HIGH LEVEL DESIGN DOCUMENT

Automated Tool for C Source Code Optimization

UE18CS390A – Capstone Project Phase – 1

Submitted by:

Khushei Meghana Meda	**PES1201800416**
Sriram Subramanian	**PES1201800655**
Adithya Bennur	**PES1201801891**
Shashank Vijay	**PES1201801828**

Under the guidance of

Prof. N S Kumar
Visiting Professor
PES University

January - May 2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

HIGH LEVEL DESIGN DOCUMENT

TABLE OF CONTENTS

1. Introduction

The purpose of this document is to provide an overview of the tool we are developing for the automated optimization of C source code. With this tool, we aim to convert C source code into an optimized version by applying some common techniques, as well as implementing Jon Bentley's rules to optimize code. The high level design involves parsing the source code to perform optimizations and generate the output C code from the intermediate structures.

2. Design Considerations

2.1. Design Goals

- The existing tools for profiling code perform static and dynamic analyses but do not refactor the code to a better version. The latter we hope to achieve along with the former.
- Compilers perform some optimizations, especially using some intermediate representations of the source code, but do not display the high level optimized code to the user.
- Our tool will generate as the output, optimized C source code enabling a clear understanding of the optimizations applied and maybe a better insight into writing programs that increase performance.
- Our tool will also try to implement some optimizations that are not implemented by popular C compilers. Some of these optimizations are suggested by Jon Bentley.
- Our tool will also highlight performance metrics that will contrast the input source code and the generated output source code. The metrics we are considering are execution time and amount of memory used.

2.2. Architecture Choices

The program execution environment during the development and testing of the tool has been the Python-lex-yacc (PLY) module, python version 3.6+ and Unix/Linux based systems for supporting tools such as clang-tidy, indent and kcachegrind. However, the requirements can easily be installed on other popular OS such as Windows and OSX.

Other than a minimal computer system setup (cpu, disk and memory) along with peripherals (mouse, monitor and keyboard), there are no explicit hardware requirements/constraints on the server.

The client would require internet access as well as a modern browser to access the online interface of the tool.

Pros-

- We are implementing the tool with a web based GUI with all the computational work at the server side, so the end user will not have to install or setup any software.
- All the architecture requirements on the server are quite easily achieved on modern systems.
- The use of open source software does not impose any financial constraints for both the client as well as the server.

Cons-

- The profiling metrics such as execution time will be dependent on the server side execution environment and there is no guarantee that the generated output code will perform equivalently on the client's system.
- Since a client server architecture is being followed , scalability problems may arise. However this could potentially be solved using cloud services with ease.

