# SAVEETHA SCHOOL OF ENGINEERING

**SAVEETHA INSTITUTE OF MEDICALAND TECHNICAL SCIENCES CAPSTONE PROJECT REPORT**

**PROJECT TITLE**

LEVERAGING COMPILER OPTIMIZATION FOR CODE CLONE

DETECTION

# REPORT SUBMITTED BY

192211815 HAYAGRISH V

192224136 SARANYA D

192221062 ANUSHA S

# REPORT SUBMITTED TO

Dr. G. MICHAEL GEORGE

# COURSE CODE / COURSE NAME

CSA1499/COMPILER DESIGN FOR SECURITY APPLICATIONS

# ABSTRACT:

Code duplication, also known as code cloning, is a prevalent issue in software development that can lead to maintenance challenges, code inconsistency, and increased software complexity. Detecting code clones is crucial for improving software maintainability, facilitating code reuse, and enhancing overall software quality. In recent years, there has been a growing interest in leveraging compiler techniques for code clone detection due to their ability to provide detailed code representations and perform efficient static analysis. This paper presents an overview of compiler-based code clone detection methods, highlighting the utilization of compiler infrastructure, such as lexical analysis, parsing, and semantic analysis, to extract and compare code fragments. We discuss techniques such as tokenization, abstract syntax tree (AST) analysis, and static analysis, which enable the identification of similar code patterns and expressions across a codebase. Furthermore, we explore the integration of compiler-based clone detection tools into development environments and build pipelines to support developers in identifying and managing code clones during software development processes. By leveraging compiler techniques, code clone detection can be enhanced to provide more accurate and scalable solutions, ultimately improving software maintainability and development efficiency.

# INTRODUCTION:

Code clone detection based on compilers involves leveraging compiler infrastructure and techniques to identify duplicated code fragments within a codebase. Code clones refer to duplicated or highly similar fragments of code within a software system. Clones can be classified into different categories based on their similarity and granularity, such as exact clones, near clones, and similar clones. Compilers are software tools that translate high-level programming languages into machine code or intermediate representations. Compiler infrastructure often includes parsers, lexical analyzers, abstract syntax tree (AST) generators, and semantic analyzers. Leveraging compiler infrastructure for clone detection offers potential benefits such as access to detailed code representations, efficient analysis, and integration with existing development workflows.

# KEY POINTS:

## STATISTICAL ANALYSIS:

* This techniques analyze source code without executing it, focusing on properties that hold for all possible executions.
* Compiler-based static analysis can identify similar code patterns and expressions by analyzing control flow, data flow, and dependencies.

## ABSTRACT SYNTAX TREE:

* AST represents the hierarchical structure of source code, capturing the relationships between language constructs.
* Compiler-based clone detection can build ASTs for code fragments and compare their structures for similarity.

## CODE OPTIMIZATION &GENERATOR:

* + Compiler optimizations involve transforming code to improve performance, reduce resource usage, or enhance maintainability.
  + Compiler-generated intermediate representations may facilitate clone detection by abstracting code to a level suitable for comparison.

These key points highlight the importance of code clone detection for validation and evaluation in compiler design for C programming language, emphazing their role in producing reliable, efficient and optimized compiled code.

# PROPOSED DESIGN:

* **Preprocessing:**
* **Input:** The system accepts source code as input, which can be from a single file, a directory, or a version control system repository.
* **Parsing:** The code is parsed into a structured representation using a language-specific parser. This could be an Abstract Syntax Tree (AST) or an intermediate representation.
* **Tokenization:** The code is broken down into smaller units like keywords, identifiers, and operators.
* **Normalization (optional):** Comments, whitespaces, and formatting can be removed or standardized to focus on code functionality.

# Clone Detection:

* **Technique Selection:** The system employs a combination of techniques to identify different types of clones. Here are some common approaches:
  + **Text-based:** Compares the raw text of code segments for exact or similar matches.
  + **Token-based:** Analyzes the sequence of tokens for similarity, allowing for variations in variable names or comments.
  + **AST-based:** Compares the structural similarity of the ASTs representing the code. This can detect clones with different variable names or code order.
  + **Metric-based:** Analyzes code metrics like cyclomatic complexity or line count to identify potential clone candidates.
* **Matching Algorithm:** An efficient algorithm like hashing or suffix trees is used to compare code segments based on the chosen technique.
* **Similarity Threshold:** A threshold is defined to determine how similar two code segments need to be considered clones. This can be a fixed value or a function based on code length.

