Code Optimization

Code optimization is an important part of compiler design. It is highly integrated in the code generation phase. This is because an important aspect of code optimization is the efficient use of registers and this can only be performed during code generation. It is the process of recognizing patterns & substituting them for efficient ones. Code optimization has two basic goals:-

1) The optimized code should be more efficient than the original code in terms of space and/or time required.

2) The optimized code should perform the exact same functions as the original code. This means that a correct code should not be mapped into an incorrect one while trying to optimize.

Other than the efficient use of registers, another very important part of code optimization is loop optimization. This is because 80% of the time in a program is spent in 20% of the code → that is in the loops. Thus if we optimize the code in a loop, the whole program works more efficiently.

Main types of code optimization:-

1) Removal of Dead/unreachable code.
2) Common sub expression removal
3) Loop optimization.

Common Subexpression Removal :-

For a piece of code where values of variables do not change, a compiler may use a temporary variable to save a common subexpression and reuse it as many times as needed.

Ex → $a = (b+c) - (d+e) + (b+c)$

$f = (b+c) - d$

Optimized code:-

$t_1 = b + c$

$a = t_1 - (d+e) + t_1$

$f = t_1 - d$

* This can only be done if the value of b & c does not change in between *

Removal of Dead / Unreachable code :-

A piece of code that cannot be reached by under any circumstance is called dead code or unreachable code. The code optimizer can identify such blocks of code using data flow analysis and eliminate them.

```
eg> if (a = b)
        print a
    else if (a < b)
        print a
    else if (a > b)
        print b
    else
        print 6        → unreachable code.
```

30%
40%
9%
1%
10%
10%

Loop Optimization :-

Loop optimization is a very important aspect of code optimization. In case of nested loops, if inner loops are well optimized, it makes a huge difference in the efficiency of the program.

Loop optimization involves:-
1) Loop invariants
2) Loop jamming
3) Loop unrolling
4) Induction variables.

1) Loop invariants :- Some variables do not change their value within a loop but may get redefined everytime the loop is being run. Such elements are called loop invariants. A code optimizer places loop invariants outside the loop.

eg> for i=1 to 10
{ a = 1 → loop invariant
b = i+1.
c = b+a
}

a does not change its value in the loop and hence may be placed outside the loop to enhance performance.

Date
Page 89

eg> a = 1
for i=1 to 10
{ b = i+1
c = b+a
}

2) Loop Jamming :- Often, programmers, for the sake of improving readability, use separate loops to perform the actions that can be performed in a single loop. The code optimizer identifies & fuses such loops.

eg> for i=1 to 10
A[i]=0;
for j=1 to 10
B[j]=0;

optimized code :-

for i=1 to 10
{
A[i]=0
B[i]=0
}

3) Loop Unrolling :- Sometimes, it is more efficient to not increase the number of statements in a loop and decrease the number of times loops are executed. This is because it saves the extra step of updating the loop counter. This is called loop unrolling

sum=0
Eg> for (i=1; i<10; i++) — 1 — 2 — 3 — 3x10 = 30 — 33 steps.
{
sum = sum+i — 1
}
print sum — 1

Optimized code:-

```
sum = 0                                  -1
for (i=1; i<10; i=i+2) - 2               -1
{                                        4*5 = 20   23 steps
    sum = sum + i                        -2
    sum = sum + i+1                      -2
}
print sum                                -1
```

Thus code runs 10 steps less in second case.

Date Page 91

4) Induction Variables:- Some calculations in loops cannot be directly modified and brought outside like loop invariants, but using some other variable outside the loop, its purpose can be fulfilled. Such variable which indirectly optimize a loop are called ~~loop invariant~~ induction variables.

Eg>

```
I = 1
for i = 1 to 10
{   A[i] = I * 4 - 4
    I = I + 2
}
```

the statement A[i] = I * 4 - 4
is broken into intermediate code as

```
T1 = I * 4
T2 = T1 - 4
A[i] = T2
```

If instead we declare I from 0, our program becomes:-

```
I = 0
for i = 1 to 10                  intermediate code
{                                T1 = I * 4
    A[i] = I * 4          ->     A[i] = T1
    I = I + 2
}
```

∴ around 10 steps are reduced.

