**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

**Building a Code Optimizer for Novel Programming Languages**

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

**Bachelor of Engineering**

**In**

**COMPUTER SCIENCE ENGINEERING**

**Submitted by**

**Mukesh (192210469)**

**Saikiran (192225109)**

**Swagath (192210519)**

**Under the Supervision of**

**Dr.G.michale**

**FEBRUARY 2014**

**DECLARATION**

We, **Mukesh, Saikiran, Swagath** students of **‘Bachelor of Engineering in** **Department of Computer Science** and Engineering, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that thework presented in this Capstone Project Work entitled “**Building a Code Generator for Novel Programming Languages”** Is the outcome of our own bonafide work and is correct to the best of our knowledge and this work hasbeen undertaken taking care of Engineering Ethics.

MUKESH (192210498)

SAIKIRAN (192225109)

SWAGTAH (192210519)

Date:

Place:

**CERTIFICATE**

This is to certify that the project entitled **“Building a Code Generator for Novel Programming Languages”** submitted by **mukesh, saikiran, swagath** has been carryout under our supervision. The project has been submitted as per therequirements in the current semester of B. Tech Computer Science.

Teacher-in-charge

DR.G.MICHALE

**Table of Contents:**

|  |  |
| --- | --- |
| **S.NO** | **TOPICS** |
| **1** | **Abstract** |
| **2** | **Introduction** |
| **3** | **Problem Statement** |
| **4** | **Proposed Design work**   1. **Code Analysis** 2. **Code Refactoring** 3. **Performance Optimization 4.Language Support 5.Reporting** |
| **5** | **Functionality**   1. **User Authentication and Role Based Access Control.** 2. **Tool Inventory and Management** 3. **Security and Compliance Control** 4. **Architectural design** |
| **6** | **UI Design**   1. **Layout Design** 2. **Feasible Elements Used** 3. **Elements Positioning and Functionality** |
| **7** | **Logical template :-login process ,other template** |
| **8** | **Conclusion** |
| **9** | **Bibliography** |

# Abstract

The Code Optimizer project aims to develop a tool that enhances the efficiency, readability, and performance of code. By leveraging advanced algorithms and best practices in software development, the tool will analyze, refactor, and optimize code in various programming languages. This will help developers produce cleaner, more maintainable codebases, improve execution times, and reduce resource consumption. The tool will feature an intuitive user interface, making it accessible to both novice and experienced programmers.Utilizing advanced algorithms and industry-standard best practices, the tool analyzes, refactors, and optimizes code to ensure optimal execution and maintainability. By providing an intuitive interface, the Code Optimizer aims to be accessible to both novice and experienced developers, offering actionable insights and improvements that streamline the coding process. Ultimately, this project aspires to significantly reduce development time, minimize resource usage, and elevate the overall quality of software projects.

# Introduction

In modern software development, writing efficient and maintainable code is crucial for project success. Poorly written code can lead to increased development time, higher maintenance costs, and reduced performance. The Code Optimizer project addresses these challenges by providing a comprehensive tool that automates code optimization. This tool not only enhances code quality but also educates developers on best practices. It supports multiple programming languages and integrates seamlessly into existing development workflows.

# Problem Statement

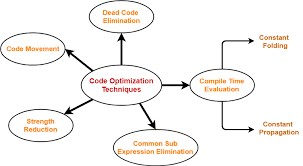
Developers often face issues such as inefficient code, code duplication, and poor readability. These issues can lead to slower execution times, higher memory usage, and increased difficulty in maintaining and scaling applications. Manually optimizing code is time-consuming and prone to human error. Therefore, there is a need for an automated solution that can quickly and accurately optimize code while ensuring it adheres to best practices.