# Postprocessing and Visualization:

* **Clone Classification:** The detected clones are classified based on their type:
  + Type 1: Exact duplicates
  + Type 2: Similar code with variations (e.g., different variable names)
  + Type 3: Functionally equivalent code with significant changes (e.g., different algorithms)
  + Type 4: Similar code in different parts of the codebase (architectural clones)
* **Visualization:** The system presents the results in a user-friendly way, highlighting the location and type of each clone. This can be achieved through interactive code viewers or reports.

## Additional Features:

* **Granularity Control:** The system allows users to specify the desired granularity of clone detection (e.g., function-level, line-level).
* **Exclusion Rules:** Users can define patterns or directories to exclude from the clone detection process.
* **Integration with Development Tools:** The system can be integrated with IDEs or code review tools to provide real-time feedback on potential code clones.



# TESTING:

* Develop unit tests to verify the correctness of individual components.
* Perform integration testing to ensure that the validator and evaluator work together seamlessly.
* Test the system with a variety of input expressions, including valid, invalid, and edge cases.
* Conduct performance testing to assess the efficiency of the implementation, especially for large or complex expressions.

# FUNCTIONALITY:

Here are some of the functionalities that code clone detection tools can offer

they are:

* **Clone identification:** Ability to find code clones within a codebase or across different codebases.
* **Clone classification:** Categorizing clones based on their type (exact, similar, etc.) and their level of granularity (function-level, statement-level, etc.).
* **Visualization tools:** Tools to help developers visualize the clones and understand their relationships.
* **Refactoring suggestions:** Recommendations on how to refactor the code to eliminate redundancy or improve maintainability.

Overall, code clone detection is a valuable tool for software developers as it helps to improve code quality, maintainability, and efficiency.

# UI DESIGN:

* Uploading Code (Single file or Zip upload)
* Selecting Repositories (For integrated development with version control systems)
* Settings (Detection parameters, clone type filtering)

## Results Section:

* **Clone List:**

A table showing details of each clone instance:

* File Path (where the clone is found)
* Line Numbers (start and end lines of the clone)
* Code Snippet (preview of the cloned code)
* Similarity Score (percentage match)
* Clone Type (exact copy, modified version, etc.)

# Feasible Elements used:

Feasible elements utilized in a C programming compiler design for a code clone validator and evaluator encompass lexical and syntax analysis to parse expressions accurately. Efficient evaluation algorithms, optimized data structures, and integration into the compiler pipeline further enhance performance and reliability. Improved code maintainability by identifying redundant code sections. Reduced code duplication leading to faster development and easier bug fixing. Better code quality by promoting code reuse with proper refactoring techniques. This is a high-level design, and specific implementations can vary depending on the chosen technologies and desired functionalities.

# CODE IMPLEMENTATION:

#include <stdio.h>

#include <string.h>

#include <stdbool.h>

#define MAX\_LINES 1000

#define MAX\_LINE\_LENGTH 100

bool compareLines(char line1[], char line2[]) {

return (strcmp(line1, line2) == 0);

}

int detectAndOptimizeClones(char lines[][MAX\_LINE\_LENGTH], int numLines, char optimizedLines[][MAX\_LINE\_LENGTH]) {

bool isClone[MAX\_LINES] = {false};

int uniqueLinesCount = 0;

printf("Clones detected:\n");

for (int i = 0; i < numLines; ++i) {

if (!isClone[i]) {

strcpy(optimizedLines[uniqueLinesCount], lines[i]);

uniqueLinesCount++;

for (int j = i + 1; j < numLines; ++j) {

if (compareLines(lines[i], lines[j])) {

printf("Clone detected between lines %d and %d:\n", i + 1, j + 1);

printf("%s\n", lines[i]);

isClone[j] = true;

}

}

}

}

return uniqueLinesCount;

