Saturday
20-6-2020

CD Assignment - 5

UNIT-5

N.V.S.k. kalyani
17UNIA0584
3rd B.Tech CSE-C

2 Marks

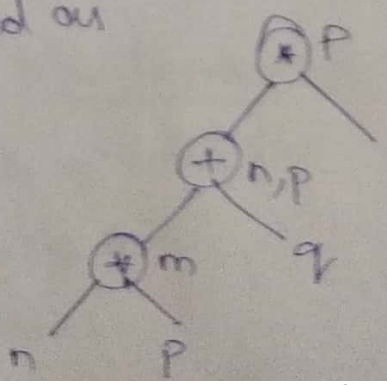
1A) Loop optimization is most valuable machine independent optimization because program's inner loop takes bulk to time of a programmer. If we decrease the no. of instructions in an inner loop then the running time of a program may be improved even if we increase the amount of code outside that loop.

2A) Direct Acyclic Graph :- It a tool that depicts the structure of basic blocks, helps to see the flow of values following among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks DAG can be un-determ : leaf node represent identifiers names.

DAG is a pictorial representation of basic block DAG is constructed for constructed for the basic block and transformation are applied.

Ex:- m : n * p where n=1, p=2, q=3
n : m+p
p : n*p
q : m+q

The DAG is constructed as



3A) Algebraic transformations can be used to simp the set of expressions computed by a basic bloc This can be done by the following:

a. Identity Rules
b. Strength Reduction
c. Constant Folding
d. Associativity and Commutativity

4A) Machine dependent code optimization perform based on the characteristics of the machine like Instruction sets used, addressing Target code.

5A) Graph coloring allocates register and attempts to minimize the cost of spills by building a Interface graph based on how a variable Interact and Interface with each other.

* Graph coloring technique of finding at least no. of colors required to color a graph such that no adjacent vertices will have the same color. The resultant graph is called as k-coloring graph.

10 marks

2) principal source of code optimization:-

There are many ways in which a compiler can improve a program few of them are shown below:

1. Function preserving transformation

Ex: common sub-expression elimination
- copy propogation
- Dead code elimination
- constant folding

2. Common sub-expression elimination
3. copy propogation
4. Dead code elimination
5. compile time evaluation
   - constant Folding
   - constant propogation
6. loop optimization
   - code motion
   - loop invariant method
   - strength Reduction

1. Function preserving transformation: There are some ways in which a compiler can improve the program efficiency without changing the function it compiles.

2. common sub-expression elimination: An occurrence of an Expression E is called as common sub-expression if E was previously computed and the value of variables was previously computed, and the value of variable in E have not changed since previous computation.

* we can avoid re-computing expression if we can use the previously computed value.

Ex:
```
a : b+c          a : b+c
c : b+c    =>    c : a
d : b+c          d : b+c
```

3. copy propogation: An Assignment statement of the form f:=g is called as copy statement.

* we can reduce the copying statements if possible so that program code will be minimized

Ex: x := y;
S := x * f(x) => S := y * f(y)

4. Dead code elimination: A variable is said to be live in a program if its value is used subsequently.

* Optimization can be performed by eliminating such dead code.

Ex:
```
x := y+1          y := 5
y := 5      =>    x := 2 * z
x := 2 * z
```

5. Compiler Time Evaluation :- It is a process of shifting of computation from runtime to compile time if ~~provides~~ possible. So that we can solve execution time.

a) constant folding : computation of constants are done at compile time instead of runtime

Ex: float d=5 length ; => float d=5, length ;
length = (22/7) * d ; length = 3.1428 * d ;

b) constant propogation : The value or variable is replaced and computation of expression is done at compile time

Ex: float pi = 3.14 , r=5, area ;
area = pi * r * r ; => float pi=3.14 ; r=5, area ;
area = 3.14 * 5 * 5 ;

6. loop optimization :- The running time of a program may be improved if we decrease the no. of instructions in the loop body. even if we increase the amount of code outside the loop.

a) code motion : which moves code outside the loop with out effecting the output.

Ex: while (i<limit-2)
{
}....

t = limit - 2
while (i<=t)
{
...
}

b. loop invarient :- which eliminates variables whose value will not be changed in the loop.

Ex:-
```
for (i=0;ic=10;i++)
{
    x=y*5;
    k=(y*5)+50;
    ....
}
```
=>
```
x=y*5
for (i=0;ic=10;i++)
{
    x=t;
    k=t+50;
    ...
}
```

c) Reduction in strength : which replaces an expensive operation by cheaper operation.

Ex:-
```
for (i=1;ic=50;i++)
{
    count=i*7;
    printf("%d",count);
}
```
=>
```
temp=7;
for(i=1;ic=50;i++)
{
    count=temp;
    temp=count+7;
    printf("%d",count);
}
```

30) Various Issues in code Generation :-

Code generation is the final phase of a compilation process.

Issues in the design of code Generator:

1. Input to the code generator :-

* It is intermediate representation of the ~~(generator is the intermediate)~~ source program produced by the front end along with information in the symbol table.

2. The Target program :- The instruction-set architecture of the target machine has a significant impact on constructing a good code generator that produces high quality machine code.

* The most common target-machine architectures are RISC, CISC, Stack based.

* The output of the code generator is an object code the object code normally comes in

=> Absolute code :-

It is a machine code that contains reference to actual address with in program space

* The generated absolute code can be placed directly in the memory and execution can be done immediately.

Re-locatable Machine code:- Execution requires linking and loading operations.

* A set of re-locatable object modules can be linked together and loaded for execution with help of linker and loader.

Assembly language code:- It produces an output is some what easier.

* After generating Asembly language code, we need to use the translator Assembler to produce an executable code.

3. Instruction selection:- The code generator must map the intermediate representation program into a code sequence that can be executed by the target machine.

4. Register allocation & assignment:- Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values.

The use of Register is often divided into two sub-problems.

1. Register allocation during which we select the set of variables that will reside in regist at each point in the program.
2. Register Assignment during which we pick the specific register in which a variable will reside in.

5. Evaluation order: The order in which computation are performed can affect the efficiency of the target code.

* Some computation orders require lower registers to hold intermediate results than others.
* However, picking a best order in the general case is a difficult NP-complete problem.

3b) Given that $x = (a+b) * (c-d) + (e/f) * (a+b)$

It's three address code is

$t_1 := a+b$
$t_2 := c-d$
$t_3 := e/f$
$t_4 := t_1 * t_2$
$t_5 := t_3 * t_1$
$t_6 := t_4 + t_5$
$x := t_6$

Its equallent target code is shown below:

Statement	Target code	Register Descriptor	Address Descriptor
$t_1 : a+b$	LD R0, a	R0 Contains a	
	ADD R0, R0, b	R_0 " t_1	t_1 is in R_0
$t_2 := c-d$	LD R1, c	R1 contains c	
	SUB R1, R1, d	R_1 " t_2	t_2 is in R_1
$t_3 := e/f$	LD R2, e	R_2 Contains e	
	DIV R2, R2, f	R_2 Contains t_3	t_3 is in R_2

$t_4 := t_1 * t_2$	MUL R_3, R_0, R_1	$R_3 \Rightarrow t_4$	t_4 is in R_3
$t_5 := t_3 * t_1$	MUL R_4, R_2, R_0	$R_4 \Rightarrow t_5$	t_5 is in R_4
$t_6 := t_4 * t_5$	ADD R_5, R_3, R_4	$R_5 \Rightarrow t_6$	t_6 is in R_5
$x := t_6$	ST x, R_5	$R_5 \Rightarrow t_6$	t_6 is in R_5 t_6 is in x