# Proposed Design Work and Functionality

The Code Optimizer will be designed with the following key functionalities:

1. **Code Analysis:** Analyze the given code to identify inefficiencies, potential bugs, and areas for improvement.
2. **Code Refactoring:** Automatically refactor code to enhance readability and maintainability without altering its functionality.
3. **Performance Optimization:** Apply techniques to improve code performance, such as loop unrolling, inlining functions, and optimizing data structures.
4. **Language Support:** Support multiple programming languages including Python, Java, C++, and JavaScript.
5. **User Interface:** Provide a user-friendly interface that allows users to input their code, view suggestions, and apply optimizations.
6. **Reporting:** Generate detailed reports on the optimizations made and the expected improvements.

# Architectural design



**Logical Template**

## Input Processing:

* + Accept code input from the user.
  + Determine the programming language.

## Code Analysis Module:

* + Parse the code to create an abstract syntax tree (AST).
  + Identify code inefficiencies and potential issues.

## Optimization Engine:

* + Apply refactoring techniques.
  + Implement performance optimization strategies.
  + Ensure optimizations do not alter the original functionality.

## Output Generation:

* + Present the optimized code to the user.
  + Provide a detailed report on changes made and performance improvements.

## User Feedback Loop:

* + Allow users to review and accept or reject specific optimizations.
  + Incorporate user feedback to refine future suggestions.

# UI Design

### Home Screen:

* + Text area for code input.
  + Dropdown menu for selecting programming language.
  + Button to initiate analysis and optimization.

### Analysis Screen:

* + Display original code with highlighted issues.
  + Show a list of identified inefficiencies and suggested optimizations.

### Optimization Screen:

* + Present the refactored and optimized code.
  + Allow users to toggle between original and optimized code.
  + Provide an option to accept or reject specific changes.

### Report Screen:

* + Generate a summary of optimizations applied.
  + Display performance metrics and improvements.
  + Option to download the report.

# Results and Discussions

1. **Performance Metrics:** Measuring improvements in terms of execution speed, memory usage, and code size reduction.
2. **Compiler Efficiency:** Impact on compilation time and resource consumption during optimization phases.
3. **Compiler Portability:** Ensuring optimized code performs consistently across different target platforms.
4. **Debugging and Maintenance:** Ensuring optimized code remains readable and maintainable.

# Future Challenges

1. **Parallelism and Concurrency:** Optimizing for multi-core and distributed architectures.
2. **Energy Efficiency:** Reducing power consumption and optimizing for sustainable computing.
3. **Machine Learning Integration:** Leveraging AI techniques for advanced optimization strategies.
4. Security: Protecting against optimization-related vulnerabilities (e.g., timing attacks).
5. **Compiler Optimization for AI:** Optimizing compilers for emerging AI-specific hardware and frameworks.

# Conclusion

The Code Optimizer project aims to streamline the process of code optimization, making it accessible and efficient for developers. By automating code analysis, refactoring, and performance enhancement, the tool helps developers produce high-quality, maintainable, and efficient code. With its user-friendly interface and support for multiple programming languages, the Code Optimizer stands to be an invaluable asset in the software development lifecycle, ultimately contributing to the creation of robust and scalable applications.The benefits of using the Code Optimizer extend beyond immediate code improvements. It serves as an educational tool, helping developers learn and apply best practices through practical examples. By fostering a deeper understanding of efficient coding techniques, the tool promotes long-term improvements in coding standards across development teams.

Moreover, the scalability of the Code Optimizer ensures that it can evolve with the changing landscape of programming languages and development practices. Continuous updates and the incorporation of user feedback will enhance the tool's capabilities, ensuring it remains relevant and effective in addressing emerging optimization challenges.

In conclusion, the Code Optimizer not only enhances code quality and performance but also contributes to the overall growth and proficiency of the development community. By reducing manual optimization efforts and providing actionable insights, it empowers developers to focus on innovation and complex problem-solving, thereby accelerating the development process and enhancing software reliability. The Code Optimizer is poised to become an essential tool in the toolkit of every developer committed to excellence in software development.