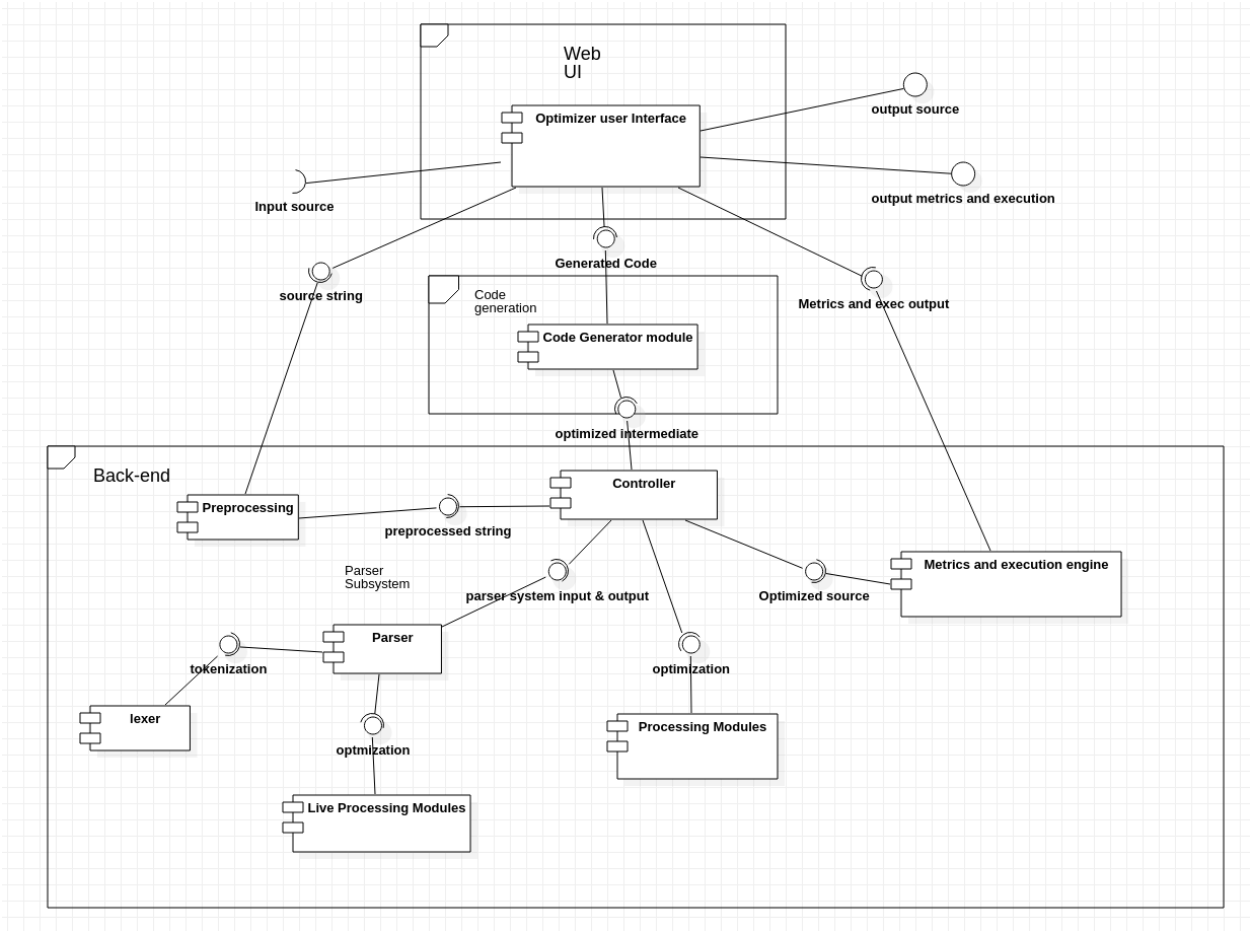
2.3. Constraints, Assumptions and Dependencies

- The GUI must be accessible irrespective of browser used by the client.
- The GUI must provide interfaces to select required optimizations , upload input files, download output files as well as a way to view the outputs and metrics of performance.
- The tool must output generated programs in second scale latencies and it must not increase the load on compilers.
- The user is required to input syntactically and semantically correct programs for the tool to output a logically equivalent and optimized version of the code. Otherwise the output program may not compile or the underlying logic may be undefined.
- Some of the optimizations being implemented are suggested programming practices that are not implemented by popular C compilers. Fully automating such optimizations is not possible with a rule-based approach.
- The user is required to have access to a stable internet connection to access the online graphical user interface of the tool.

- There could be issues in scalability after deployment. But could possibly be solved by migrating to a cloud service. The system is highly portable.

