CODE OPTIMIZATION
A COURSE PROJECT REPORT

By

Adrija Mukherjee
(RA2011033010020)
Shrey Vardhan Dhiman
(RA2011033010055)
Aoushnik Aich
(RA2011033010045)

Under the guidance of

Dr. Sheryl Oliver

In partial fulfillment for the Course

of

18CSC304J – COMPILER DESIGN

in Computing Technologies

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District

May 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this mini project report Code Optimization is the Bonafide work of Adrija Mukherjee (RA2011033010020), Shrey Vardhan Dhiman (RA2011033010055)and Aoushnik Aich (RA2011033010045) who carried out the project work under my supervision.

SIGNATURE

Dr. Sheryl Oliver
Assistant Professor
Department of Computing Technologies
SRM Institute of Science and Technology

ABSTRACT

Code optimization is a crucial step in the compiler design process that aims to improve the efficiency and performance of the generated code. The optimization process involves analyzing the code to identify and eliminate inefficiencies, such as redundant calculations, unnecessary memory operations, and excessive control flow. The optimization techniques range from simple local transformations to complex global optimizations that consider the entire program. 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.

CONTENTS

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.

It takes input.txt file as input which contains the three address code in quadruple form, performs Constant Folding and Dead Code optimization and provides the optimized code as output in output.txt which is downloaded.

1.1 Constant Folding

Constant folding is a technique used in computer programming and compiler optimization to improve the performance of code by evaluating constant expressions at compile time instead of at runtime. In constant folding, expressions containing constant operands are evaluated at compile time and the result is substituted for the expression in the code. This reduces the computational overhead at runtime and can also simplify the code. For example, the expression "2+3" can be folded into the constant "5" at compile time, so that the code only needs to perform the assignment "x = 5" instead of the calculation "x = 2 + 3" at runtime. Constant folding can be applied to arithmetic, comparison, and logical operations, as well as other types of expressions. Overall, 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.

1.2 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. Dead code refers to the code that is never executed during the runtime of the program or is executed but has no effect on the output of the program. This code can be the result of unused variables, unreachable code, or redundant code. Dead code elimination can be an essential optimization technique in software development because it can reduce the size of the program, improve its performance, and make it easier to maintain. By removing the dead code, the compiler can produce more efficient code, which can result in faster program execution and smaller executable files. The process of dead code elimination involves analysing the code and identifying any portions of code that are dead. Once the dead code has been identified, it can be safely removed from the program without affecting the program's behaviour. This process can be performed automatically by the compiler or by using specialized software tools designed for this purpose.

METHODOLOGY AND TECHNIQUES

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 first prompts the user to enter the text and then uses a while loop to read each line of the input until it encounters an EOFError. The input lines are then appended to a list named "text_input" and joined together using the "\n" separator to form a single string "text_input_str".

3. The program then opens a file named "input.txt" in write mode and writes the text input to the file. It then reopens the same file in read mode and reads the lines of the file into a list named "list_of_lines".

4. The program initializes a dictionary named "dictValues" to store the values of variables in the code, three lists named "constantFoldedList", "constantFoldedExpression", and "dead Code Elimination" to store intermediate results, and an empty string "output_string" to store the final optimized code.

5. 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 ("+", "-", "*", "/"). If it is, it checks if both the arguments are constants, if yes, it evaluates the expression and stores the result in the "dictValues" dictionary and appends the optimized instruction in "constantFoldedList". If one or both of the arguments are variables, it checks if the values of the variables are present in the "dictValues" dictionary. If both the values are present, it evaluates the expression, stores the result in the "dictValues" dictionary and appends the optimized instruction in

"constantFoldedList". If only one value is present, it replaces the value in the instruction with the value from the "dictValues" dictionary and appends the instruction to the "constantFoldedList". If none of the values are present, it appends the instruction to the "constantFoldedList" as is.

6. If the operation is an assignment ("="), it checks if the right-hand side value is a constant, if yes, it stores the constant value in the "dictValues" dictionary and appends the optimized instruction in "constantFoldedList". If it is a variable, it checks if the value of the variable is present in the "dictValues" dictionary. If yes, it replaces the value in the instruction with the value from the "dictValues" dictionary and appends the optimized instruction to the "constantFoldedList". If the value is not present, it appends the instruction to the "constantFoldedList" as is.

7. If the operation is not an arithmetic operator or an assignment, it appends the instruction to the "constantFoldedList" as is.

8. The program then prints the optimized instructions present in "constantFoldedList" and stores them in the "constantFoldedExpression" list.

9. The program then performs dead code elimination on the instructions present in "constantFoldedList". It iterates over each instruction of "constantFoldedList" and checks if it is an assignment instruction. If yes, it skips the instruction. If it is an arithmetic operation, it appends the instruction to the "dead dElamination" list. If it is a control flow instruction ("if", "goto", "label", "not"), it appends the instruction to the
"dead Code Elimination" list.

10. Finally, the program prints the instructions present in the "deadCode Elimination" list, which represents the optimized code after dead code elimination and also modifies the output.txt with the optimized code.