Basic Blocks & Data Flow Analysis:-

An important aspect of code optimization is the identification of basic blocks.

A basic block is a block of code which start with the first statement and sequentially executes statements one after the other without branching.

Each basic block begins with a leader statement.

Identifying leader statements:-

Leader statements have the following patterns:-

1) First statement of a program is a leader
2) The target of any conditional or unconditional jump is a leader.
3) Statement following a conditional jump is a leader.

Identifying basic blocks:-

A basic block begins with a leader and ends at the last statement before the next leader.

Scanned by CamScanner

Date
Page 93

Data flow analysis:-

Data flow analysis identifies how data travels from one basic block to the next.

Data flow is shown as edges on the DFD graph.

There is an edge between two basic blocks if and only if:-

1) The last statement of previous block is a jump which goes to the leader of this basic block.
2) Blocks follow each other sequentially without any jump in between.

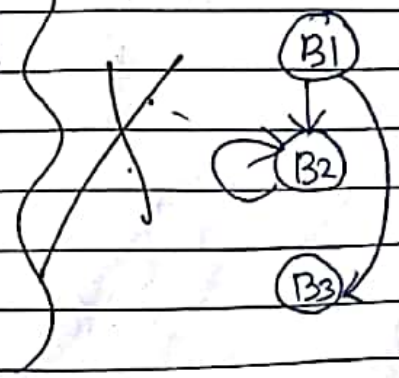
Scanned by CamScanner

Ex:- Find basic blocks and show data flow between them :-

```
int a = 5
for (i = 1; i < 10; i++)
{  a = a * i;
}
print a
```

This can be written in intermediate code as

```
1)  a = 5          ← B1
2)  i = 1
3)  if i < 10 goto 7
4)  a = a * 1      ← B2
5)    i = i + 1
6)  goto 3
7)  print a        ← B3
```



classmate
Date
Page 94

If there is a basic block which has no edge entering or leaving it, that means that data does not enter that block. Such a block of code is dead code and is removed from the program.

Code Generation

Code generation phase of the compiler takes the optimized intermediate code as input and tries to generate target code for the target machine.

The input consists of 3-address code, postfix notations and syntax trees or DAGs. It takes these inputs and tries to generate an efficient target code.

The code generator may make several intermediate codes and pass them through the code optimizer before making the final target code.

The final code generated must have the following properties:-

- It should have the same semantics as the source program and should be of high quality
- It should make optimum use of the resources of the target machine.

~~Code optimization has the~~

Code generation involves the following actions:-

1) Instruction Selection
2) Register allocation
3) Instruction ordering.

Scanned by CamScanner

classmate
Date
Page 96

Instruction Selection- This involves finding the appropriate machine language instruction to implement each line of the intermediate code

Register Allocation- This involves finding which registers are available and what to store in which register

Instruction Ordering- This involves deciding the sequence in which the instructions are to be executed.

Scanned by CamScanner

Output of Code Generator :-

The code generator produces target code which can be of 2 types :-

1) RISC
2) CISC

1) RISC – reduced instruction set computer :-

This type of machine has many registers, 3 byte addresses, simple address modes and simple instructions.

2) CISC – complex instruction set computer :-

This type of machine has few registers, 2-byte addresses, many addressing modes, many register classes and many instructions.

Register Allocation :- The code generator has to decide upon the register allocation that is to be done for each instruction in the program. Register allocation occurs in two steps :-

1) Deciding which variable needs allocation at what point in time.

2) Finding what registers are available and allocating them accordingly.

Instructions used by Code Generator :-

1) Load operation :- Loads value into specific register

LD (destination, source)

2) Computational operation :- Performs operation specified such as ADD, SUB, MUL, DIV

OP (destination, source1, source2)

3) Branch statements :-

1) Unconditional Jump →

BR <L> → Jumps to label L.

3) Branch statements :-

1) Unconditional jump →

BR <L> → jumps to label L.

2) Conditional jump →

BCond r, L - checks value in register r, if r<0, jumps to line L.

Ex:- Write target machine code for following intermediate code :-

$T_1 = a * b$
$T_2 = T_1 + c$
$d = T_2$

LD R_1, a
LD R_2, b
MUL R_2, R_1, R_2
~~ADD R2 R2 R2~~
LD R_3, c
ADD R_2, R_2, R_3
ST R_2, d

Error Handling

Error handling routines interact with all stages of compilation. Whenever the compiler faces an error, the error handler tries to repair the error and continue the process of compilation. In case of errors, a compiler is not expected to do the following:-

1) Stop entire process of compilation at first detected error.