3. High Level System Design

Component Diagram

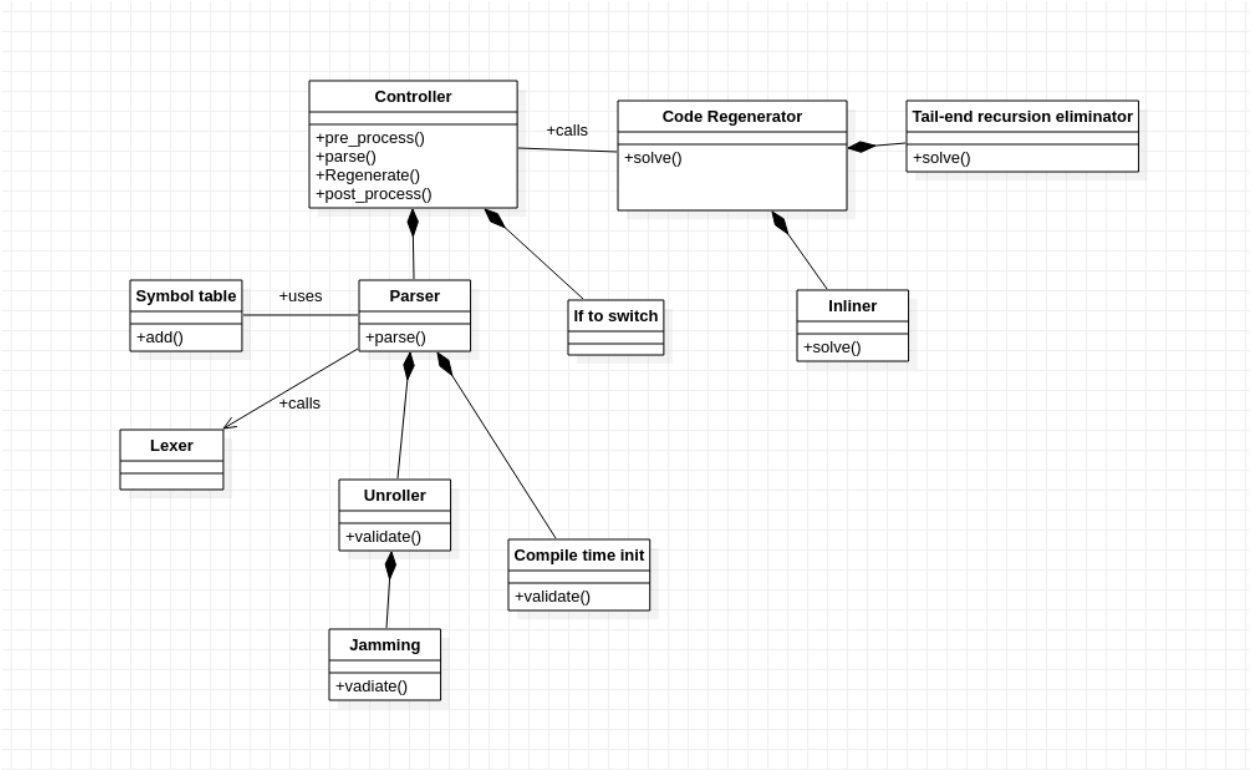


4. Design Description

interfaces listed:

1. parse
2. pre_process
3. tail_rec_eli_solve
4. fn_inline_solve
5. lookahead
6. compile_init_validate
7. for_unroll_validate
8. identify_chains
9. make_switch
10. make_compile_inits
11. post_process
12. **Code Generator**

