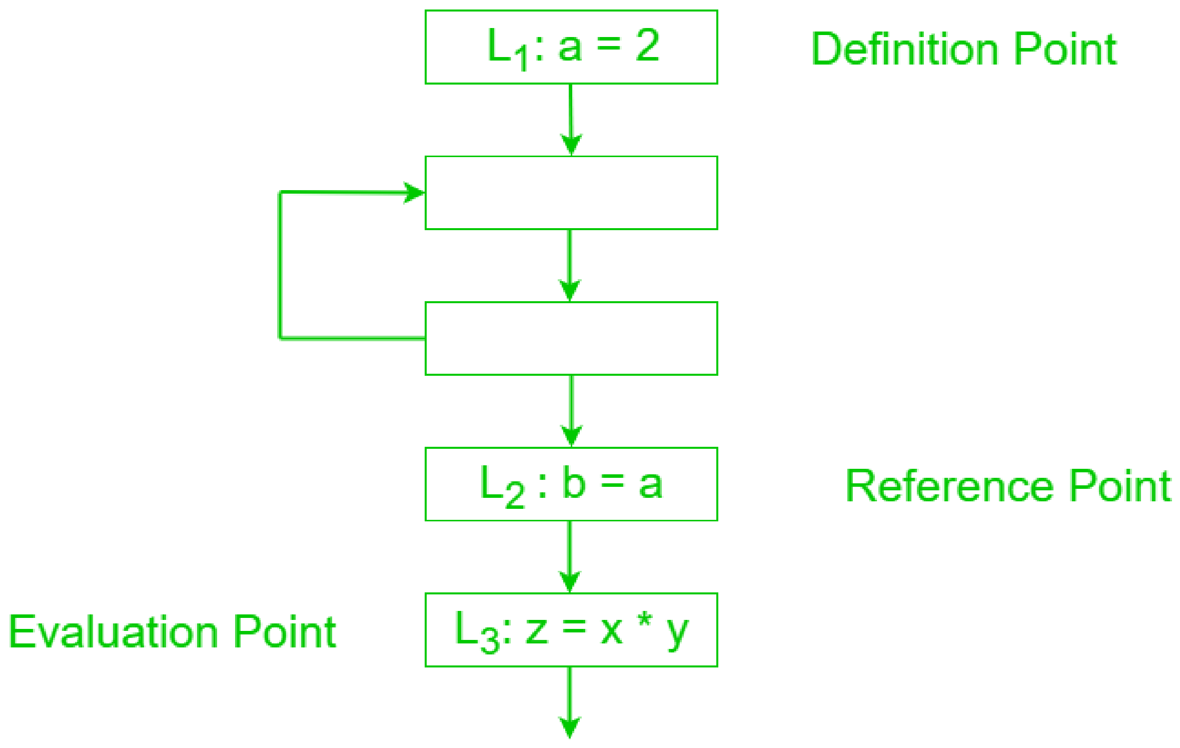
Data-flow analysis

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control-flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

A simple way to perform data-flow analysis of programs is to set up data-flow equations for each node of the control-flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fixpoint.

Basic Terminologies –

- **Definition Point:** a point in a program containing some definition.
- **Reference Point:** a point in a program containing a reference to a data item.
- **Evaluation Point:** a point in a program containing evaluation of expression.



Local Optimization

Local optimization is the process of improving the intermediate code or target code within a basic block, which is a sequence of instructions with only one entry and exit point. Global optimization is the process of improving the code across basic blocks or procedures, taking into account the control flow of the program. Local optimization is easier to implement and less costly than global optimization, but it may miss some opportunities for further improvement[1]

Why Optimize?

Optimizing an algorithm is beyond the scope of the code optimization phase. So the program is optimized. And it may involve reducing the size of the code. So optimization helps to:

- Reduce the space consumed and increases the speed of compilation.
- Manually analyzing datasets involves a lot of time. Hence we make use of software like Tableau for data analysis. Similarly manually performing the optimization is also tedious and is better done using a code optimizer.
- An optimized code often promotes re-usability.

Code Optimization is done in the following different ways:

1. Machine Independent Optimization: It positively affects the efficiency of intermediate code by transforming a part of code that does not employ hardware parts. It usually optimises code by eliminating tediums and removing unneeded code.
   a. Compile Time Evaluation
   b. Common Subexpression Elimination
   c. Variable Propagation
   d. Dead Code Elimination
   e. Code Movement
   f. Strength Reduction

   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 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.

Examples of machine-dependent code include code written in low-level languages such as assembly or machine code, which is specific to a particular type of processor. Additionally, code that makes use of specific hardware features or peripherals may also be machine dependent.

Global optimization

The local optimization involves the statements within a sil4e block (or basic block). All other optimizations are called global optimizations- The global optimization allows the compiler/optimizer to look at the overall program and determine how best to apply the desired optimization leveL The global optimization is generally performed by using data flow analysis the transmission of used relationships from all parts of the program to the places where the information can be of use.

- Global Optimization: Locate the optima for an objective function that may contain local optima.

  An objective function always has a global optima (otherwise we would not be interested in optimizing it), although it may also have local optima that have an objective function evaluation that is not as good as the global optima.

  The global optima may be the same as the local optima, in which case it would be more appropriate to refer to the optimization problem as a local optimization, instead of global optimization.

  The presence of the local optima is a major component of what defines the difficulty of a global optimization problem as it may be relatively easy to locate a local optima and relatively difficult to locate the global optima.

  Global optimization or global search refers to searching for the global optima.
  A global optimization algorithm, also called a global search algorithm, is intended to locate a global optima. It is suited to traversing the entire input search space and getting close to (or finding exactly) the extreme of the function.