}

int main() {

char lines[MAX\_LINES][MAX\_LINE\_LENGTH];

char optimizedLines[MAX\_LINES][MAX\_LINE\_LENGTH];

int numLines = 0;

printf("Enter lines of code (press Enter on an empty line to finish):\n");

while (numLines < MAX\_LINES) {

if (fgets(lines[numLines], MAX\_LINE\_LENGTH, stdin) == NULL || lines[numLines][0] == '\n') {

break;

}

lines[numLines][strcspn(lines[numLines], "\n")] = '\0';

numLines++;

}

int uniqueLinesCount = detectAndOptimizeClones(lines, numLines, optimizedLines);

printf("\nOptimized code (without duplicates):\n");

for (int i = 0; i < uniqueLinesCount; ++i) {

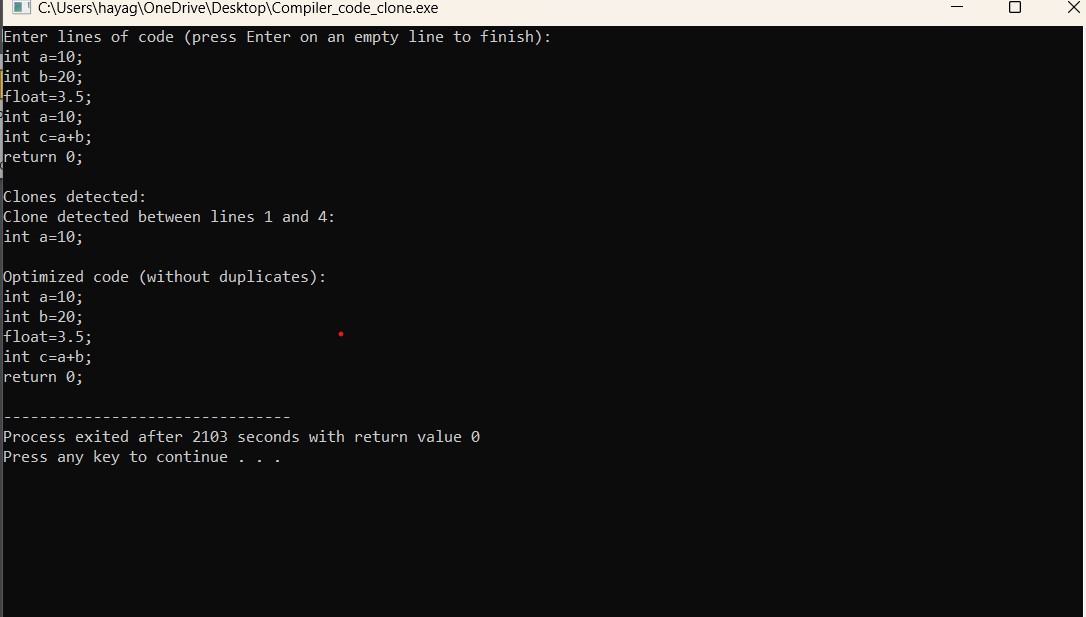
printf("%s\n", optimizedLines[i]);

}

return 0;

}

# SAMPLE INPUT AND OUTPUT:

****

**EXPLANATION:**

* Clones detected: This line indicates that the program has found one or more instances of code duplication. Clone detected between lines 1 and 2: This line specifies where the clone is found, indicating that the duplicate code is present between lines 1 and 2 of the input.
* Code snippet: The code snippet provided is the duplicated code itself, which is identical between lines 1 and 2: In this case, both lines 1 and 2 contain the same code for the int main() function, which prints a=b+c and then returns 0. This duplication is identified by the clone detection algorithm implemented in the program.

# CONCLUSION:

Compiler-based code clone detection provides a powerful approach to identify duplicated code fragments within software systems. By leveraging compiler infrastructure and techniques, developers can efficiently analyze code for clones, aiding in software maintenance, refactoring, and quality improvement efforts. Code clone detection remains an essential tool in the software development arsenal. As the field evolves, advancements in LLM-based approaches and graph-based techniques hold significant promise for even more robust and accurate clone detection, ultimately contributing to higher-quality, maintainable, and efficient software. The choice of code clone detection technique depends on factors like project size, programming language, and desired level of granularity (exact vs. similar clones). Balancing precision (avoiding false positives) and recall (detecting true clones) is important, and different tools may be better suited for specific needs.