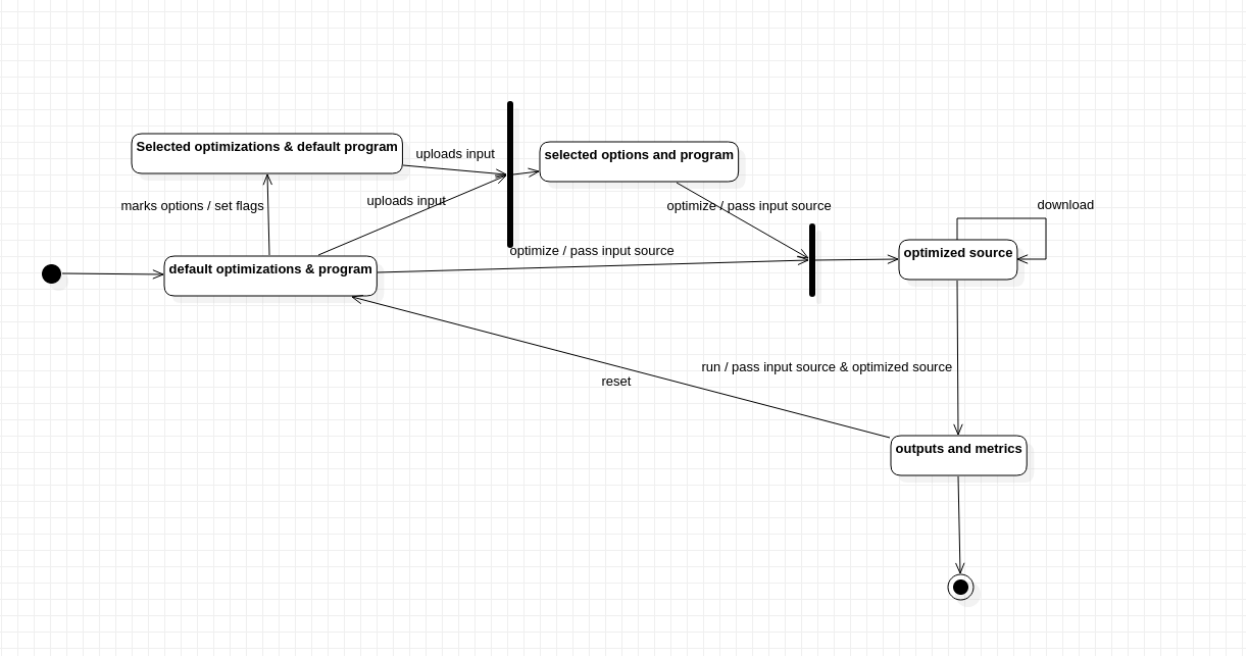
4.1. Class Diagram



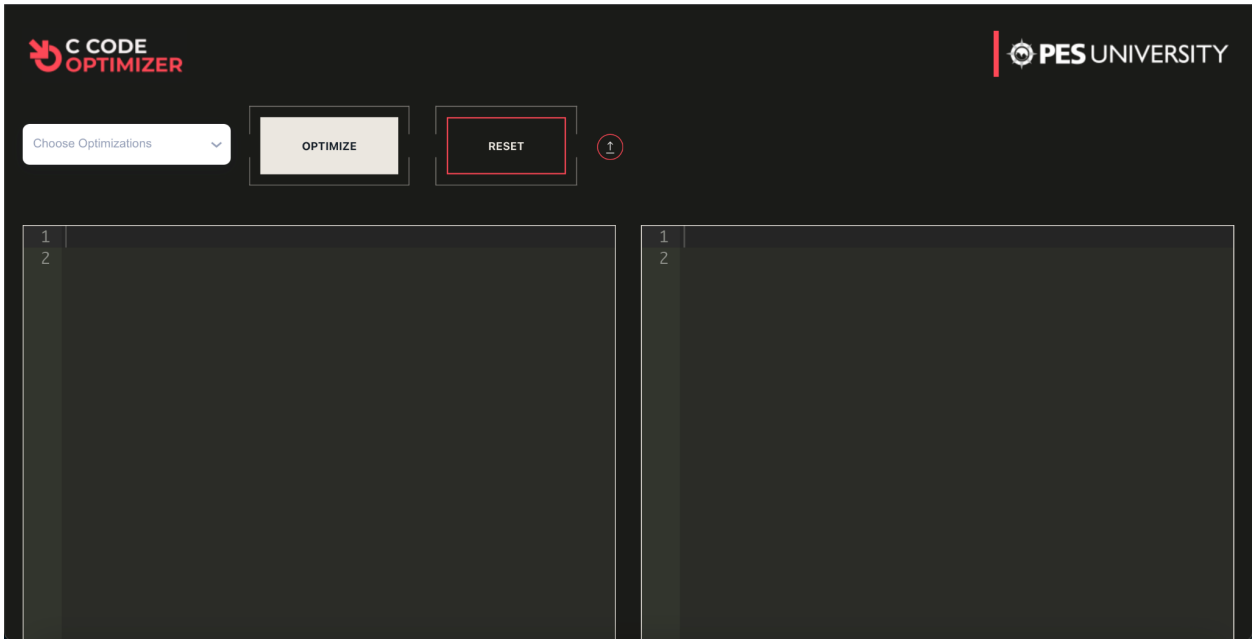
4.2. Reusability Considerations

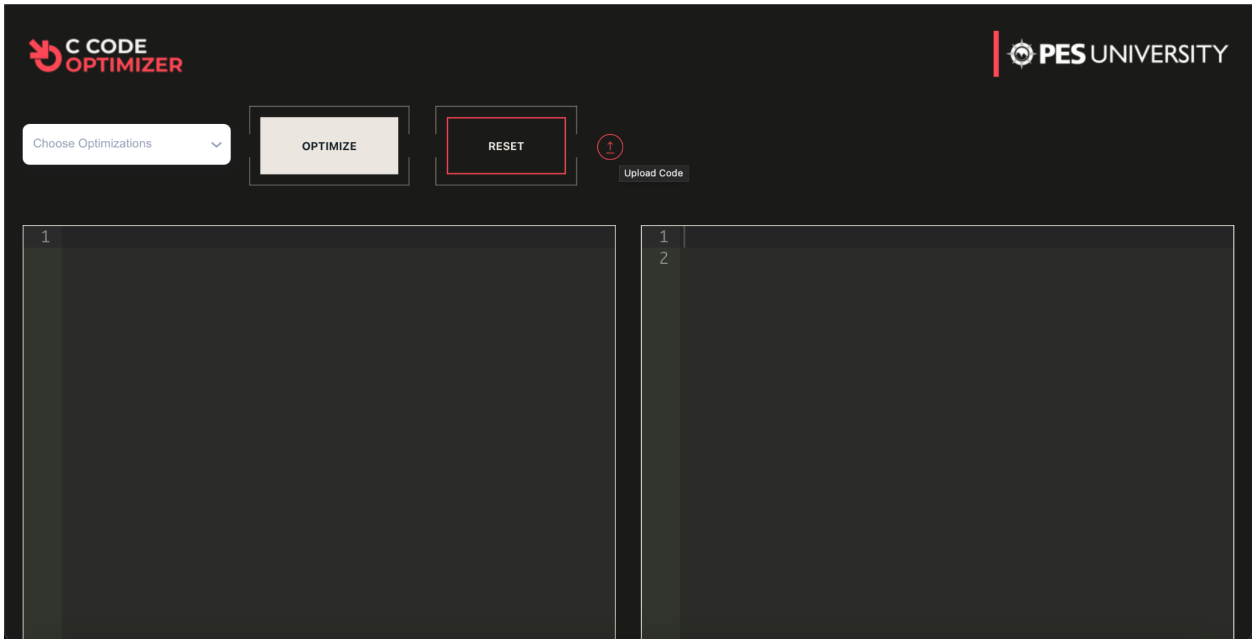
- Indent tool is used for indentation of output.
- The Clang-Tidy compiler is used to check correctness of programs written as well as preprocess the input program.
- Symbol Table Module is depended on by almost all optimization modules.
- Preprocessing and postprocessing modules can be used in general to clean code.

