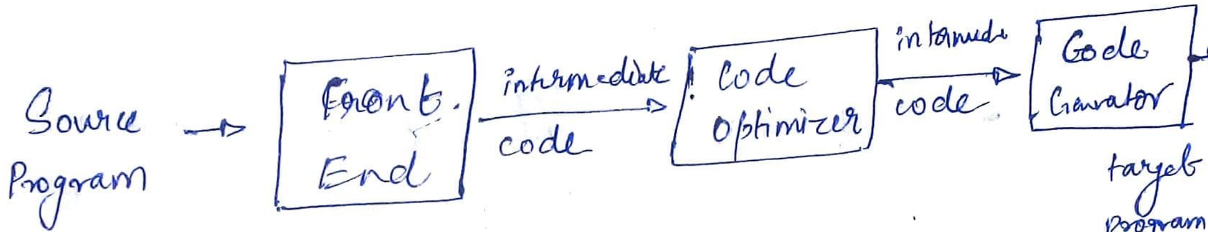
06.04.20

MODULE 6

CODE OPTIMIZATION & CODE GENERATION

(final stages of a compiler)

I Introduction



- The code generated by the compiler can be <u>made faster</u> or take <u>less space</u> or do both.
- The transformations that are done to perform these are called <u>optimization</u> or <u>optimizing transform</u>.
- Compiler that can apply <u>optimizing transforms</u> are called <u>optimizing compilers</u>
- It is an <u>optional</u> (5th) phase - but it must not change the purpose or meaning of the program.

Aim / Scope of Code Optimization

1. Aims to improve a program (rather than improving an algorithm used in a program – replacement of an algorithm is beyond the scope of code optimization)

- But highly efficient code generation (by maximum utilization of target machine's instruction set) is also beyond code optimization's reach.

- Compilers may take ~~upto~~ upto 40% of its time to optimize code (i.e. the rest of the process – lexican analysis to SDT & code generation together only took 60% of the compilation time)

- But due to optimization program occupied 25% less space & executed at least 3x faster than without optimization.

Need for Optimization

- Code produced by a compiler may not be perfect in terms of execution speed and memory occupied.
- Manual optimization will take large amount of time.
- Every program may not necessaly know the low-level details like the fundantal instructions, address mappings, ports etc.
- Advanced ~~arch~~ software archictures features like instruction pipline requires optimized code.
- Structure reusability & maintainability of the code are improved.

Critorial for Code Optimization

- It should preserve the meaning / purpose of the program.
- It implies that for a given input the corresponding output should not change.

- This is called the safe approach.

- Eventually it should improve the efficiency of the program.
  (At peculiar time the size of the code may be increased ... but the efficiency of the program in incrased)

- When compared with the effort put into optimize the output must be worthy enough.

⟹ Stages where optimization can be performed.

Source Code - - - User can cange algorithms, transform loops & refactor programs.

↓

[Frond End]

↓

Intomediate Code - - - Compiler can improve loops, temporary variables, procedure calls, address calulations.

↓

[Code Generator]

↓

Target code - - - Compiler can use registers select instruction do peep hole transformations.

⇒ Types of optimization

Machine Dependent Optimization (for particular machines)

Machine Independent optimization (for any machines)

⇒ Phases of Optimization

① Local Optimization

~ Transformations are applied over a small segment of the program - called basic block

~ In basic block the program is executed in the sequential order

~ Speedup factor is by 1.4x.

② Global Optimization

~ Transformation are applied over a large segment of the program like Lopp, procedures, fuction eatc.

~ Prerequisit for global optimization is local optimization

~ Speed up factor is by 2.7x.

Organization of code optimizer

* Basic Block

~ Basic Block is a sequence of 3 address statements which may be ~~entered~~ approached only at the begining.

~ But when approached all the statements are executed sequentially without halting or branching (or jumping).

~ To identify basic block, we have to find leader statements. Rules for leader statements are:

Input: A sequence of 3-address statements.

Output: A list of basic blocks with each three-address statement in exactly one block.

1. Determine the set of leaders, ~~the first statements~~
   (a) The first statement is the leader
   (b) Any statement that is the target of a goto is a leader
   (c) Any statement that immediately follows Goto is a leader.
2. For each leader, its basic block consis of the leader and all the statements upto the next

leader (but excluding it).

* Flow Graphs

~ It is a pictorial representation of control flow analysis in a program. ~~It shows the~~

~ It shows the relationship among basic blocks.

~ Node are basic blocks and edges are control flow. $G = (N, E, n_0)$ - is directed graph. where:

N = set of basic blocks (nodes)
E = set of control flows (edges)
n_0 = start block (starting node).

~ Eg:- $B1 \xrightarrow{E} B2$

B1 : Basic Block 1
B2 : Basic Block 2
E : control transfer from last statement of B1 to first (leader) statement of B2.

=> B1 is the predecessor of B2 or
B2 is the sucessor of B1.

~ Let's look at an exampe of quick sort.

```
void quicksort (m, n)
int m, n;
{
```

```
int i, j;
if (n <= m) return;
/* fragment begins here */
i = m - 1; j = n; v = a[n];
while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
}
x = a[i]; a[i] = a[n]; a[n] = x;
/* fragment ends here */
quicksort (m, j); quicksort (i+1, n);
}
```

⇒ 3 address code for "quicksort fragment"

1. $i = m - 1$
2. $j = n$
3. $v = a[n]$;
4. $t_1 = 4 * n$
5. $i = i + 1$
6. $t_2 = 4 * i$
7. $t_3 = a[t_2]$
8. if $t_3 < v$ goto (5)
9. $j = j - 1$
10. $t_4 = 4 * j$
11. $t_5 = a[t_4]$
12. if $t_5 > v$ goto (9)
13. if $i >= j$ goto (23)
14. $t_6 = 4 * i$
15. $x = a[t_6]$
16. $t_7 = 4 * i$
17. $t_8 = 4 * j$
18. $t_9 = a[t_8]$
19. $a[t_7] = t_9$
20. $t_{10} = 4 * j$

21. $a[t_{10}] = x$

22. goto (5)

23. $t_{11} = 4 * i$

24. $x = a[t_{11}]$

25. $t_{12} = 4 * i$

26 $t_{13} = 4 * n$

27. $t_{14} = a[t_{13}]$

28. $a[t_{12}] = t_{14}$

29. $t_{15} = 4 * n$

30. $a[t_{15}] = x$