IMPLEMENTATION (CODE)

```
f = open("input.txt","r")
# fout = open("output.txt","w")


list_of_lines = f.readlines()
dictValues = dict()
constantFoldedList = []
constantFoldedExpression = []
deadCodeElemination = []

output_string = ""


print("Quadruple form after Constant Folding")
print("-----------------------------------------------------")
for i in list_of_lines:
   i = i.strip("\n")
   op,arg1,arg2,res = i.split()
   if(op in ["+","-","*","/"]):
       if(arg1.isdigit() and arg2.isdigit()):
           result = eval(arg1+op+arg2)
           dictValues[res] = result
           print("=",result,"NULL",res)
           constantFoldedList.append(["=",result,"NULL",res])
       elif(arg1.isdigit()):
           if(arg2 in dictValues):
               result = eval(arg1+op+dictValues[arg2])
               dictValues[res] = result
               print("=",result,"NULL",res)
               constantFoldedList.append(["=",result,"NULL",res])
           else:
               print(op,arg1,arg2,res)
               constantFoldedList.append([op,arg1,arg2,res])
       elif(arg2.isdigit()):
           if(arg1 in dictValues):
               result = eval(dictValues[arg1]+op+arg2)
               dictValues[res] = result
               print("=",result,"NULL",res)
               constantFoldedList.append(["=",result,"NULL",res])
           else:
               print(op,arg1,arg2,res)
               constantFoldedList.append([op,arg1,arg2,res])
       else:
```

```
            flag1=0
            flag2=0
            arg1Res = arg1
            if(arg1 in dictValues):
                arg1Res = str(dictValues[arg1])
                flag1 = 1
            arg2Res = arg2
            if(arg2 in dictValues):
                arg2Res = str(dictValues[arg2])
                flag2 = 1
            if(flag1==1 and flag2==1):
                result = eval(arg1Res+op+arg2Res)
                dictValues[res] = result
                print("=",result,"NULL",res)
                constantFoldedList.append(["=",result,"NULL",res])else:
                print(op,arg1Res,arg2Res,res)
                constantFoldedList.append([op,arg1Res,arg2Res,res])

    elif(op=="="):
        if(arg1.isdigit()):
            dictValues[res]=arg1
            print("=",arg1,"NULL",res)
            constantFoldedList.append(["=",arg1,"NULL",res])
        else:
            if(arg1 in dictValues):
                print("=",dictValues[arg1],"NULL",res)
                constantFoldedList.append(["=",dictValues[arg1],"NULL",res])else:
                print("=",arg1,"NULL",res)
                constantFoldedList.append(["=",arg1,"NULL",res])

    else:
        print(op,arg1,arg2,res)
        constantFoldedList.append([op,arg1,arg2,res])

print("\n")
print("Constant folded expression - ")
print("----------------------------")
for i in constantFoldedList:
    if(i[0]=="="):
        print(i[3],i[0],i[1])
        constantFoldedExpression.append([i[3], i[0], i[1]])
    elif(i[0] in ["+","-","*","/","==","<=","<",">",">="]):
        print(i[3],"=",i[1],i[0],i[2])
        constantFoldedExpression.append([i[3],"=",i[0], i[2]])
    elif(i[0] in ["if","goto","label","not"]):
```

```
        if(i[0]=="if"):
            print(i[0],i[1],"goto",i[3])
            constantFoldedExpression.append([i[0], i[1],"goto",i[3]])
        if(i[0]=="goto"):
            print(i[0],i[3])
            constantFoldedExpression.append([i[0], i[3]])
        if(i[0]=="label"):
            print(i[3],":")
            constantFoldedExpression.append([i[3], ":"])
        if(i[0]=="not"):
            print(i[3],"=",i[0],i[1])
            constantFoldedExpression.append([i[0], i[1]])

print("\n")
print("After dead code elimination - ")
print("----------------------------------------")
for i in constantFoldedList:
    if(i[0]=="="):
        pass
    elif(i[0] in ["+","-","*","/","==","<=","<",">",">="]):
        print(i[3],"=",i[1],i[0],i[2])
        deadCodeElemination.append([i[3], "=", i[1], i[0], i[2]])
    elif(i[0] in ["if","goto","label","not"]):
        if(i[0]=="if"):
            print(i[0],i[1],"goto",i[3])
            deadCodeElemination.append([i[0],i[1], "goto"])

        if(i[0]=="goto"):
            print(i[0],i[3])
            deadCodeElemination.append([i[0], "=", i[3]])

        if(i[0]=="label"):
            print(i[3],":")
            deadCodeElemination.append([i[3], ":"])

        if(i[0]=="not"):
            print(i[3],"=",i[0],i[1])
            deadCodeElemination.append([i[3], "=", i[0], i[1]])
output_string = ""
output_string += "Quadruple form after Constant Folding\n"
output_string += "----------------------------------------------------\n"
for i in constantFoldedList:
    output_string += str(i) + "\n"

output_string += "\n"
output_string += "Constant folded expression - \n"
output_string += "-------------------------------\n"
```

```
for i in constantFoldedExpression:
    output_string += str(i) + "\n"

output_string += "\n"
output_string += "After dead code elimination \n"
output_string += "-------------------------------\n"
for i in deadCodeElemination:
    output_string += str(i) + "\n"


with open("output.txt", "w") as f:
    f.write(output_string)
```

RESULT

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.

The project demonstrated the importance of code optimization in improving the overall performance of software applications.

The project showed how different optimization techniques can be combined to achieve even better results.

Overall, a successful code optimization compiler design mini project should result in improved performance of the compiled code and demonstrate the importance of code optimization in software development

REFERENCE

- 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