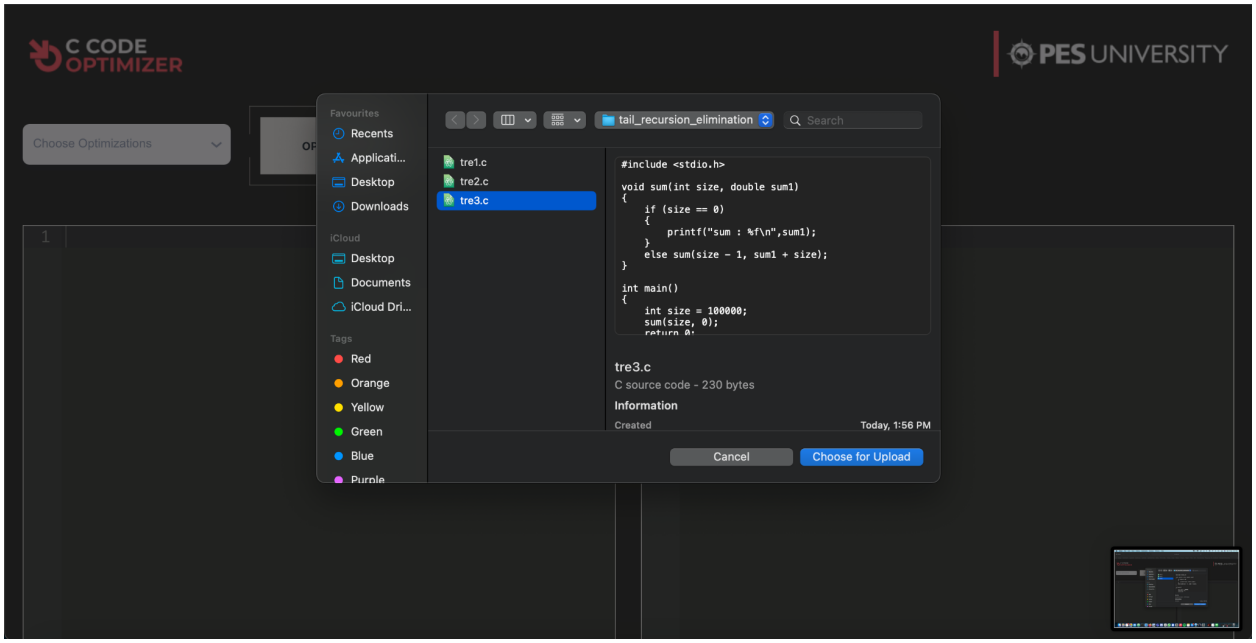
5. State Diagram

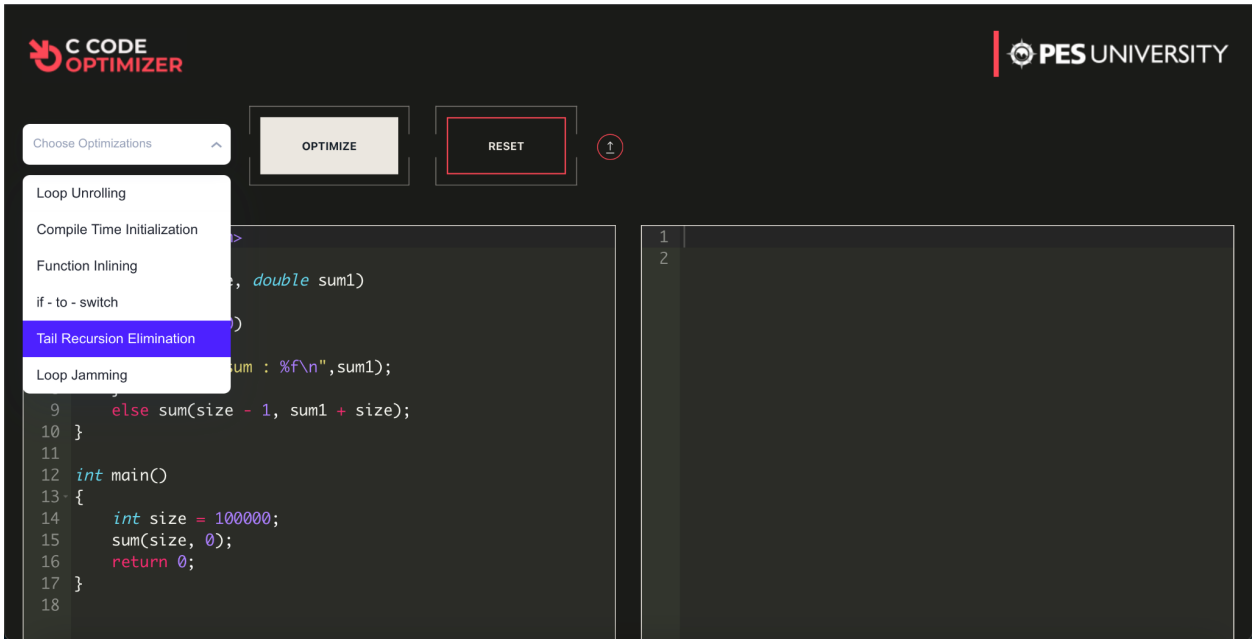


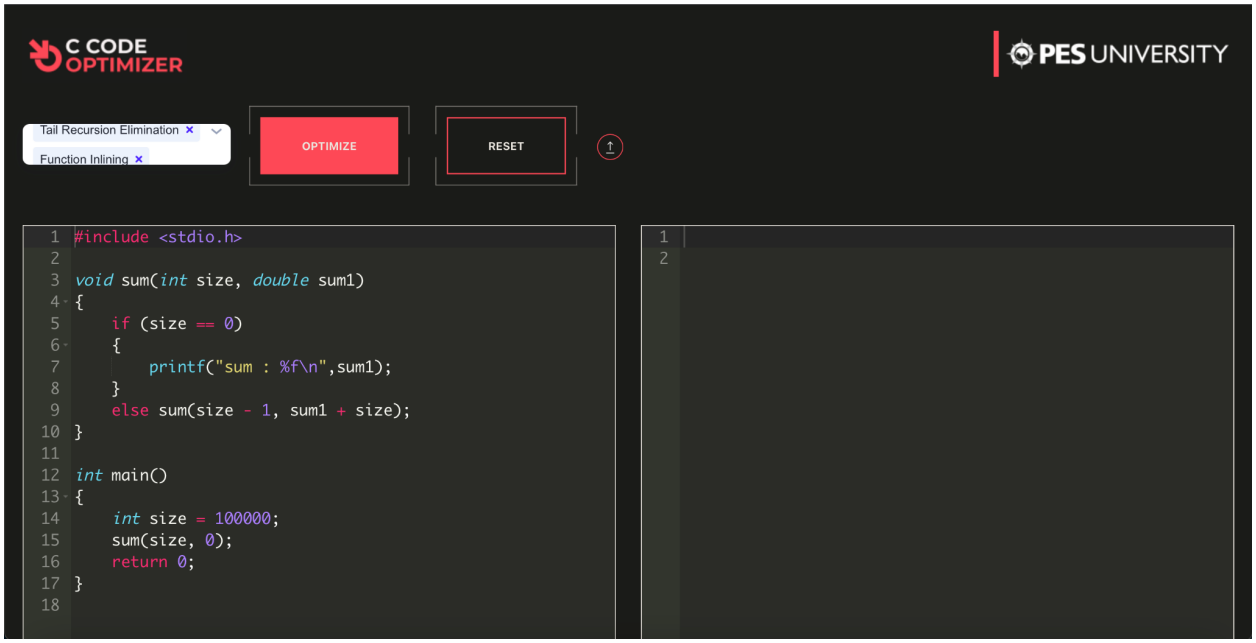
6. User Interface Diagrams

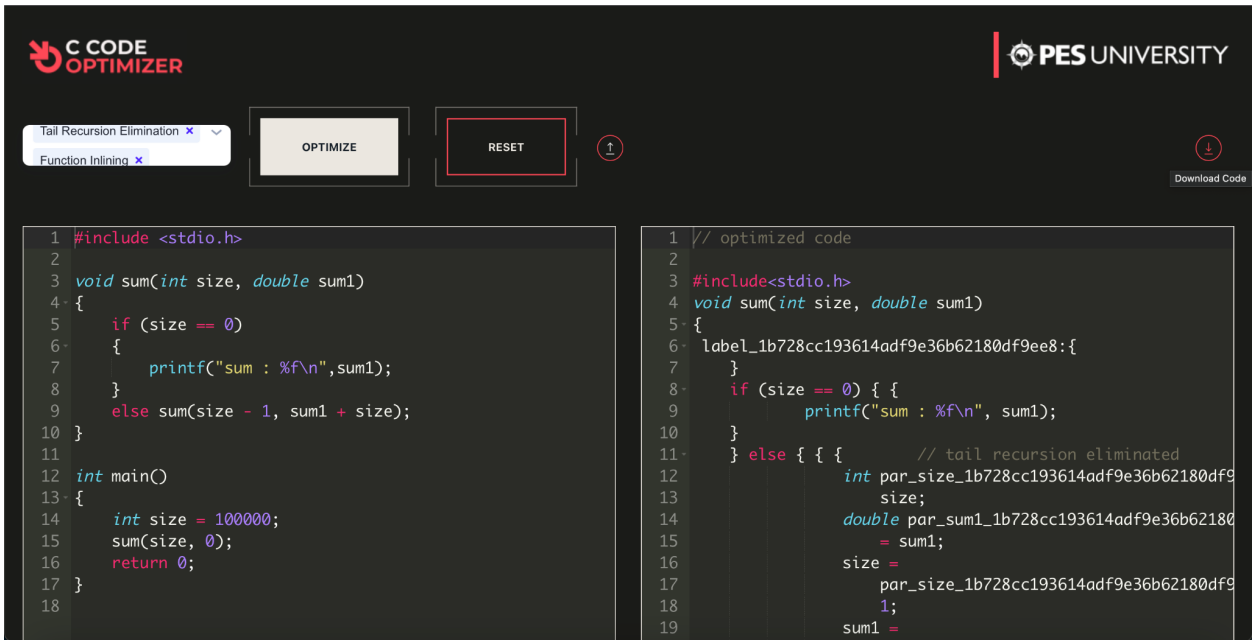





C CODE OPTIMIZER
PES UNIVERSITY
Choose Optimizations
Favourites
Recents
Applicati...
Desktop
Downloads
iCloud
Desktop
Documents
iCloud Dri...
Tags
Red
Orange
Yellow
Green
Blue
Purple
tail_recursion_elimination
Search
tre1.c
tre2.c
tre3.c
#include <stdio.h>
void sum(int size, double sum1)
{
if (size == 0)
{
printf("sum : %f\n",sum1);
}
else sum(size - 1, sum1 + size);
