CODE OPTIMIZATION

Palak Rani
(RA2011003010661)
Palak Patel
(RA2011003010666)
Ananya Mishra
(RA2011003010679)

INDEX

ABSTRACT

1. Code optimization is a crucial step in the compiler design process that aims to improve the efficiency and performance of the generated code.
2. The optimization process involves analyzing the code to identify and eliminate inefficiencies, such as redundant calculations, unnecessary memory operations, and excessive control flow.
3. The optimization techniques range from simple local transformations to complex global optimizations that consider the entire program.
4. The ultimate goal of code optimization is to reduce the execution time and memory usage of the code while preserving the functionality and correctness. This abstract provides an overview of the principles and techniques of code optimization and highlights their importance in modern compiler design.

INTRODUCTION

The code optimizer maintains a key-value mapping that resembles the symbol table structure to keep track of variables and their values (possibly after expression evaluation). This structure is used to perform constant folding followed by dead code elimination.

CONSTANT FOLDING

1. Expressions containing constant operands are evaluated at compile time and the result is substituted for the expression in the code.

2. Constant folding can lead to faster and more efficient code execution, which can be especially important for programs that perform many calculations or run on resource-constrained devices

DEAD CODE ELIMINATION

Dead code elimination is a technique used in computer programming to remove portions of code that are not needed or do not have any effect on the output of the program.

This code can be the result of unused variables, unreachable code, or redundant code.

By removing the dead code, the compiler can produce more efficient code, which can result in faster program execution and smaller executable files.

METHODOLGY

1. The given code is a Python program that takes input text from the user, writes it to a file named "input.txt", reads the file, performs constant folding optimization on the code present in the file and prints the intermediate output of the optimization, and then performs dead code elimination and prints the final optimized code.
2. The program then performs constant folding optimization on the code present in the "list_of_lines" list. It iterates over each line of the list and checks if the operation is one of the arithmetic operators ("+", "-", "*", "/").
3. If the operation is not an arithmetic operator or an assignment, It appends the instruction to the "constantFoldedList" as is. Else if it is a constant it is added to dictValues()
4. The program then prints the optimized instructions present in "constantFoldedList" and stores them in the "constantFoldedExpression" list.
5. If it is a control flow instruction ("if", "goto", "label", "not"), it appends the instruction to the "deadCodeElemination" list.
6. The program prints the instructions present in the "deadCodeElemination" list, which represents the optimized code after dead code elimination and also modifies the output.txt with the optimized code.

RESULT

Taking input:

```
PS C:\Users\hp\OneDrive\Desktop\optimization> python -m flask run
Enter your text, then press Ctrl-D (Unix/Linux) or Ctrl-Z (Windows) to end input.
= 3 NULL a
+ a 5 b
+ a b c
* c e d
= 8 NULL a
* a 2 f
if x NULL L0
+ a e a
^Z
```

Output:

```
Quadruple form after Constant Folding
-------------------------------------
= 3 NULL a
= 8 NULL b
= 11 NULL c
* 11 e d
= 8 NULL a
= 16 NULL f
+ 8 e a


Constant folded expression -
---------------------
a = 3
b = 8
c = 11
d = 11 * e
a = 8
f = 16
if x goto L0
a = 8 + e


After dead code elimination -
-----------------------------
d = 11 * e
if x goto L0
a = 8 + e
```

CONCLUSION

This program provides a simple implementation of constant folding and dead code elimination for a list of quadruple-form instructions. While it is not a complete compiler, it provides a useful tool for optimizing code by reducing redundant computations and eliminating unnecessary instructions.

REFERENCES

- https://www.geeksforgeeks.org/code-optimization-in-compiler-design/amp/
- https://www.tutorialspoint.com/compiler_design/compiler_design_code_optimization.htm
- https://www.codingninjas.com/codestudio/library/code-optimization-in-compiler-design
- Advanced Compiler Design and Implementation" by Steven Muchnick
- "Optimizing Compilers for Modern Architectures: A Dependence-based Approach" by Randy Allen and Ken Kennedy
- "Engineering a Compiler" by Keith D. Cooper and Linda Torczon
- "Compiler Construction: Principles and Practice" by Kenneth C. Louden
- "Computer Systems: A Programmer's Perspective" by Randal E. Bryant and David R. O'Hallaron

THANK YOU