ASSIGNMENT NO: 8.

Title: Implement local & global code optimizations such as common such expression elimination, copy propagation, dead code elimination, loop & basic block optimization (optional)

Problem Statement: WAP to implement different code optimization technique such as common sub expression elimination copy propagation, dead code elimination, loop & basic block optimization.

Objective: To learn & understand
- Different techniques of code optimization.
- Implementation of code optimization techniques.

S/w packages: C/C++ editors
Linux OS/ fedora/ Ubuntu.
keyboard, Mouse.

Theory: Code Optimization

Optimization is a program transformation technique which tries to improve the code by making it consume less resources (i.e. CPU Memory) & deliver high speed.
In optimization, high-level general programming constructs are replaced by very efficient low-level programming code. A code optimizing process must follow the three rules given below:

1. The output code must not, in any way, change the meaning of the program.

2. Optimization should increase the speed of the program & if possible, the program should demand less number of resources.

3. Optimization should itself be fast and should not delay the overall compiling process.

4. Machine-Independent Optimization

In this optimization, the compiler takes in the intermediate code & transform a part of the code that does not involve any CPU registers and/or reduce absolute memory locations.

Machine-dependent Optimization.

Machine-dependent optimization is done after the target code has been generated & when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory reference rather than relative references Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

Code Optimization techniques.

1. Common sub expression elimination.
2. Copy propagation.
3. Dual code elimination.
4. loop & basic block optimization.

Common sub expression elimination.

Common sub expression elimination is a compiler optimization technique of finding redundant expression evaluation and replacing them with a single representational computation. This saves the time overhead resulted by evaluating the expression for more than once.

eg:-

In the foll^{n} code:

```
a = b + c + g;
d = b + c + e;
```

it may be worth transforming the code to:

```
tmp = b + c;
a = tmp + g;
d = tmp + e;
```

if the cost of storing and retrieving tmp is less than the cost of calculating b+c an extra time.

Copy Propagation:

Copy propagation is the process of replacing the occurrences of targets of direct assignments with their values. A

direct assignment is an instruction of the form $x=y$, which simply assigns the value of y to x.

From the following code:

```
y=x;
z=3+y;
```

Copy propagation would yield:

```
z=3+x;
```

Algorithm:

1) Generate the flow graph from list of instructions.
   do {

2) Perform reaching copy analysis.

3) for each node in flow graph
   for each use
   if use is reached by the copy where it is the target.
   change the use to the source in the move statement tuple

   While (changes)

4) Generate the list of instruction from modified flow program.

Dead Code Elimination

Code that is unreachable or that does not affect the program (eg. dead stores) can be eliminated.

Eg:
In the eg below, the value assigned to i is never used, & the dead store can be eliminated. The first assignment to global is dead, & 3rd assignment to global is unreachable both can be eliminated.

```
int global;
void f()
{ int i;
  i=1;        /* dead store */
 global=1;    /* dead store */
  global=2;
  return
  global=3;   /* unreachable */
}
```

Below is the code fragment after dead code elimination

```
int global;
void f()
{ global=2;
  return;
}
```

Algo:

1) Generate the flow graph from list of instructions

do {

2) perform liveness on graph.

3) for each node in flow graph.

if the define a set contains only one memory

if the temporary being defined is not in the live out set

Remove the node from the flow graph.

While (changes)

4) Generate the list of inst^{n} from modified flow graph.

Conclusion:

Hence, implemented local & global code optimization such as dead code elimination, common sub expression elimination.