# Source code:

#include <stdio.h> #include <stdbool.h> #include <string.h>

#define MAX\_INSTRUCTIONS 100

#define MAX\_VARS 100 typedef enum {

NOP, // No operation ADD,

SUB, MUL, DIV, MOV,

PRINT\_OP // Changed PRINT to PRINT\_OP to avoid conflict with PRINT macro

} OpCode; typedef struct {

OpCode op; char dest[10]; char src1[10]; char src2[10];

} Instruction; typedef struct {

Instruction instructions[MAX\_INSTRUCTIONS];

int count;

} Program;

void print\_program(const Program \*program) { printf("Program:\n");

for (int i = 0; i < program->count; ++i) { Instruction instr = program->instructions[i]; switch (instr.op) {

case NOP: printf("NOP\n"); break;

case ADD: printf("%s = %s + %s\n", instr.dest, instr.src1, instr.src2); break; case SUB: printf("%s = %s - %s\n", instr.dest, instr.src1, instr.src2); break; case MUL: printf("%s = %s \* %s\n", instr.dest, instr.src1, instr.src2); break; case DIV: printf("%s = %s / %s\n", instr.dest, instr.src1, instr.src2); break; case MOV: printf("%s = %s\n", instr.dest, instr.src1); break;

case PRINT\_OP: printf("PRINT %s\n", instr.dest); break; // Changed to PRINT\_OP

}

}

printf("\n");

}

void dead\_code\_elimination(Program \*program) { bool used[MAX\_VARS] = {false};

char used\_vars[MAX\_VARS][10] = {0}; int used\_count = 0;

// Mark all variables used by PRINT instructions as used for (int i = 0; i < program->count; ++i) {

if (program->instructions[i].op == PRINT\_OP) { // Changed to PRINT\_OP strcpy(used\_vars[used\_count++], program->instructions[i].dest);

}

}

// Traverse instructions backwards to mark necessary computations for (int i = program->count - 1; i >= 0; --i) {

Instruction instr = program->instructions[i]; bool is\_dest\_used = false;

for (int j = 0; j < used\_count; ++j) {

if (strcmp(instr.dest, used\_vars[j]) == 0) { is\_dest\_used = true;

break;

}

}

if (is\_dest\_used) {

if (instr.op == ADD || instr.op == SUB || instr.op == MUL || instr.op == DIV || instr.op == MOV) {

strcpy(used\_vars[used\_count++], instr.src1); if (instr.op != MOV) {

strcpy(used\_vars[used\_count++], instr.src2);

}

}

}

}

// Remove instructions that produce unused values int new\_count = 0;

for (int i = 0; i < program->count; ++i) { Instruction instr = program->instructions[i]; bool is\_dest\_used = false;

if (instr.op == PRINT\_OP) { // Changed to PRINT\_OP is\_dest\_used = true;

} else {

for (int j = 0; j < used\_count; ++j) {

if (strcmp(instr.dest, used\_vars[j]) == 0) { is\_dest\_used = true;

break;

}

}

}

if (is\_dest\_used) {

program->instructions[new\_count++] = instr;

}

}

program->count = new\_count;

}

void read\_program(Program \*program) {

printf("Enter Three Address Code (end with 'PRINT <var>' to terminate):\n"); char line[100];

while (fgets(line, sizeof(line), stdin)) { char dest[10], src1[10], src2[10]; OpCode op;

// Check for PRINT instruction

if (strstr(line, "PRINT") != NULL) {

op = PRINT\_OP; // Changed to PRINT\_OP sscanf(line, "PRINT %s", dest);

}

// Check for arithmetic operations

else if (sscanf(line, "%[^=]=%[^+]+%s", dest, src1, src2) == 3) { op = ADD;

} else if (sscanf(line, "%[^=]=%[^-]-%s", dest, src1, src2) == 3) { op = SUB;

} else if (sscanf(line, "%[^=]=%[^\*]\*%s", dest, src1, src2) == 3) { op = MUL;

