UNIT-5

CODE OPTIMIZATION& GENERATION

SIVA KUMAR RONANKI

CODE OPTIMIZATION

- Introduction
- Principles sources of optimization

CODE GENERATION

- Issues in the design of code generator
- Target languages
- Basic blocks
- Flow graphs
- Peephole optimization

Introduction

- The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result.

- Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

Objectives:

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

Types of Code Optimization :

1)Machine Independent Optimization:

- It is also called **Platform Independent techniques**
- 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.

2)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.

Platform dependent techniques

- Peephole optimization
- Instruction level parallelism
- Data level parallelism
- Cache optimization
- Redundant resources

Phases of Optimization:

There are generally two phases of optimization:

- **Global Optimization:**
  Transformations are applied to large program segments that include functions, procedures and loops.
- **Local Optimization:**
  Transformations are applied to small blocks of statements. The local optimization is done prior to global optimization.

PRINCIPAL SOURCES OF OPTIMIZATION

- 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.

Function-Preserving Transformations:

- There are a number of ways in which a compiler can improve a program without changing the function it computes.

Examples:

- Common sub expression elimination
- Copy propagation,
- Dead-code elimination
- Constant folding

i)Common Sub expressions elimination:

Consider the sequence of statements:

a = b + c

b = a - d

c = b + c

d = a – d

Since the second and fourth expressions compute the same expression, the code can be transformed as:

a = b + c

b = a - d

c = a

d = b

ii)Renaming temporary variables:

A statement t : = b + c (t is a temporary) can be changed to

u : = b + c (u is a new temporary) and all uses of this instance of **t** can be changed to **u** without changing the value of the basic block. Such a block is called a normal-form block.

iii)Dead-Code Eliminations:

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used.

Example:

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

Here, 'if' statement is dead code because this condition will never get satisfied.

iv)Interchange of statements:

Suppose a block has the following two adjacent statements:

t1 : = b +c

t2 : = x +y

We can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2.

LOOP OPTIMIZATIONS

- In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Four techniques are important for loop optimization:

- **Code motion**, which moves code outside a loop;
- **Induction-variable elimination[**Loop fusion or jamming]. which we apply to replace variables from inner loop.
- **Reduction in strength**, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.
- **Constant folding and constant propagation**

Code Motion:

- An important modification that decreases the amount of code in a loop is code motion.
- This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop.

```
a=100
while(a>0)
{
x=y+z
If(a%x==0)
Printf("%d",a)
}
```

Code Motion:

```
a=100
x=y+z
while(a>0)
{
If(a%x==0)
Printf(“%d”,a)
}
```

Induction-variable elimination (or)

Loop fusion or jamming:

```
int  i, a[100], b[100];
for(i=0,i<100,i++)
      a[i]=1;
for(i=0,i<100,i++)
      b[i]=2;
```

```
int  i, a[100], b[100];
for(i=0,i<100,i++)
      a[i]=1;
for(i=0,i<100,i++)
      b[i]=2;
```

- Even If we remove second **for** loop the value of b[i]=2 will not change

Reduction In Strength:

These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

```
c = 7;
for (i = 0; i < N; i++)
{
y[i] = c * i;
}
```

can be replaced with successive weaker additions

Reduction In Strength:

```
c = 7;
for (i = 0; i < N; i++)
{
y[i] = k;
k = k + c;
}
```

Constant folding:

It is the process of recognizing and evaluating statements with constant expressions (i=22+222+2222 to i=2466), string concatenation (“abc”+”def” to “abcdef”) and expressions with arithmetic identities (x=0; z=x*y[i]+x*2 to x=0; z=0;) at compile time rather than at execution time.

Constant propagation:

It is the process of substituting values of known constants in an expression at compile time.