2) Generate a wrong output

3) Initiate a system crash.

A good compiler should try to at least complete the lexical and syntactic analysis of the program so that all errors can be detected.

A more advanced compiler may try to repair the error using a number of strategies.

Whenever an error is detected, the compiler should generate an error message which will be displayed at the end of the compilation process.

Features of a good error diagnosis:-

1) The error message should pinpoint the error in terms of source code and not in terms of some internal representation.

2) The error message should be tasteful and clearly defined

3) The error message should state the error understandably and not in terms of some obscure error code

4) Error messages should not be redundant.

Lexical Phase Errors :-

Lexical phase errors occur when an invalid keyword is present or an invalid lexical syntax is generated. Lexical errors are easy to locate. The lexical analyzer may perform one of two actions when faced with an error:-

1) It may skip the invalid token and search for the next lexeme. This generates a deleted character error

2) It may try to repair the error by minimum distance matching or string matching techniques so as to avoid further errors.

Scanned by CamScanner

Date
Page 108

Minimum Distance Matching :-

For an erroneous input 'x' in the lexical analysis phase, the analyzer tries to repair it by performing one of the following error transformations on existing valid keywords so as to arrive at 'x'.

Error transformations include:-

1) Inserting a character
2) Deleting a character
3) Transposing two adjacent characters
4) Modifying a character.

Lexical phase errors can be corrected by various string matching techniques.

Strings : Parts of a string :-

A string can have the following parts :-

Prefix → removal of 0 or more letters from the end.

Suffix → removal of 0 or more letters from the start

Sub string → removal of a prefix or suffix or both.

Proper Prefix & Suffixes → All prefixes & suffixes except the string itself

Subsequence → Deletion of one or more letters from string

Scanned by CamScanner

Suffix → removal of 0 or more letters from the start

Substring → removal of a prefix or suffix or both.

Proper Prefix & Suffixes → All prefixes & suffixes except the string itself

Subsequence → Deletion of one or more letters from string

Date
Page 109

Ex:- Generate all prefixes, suffixes, substrings and subsequences of the word

HERMIONE

Prefix → H, HE, HER, HERM, HERMI, HERMIO, HERMION, HERMIONE

Suffix → HERMIONE, ERMIONE, RMIONE, MIONE, IONE, ONE, NE, E

Substring → H, E, R, M, I, O, N
HE, ER, RM, MI, IO, ON, NE
HER, ERM, RMI, MIO, ION, ONE
HERM, ERMI, RMIO, MION, IONE
HERMI, ERMIO, RMION, MIONE
HERMIO, ERMION, RMIONE
HERMION, ERMIONE
HERMIONE

Subsequence → H, E, R, M, I, O, N
HE, HR, HM, HI, HO, HN
ER, EM, EI, EO, EN, EE
RM, RI, RO, RN, RE
MI, MO, MN, ME
IO, IN, IE
ON, OE
NE
HER, HEM, HEI, HEO, HEN, HEE
(
)
/
(

* not wasting further time *

Syntactic Errors :-

Syntactic errors are the easiest to spot for the compiler. This is because syntax analysis follows strict rules of context free grammar to generate parse trees.

Whenever the parser finds itself in a position in the parse tree from which it has no legal move left, it knows it has detected an error.

The parse tree decides it has faced an error by judging its state, stack contents and input string.

Ideally, the parser should be able to detect the error encountered and recover from it to continue parsing the rest of the input.

Panic Mode :- A parser goes into panic mode when it cannot reconfigure itself to resume parsing. In panic mode, the parser rejects all input till it finds a synchronizing token such as a delimiter (\n or ;) It then empties its stack till its synchronizing token and then resumes parsing. The advantage of this method is that it is easy to implement

Storage Allocations

The allocation and deallocation of data items during runtime depends on the procedures called and number of active procedures at a given time. It is handled by run time packages which call subroutines to handle allocation and Deallocation of data objects.

Activation Trees :-

An activation tree is a syntactic structure which shows the number of instances of a procedure that are alive or active at a given time and the data flow between them.