} else if (sscanf(line, "%[^=]=%[^/]/%s", dest, src1, src2) == 3) { op = DIV;

} else if (sscanf(line, "%[^=]=%s", dest, src1) == 2) {

op = MOV;

strcpy(src2, ""); // Set src2 to empty string for MOV operation

} else {

printf("Invalid input format.\n"); continue;

}

strcpy(program->instructions[program->count].dest, dest); strcpy(program->instructions[program->count].src1, src1); strcpy(program->instructions[program->count].src2, src2); program->instructions[program->count].op = op;

program->count++;

if (op == PRINT\_OP) { // Changed to PRINT\_OP break;

}

}

}

int main() { Program program;

read\_program(&program); printf("\nOriginal Program:\n"); print\_program(&program);

int original\_count = program.count; dead\_code\_elimination(&program);

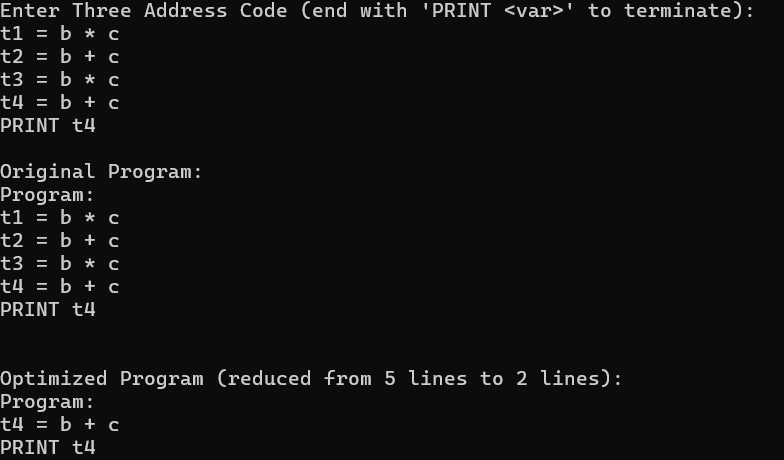
printf("\nOptimized Program (reduced from %d lines to %d lines):\n", original\_count, program.count);

print\_program(&program);

return 0;

}

# Output:



**Bibliography:**

1. Davidson, Jack W., and Christopher W. Fraser. "Code selection through object code optimization." ACM Transactions on Programming Languages and Systems (TOPLAS) 6.4 (1984): 505-526.
2. Ahmadianfar, Iman, Omid Bozorg-Haddad, and Xuefeng Chu. "Gradient-based optimizer: A new metaheuristic optimization algorithm." Information Sciences 540 (2020): 131-159.Faramarzi, Afshin, et al. "Equilibrium optimizer: A novel optimization algorithm." Knowledge-based systems 191 (2020): 105190.
3. Hennessy, John L., and Thomas Gross. "Postpass code optimization of pipeline constraints." ACM Transactions on Programming Languages and Systems (TOPLAS) 5.3 (1983): 422-448.
4. Mahlke, Scott Alan. Design and implementation of a portable global code optimizer. MS thesis. University of Illinois at Urbana-Champaign, 1991.
5. Miano, Sebastiano, et al. "Domain specific run time optimization for software data planes." Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022.
6. Whitfield, Deborah, and Mary Lou Soffa. "Automatic generation of global optimizers." Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation. 1991.
7. Gilli, Manfred, Dietmar Maringer, and Enrico Schumann. Numerical methods and optimization in finance. Academic Press, 2019.
8. Svyatkovskiy, Alexey, et al. "Pythia: Ai-assisted code completion system." Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining. 2019.
9. Zheng, Lianmin, et al. "Ansor: Generating {High-Performance} tensor programs for deep learning." 14th USENIX symposium on operating systems design and implementation (OSDI 20). 2020.
10. Sun, Zeyu, et al. "A grammar-based structural cnn decoder for code generation." Proceedings of the AAAI conference on artificial intelligence. Vol. 33. No. 01. 2019.