Example Constant propagation:

```
int x=14;
int y=7+x/2;
return y*(28/x+2);
```

Applying constant folding and constant propagation,

```
int x=14;
int y=14;
return 56;
```

CODE GENERATION

- Issues in the design of code generator
- Target languages
- Basic blocks
- Flow graphs
- Peephole optimization

1.ISSUES IN THE DESIGN OF CODE GENERATOR

- Input to the Code Generator
- The Target Program
- Instruction Selection
- Register Allocation
- Evaluation Order

i)Input to the Code Generator

We assume, front end has

- Scanned, parsed and translate the source program into a reasonably detailed intermediate representations(IR)
- Type checking, type conversion and obvious semantic errors have already been detected
- Symbol table is able to provide run-time address of the data objects
- Intermediate representations may be
  - **Three address representation –quadruples, triples, indirect triples.**
  - **Linear representation -Postfix notations**
  - **Virtual machine representation – bytecode, Stack machine code**
  - **Graphical representation - Syntax tree, DAG**

ii)The Target Program:

- The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.
- A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
- In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.

iii)Instruction Selection:

- The code generator must map the IR program into a code sequence that can be executed by the target machine.
- The factors to be considered during instruction selection are:
  - The uniformity and completeness of the instruction set.
  - Instruction speed and machine idioms.
  - Size of the instruction set.

- Eg: for the following address code is:

  a := b + c
  d := a + e

  inefficient assembly code is:

MOV b, R_0	$R_0 \leftarrow b$
ADD c, R_0	$R_0 \leftarrow c + R_0$
MOV R_0, a	$a \leftarrow R_0$
MOV a, R_0	$R_0 \leftarrow a$
ADD e, R_0	$R_0 \leftarrow e + R_0$
MOV R_0 , d	$d \leftarrow R_0$

Here the fourth statement is redundant, and so is the third statement if 'a' is not subsequently used.

iv)Register Allocation

- Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore efficient utilization of registers is particularly important in generating good code.

- During register allocation we select the set of variables that will reside in registers at each point in the program.

- During a subsequent register assignment phase, we pick the specific register that a variable will reside in.

v)Evaluation Order:

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.

2.TARGET LANGUAGES

The Target Language:

- A Simple Target Machine Model
- Program and Instruction Costs

A Simple Target Machine Model

- Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps.
- The underlying computer is a byte-addressable machine with n general-purpose registers, R0,R1, ... ,Rn - 1.
- Most instructions consist of an operator, followed by a target, followed by a list of source operands. A label may precede an instruction.

A Simple Target Machine Model Instructions:

- Load operations
- Store operations
- Computation operations
- Unconditional jumps
- Conditional jumps

3.Basic Blocks

- Our first job is to partition a sequence of three-address instructions into basic blocks.
- We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction.
- In the absence of jumps and labels, control proceeds sequentially from one instruction to the next.

The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.

2. Any instruction that is the target of a conditional or unconditional jump is a leader.

Any instruction that immediately follows a conditional or unconditional jump is a leader.

3.Basic Blocks Example:

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

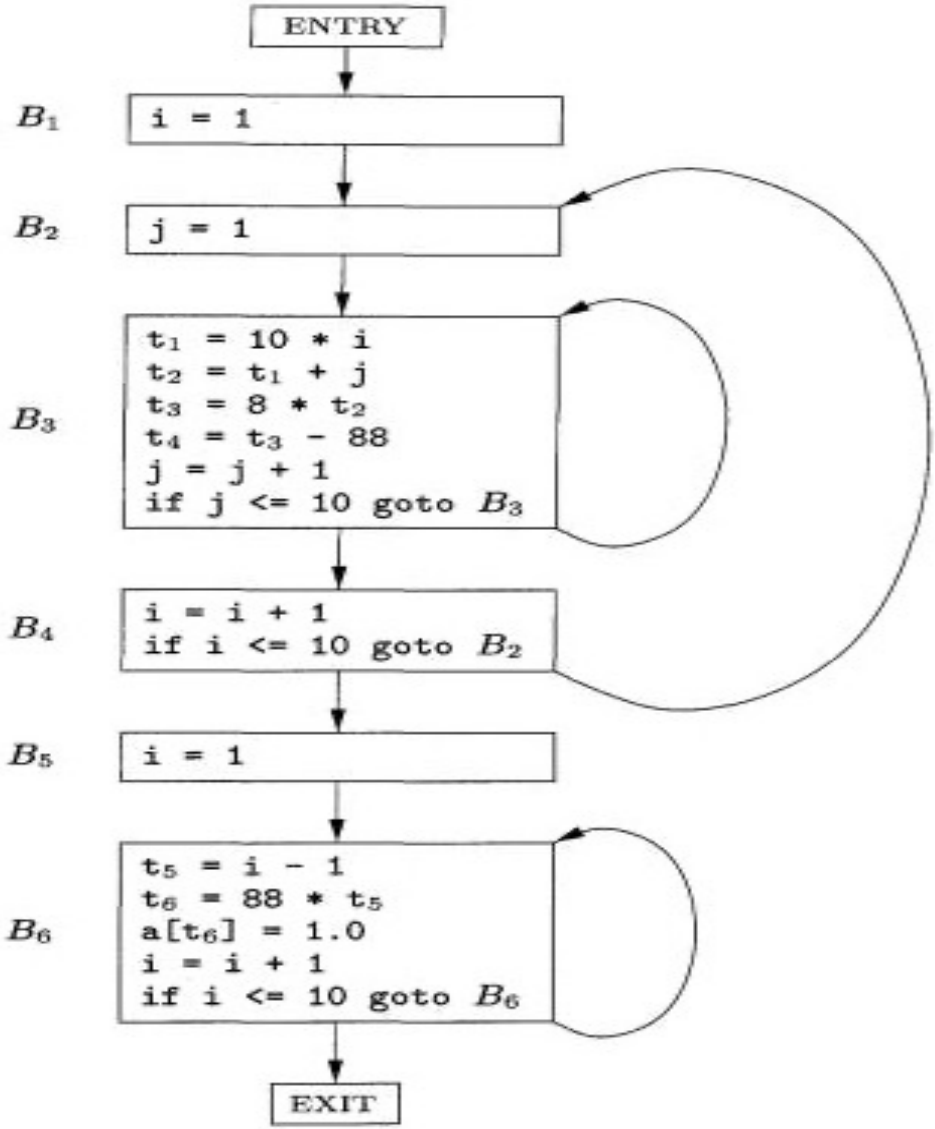
Basic Blocks explanation

- First, instruction 1 is a leader by rule (1) of Algorithm . To find the other leaders, we first need to find the jumps.
- In this example, there are three jumps, all conditional, at instructions 9, 11, and 17.
- By rule (2), the targets of these jumps are leaders; they are instructions 3, 2, and 13, respectively.
- Then, by rule (3), each instruction following a jump is a leader; those are instructions 10 and 12.
- Note that no instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.
- We conclude that the leaders are instructions 1, 2, 3, 10, 12, and 13.
- The basic block of each leader contains all the instructions from itself until just before the next leader

4.Flow Graphs

- Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph.
- The nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B.

Flow Graphs


ENTRY
B_1
i = 1
B_2
j = 1
B_3
t_1 = 10 * i
t_2 = t_1 + j
t_3 = 8 * t_2
t_4 = t_3 - 88
j = j + 1
if j <= 10 goto B_3
B_4
i = i + 1
if i <= 10 goto B_2
B_5
i = 1
B_6
t_5 = i - 1
t_6 = 88 * t_5
a[t_6] = 1.0
i = i + 1
if i <= 10 goto B_6
EXIT

5.Peephole Optimization

- A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible.
- Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

Characteristics of peephole optimizations:

- Eliminating Redundant Loads and Stores
- Eliminating Unreachable Code
- Flow-of-Control Optimizations
- Algebraic Simplification and Reduction in Strength
- Use of Machine Idioms

(i) Redundant instruction elimination:

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
First instruction can be rewritten as
MOV x,R1

(ii) Unreachable Code

- It is the removal of unreachable instructions. An unlabelled instruction immediately following an unconditional jump may be removed.
- This operation can be repeated to eliminate a sequence of instructions.

if debug == 1 goto LI

goto L2

L1: print debugging information

L2:

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of debug, the code sequence above can be replaced by

```
    if debug != 1 goto L2
    print debugging information
L2:
```

(iii)Flow-of-Control Optimizations:

- Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps.
- These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the sequence

if a < b goto LI

LI: goto L2

Can be replaced by the sequence

if a < b goto L2

(iv)Algebraic Simplification and Reduction in Strength:

- The algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as
- x = x + 0

  Or

  x = x * 1
- Similarly, reduction-in-strength transformations can be applied in the peep-hole to replace expensive operations by equivalent cheaper ones on the target machine.

(v) Use of Machine Idioms

- The target machine may have hardware instructions to implement certain specific operations efficiently.
- For example, some machines have auto-increment and auto-decrement addressing modes.
- These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing.
- These modes can also be used in code for statements like.

i:=i+1 → i++

i:=i-1 → i- -