Activation trees are normally made for recursive procedures.

Points in connection to activation tree :-

- Each node represents an active procedure
- The root represents the main program
- node 'a' is the parent of node 'b' if node 'a' activates node 'b'
- node 'a' is on the left of node b if node 'a''s lifetime of execution ends before node 'b'.

Activation records

Each node in an activation tree maintains an activation record. This record is maintained by the run-time stack. The activation records in the stack show the order in which the nodes in an activation tree have been activated. The last activated node has its record on top.

The information recorded in an activation record vary from language to language. In PASCAL, activation records contain:-

1) Temporaries → All locally computed data. It is not the actual registers, merely the local variables

2) Local Data → All locally declared variables and data

3) Access Link → Contains links to all the data that is being accessed by the record

4) Control Link → Contains a pointer to the caller routine

5) Saved Machine Status → Contains information about the state of the control stack before execution of subroutine. When subroutine ends, this value is reset into control stack

6) Space For Return Value :→ If the subroutine returns any values, it is saved here

7) Actual Parameters → This is where the data is actually saved in the registers. It is required for efficient data storage.

Static Storage Allocation

A compiler has to allocate and deallocate memory space for smooth operation of the source program. One method of allocating memory space is Static allocation.

In static allocation, pre-defined datatypes with fixed sizes can easily be stored in contiguous memory locations. Data structures such as arrays and linked lists can also be stored in memory block which are sufficiently large.

Static memory allocation needs no runtime support and is hence easy to implement.

Drawbacks – It cannot account for recursive calls where many instances may be active at the same time.

It does not work for data structures which change size dynamically.

Dynamic Memory Allocation :-

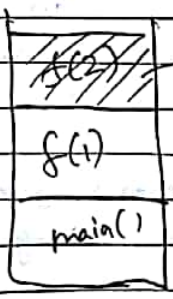
Dynamic storage allocation is required when multiple instances of same variable, as in the case of recursion, is required. It may also be needed for data structures which change size dynamically at run-time. Dynamic memory allocation requires runtime support. Two common ways for dynamic allocation are :-

1) Stack
2) Heap

Stack Allocation:- In stack allocation, memory locations are assigned every time a procedure is called. All data members used by the procedure is kept in this memory location. Once the procedure is over, the stack automatically deallocates the memory used by the procedure.

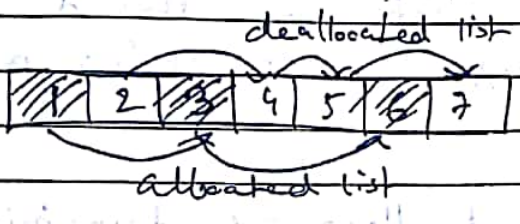
Stack allocation is especially useful for recursive procedures.

Eg



→ deallocated once f(2) is over

Heap Allocation- Another method for dynamic allocation is heap allocation. Heap allocations allocate memory spaces in contiguous locations and keep a link between them. When a memory space is deallocated, it is linked to the available free space.



Deallocation of memory may be explicit (as in PASCAL) or implicit (as in LISP)

The memory locations may be of fixed size or variable size.

b

For blocks of variable size, the memory gets fragmented over time with each block of free memory being added to the list of deallocated memory.

Now allocation is done by FIRST FIT method. That is for a block of code of size s, the first location of size f such that $s \leq f$ is chosen and f is allocated to it leaving a free block of size $(f-s)$. If two consecutive blocks are free, they are joined to form a larger block.

Storage allocation Structure of C :-

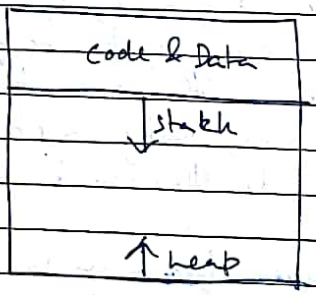
C uses a combination of static and dynamic allocation techniques.

Memory is divided into four parts :-
Code, Data, Heap, Stack.

Code and Data are stored statically. It comprises of all codes and global variables.

Stack and Heap are dynamically stored in same memory location but having opposite direction of growth

Activation records are stored in the stack while dynamic structures are stored in the heap.



Memory Allocation in C.