  Global optimization is used for problems with a small number of variables, where computing time is not critical, and the value of finding the true global solution is very high.

  Global search algorithms may involve managing a single or a population of candidate solutions from which new candidate solutions are iteratively generated and evaluated to see if they result in an improvement and taken as the new working state.

  There may be debate over what exactly constitutes a global search algorithm; nevertheless, three examples of global search algorithms using our definitions include:

- Genetic Algorithm

- Simulated Annealing
- Particle Swarm Optimization

Now that we are familiar with global and local optimization, let’s compare and contrast the two.

Loop Optimization

Loop optimization is most valuable machine-independent optimization because program's inner loop takes bulk to time of a programmer.

If we decrease the number of instructions in an inner loop then the running time of a program may be improved even if we increase the amount of code outside that loop.

For loop optimization the following three techniques are important:

4.1M

704

Polymorphism in Java | Dynamic Method Dispatch

Next

Stay

1. Code motion
2. Induction-variable elimination
3. Strength reduction

1.Code Motion:

Code motion is used to decrease the amount of code in loop. This transformation takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.

For example

In the while statement, the limit-2 equation is a loop invariant equation.

```
while (i<=limit-2)     /*statement does not change limit*/
After code motion the result is as follows:
        a= limit-2;
        while(i<=a)    /*statement does not change limit or a*/
```

2.Induction-Variable Elimination

Induction variable elimination is used to replace variable from inner loop.

It can reduce the number of additions in a loop. It improves both code space and run time performance.

In this figure, we can replace the assignment t4:=4*j by t4:=t4-4. The only problem which will be arose that t4 does not have a value when we enter block B2 for the first time. So we place a relation t4=4*j on entry to the block B2.

3. Reduction in Strength

- Strength reduction is used to replace the expensive operation by the cheaper once on the target machine.
- Addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop.
- Multiplication is cheaper than exponentiation. So we can replace exponentiation with multiplication within the loop.

Example:

```
while (i<10)
      {
j= 3 * i+1;
a[j]=a[j]-2;
i=i+2;
      }
```

After strength reduction the code will be:

```
s= 3*i+1;
    while (i<10)
     {
         j=s;
         a[j]= a[j]-2;
         i=i+2;
         s=s+6;
     }
```

In the above code, it is cheaper to compute s=s+6 than j=3 *i

Peephole Optimization

Peephole Optimization in compiler design is a local and low-level optimization technique employed by compilers to examine a small window or "peephole" of instructions in the generated code. By analyzing this limited set of instructions, the compiler seeks to identify and eliminate redundant or inefficient code sequences, replacing them with more optimal alternatives

Peephole Optimization Techniques

Here are some of the commonly used peephole optimization techniques:

Constant Folding

Constant folding is one of the peephole optimization techniques that involves evaluating constant expressions at compile-time instead of run-time. This optimization technique can significantly improve the performance of a program by reducing the number of computations performed at run-time.

Here is an example of Constant folding:

Initial Code:

```
int x = 10 + 5;

int y = x * 2;
```

Optimized Code:

```
int x = 15;

int y = x * 2;
```

Explanation: In this code, the expression 10 + 5 is a constant expression, which means that its value can be computed at compile-time. Instead of computing the value of the expression at run-time, the compiler can replace the expression with its computed value, which is 15.

Strength Reduction

Strength reduction is one of the peephole optimization techniques that aims to replace computationally expensive operations with cheaper ones, thereby improving the performance of a program.

Here is an example of strength reduction:

Initial Code:

```
int x = y / 4;
```

Optimized Code:

```
int x = y >> 2;
```

Explanation: In this code, the expression y / 4 involves a division operation, which is computationally expensive. So, we can replace this with a shift right operation, as bit-wise operations are generally faster.

Instruction Scheduling

When instructions are handled in parallel, it is required to detect and resolve dependencies between instructions. It can generally discuss dependency detection and resolution as it associates to processor classes and the processing functions contained independently.

An instruction is resource-dependent on an issued instruction if it needed a hardware resource that can be utilized by a previously issued instruction. If, for instance, only a single non-pipelined division unit is accessible, as in general in ILP-processors, thus in the code sequence the second division instruction is resource-dependent on the first one and cannot be implemented in parallel.

Resource dependencies are constraints generated by multiple resources including EUs, buses, or buffers. They can lower the degree of parallelism that can be managed at multiple phases of execution, including instruction decoding, issue, renaming, and execution, etc.

There are two basic approaches as static and dynamic. **Static detection** and resolution are adept by the compiler, which prevents dependencies by reordering the code. Thus the output of the compiler is reordered into dependency-free code. VLIW processors continually expect dependency-free code, whereas pipelined and superscalar processors generally do not.

In contrast, **dynamic detection** and resolution of dependencies are implemented by the processor. If dependencies have to be recognized in connection with instruction issues, the processor generally maintains two gliding windows.

There are several types of instruction scheduling:

1. Local (basic block) scheduling: instructions can't move across basic block boundaries.
2. Global scheduling: instructions can move across basic block boundaries.
3. Modulo scheduling: an algorithm for generating software pipelining, which is a way of increasing instruction level parallelism by interleaving different iterations of an inner loop.
4. Trace scheduling: the first practical approach for global scheduling, trace scheduling tries to optimize the control flow path that is executed most often.
5. Superblock scheduling: a simplified form of trace scheduling which does not attempt to merge control flow paths at trace "side entrances". Instead, code can be implemented by more than one schedule, vastly simplifying the code generator.