}
int main()
{
int size = 100000;
sum(size, 0);
tre3.c
C source code - 230 bytes
Information
Created
Today, 1:56 PM
Cancel
Choose for Upload


C CODE OPTIMIZER
PES UNIVERSITY
Choose Optimizations
OPTIMIZE
RESET
Loop Unrolling
Compile Time Initialization
Function Inlining
if - to - switch
Tail Recursion Elimination
Loop Jamming
double sum1)
sum : %f\n",sum1);
else sum(size - 1, sum1 + size);
}
int main()
{
int size = 100000;
sum(size, 0);
return 0;
}


C CODE OPTIMIZER
PES UNIVERSITY
Tail Recursion Elimination
Function Inlining
OPTIMIZE
RESET
#include <stdio.h>
void sum(int size, double sum1)
{
if (size == 0)
{
printf("sum : %f\n",sum1);
}
else sum(size - 1, sum1 + size);
}
int main()
{
int size = 100000;
sum(size, 0);
return 0;
}


C CODE OPTIMIZER
PES UNIVERSITY
Tail Recursion Elimination
Function Inlining
OPTIMIZE
RESET
Download Code
#include <stdio.h>
void sum(int size, double sum1)
{
if (size == 0)
{
printf("sum : %f\n",sum1);
}
else sum(size - 1, sum1 + size);
}
int main()
{
int size = 100000;
sum(size, 0);
return 0;
}
// optimized code
#include<stdio.h>
void sum(int size, double sum1)
{
label_1b728cc193614adf9e36b62180df9ee8:{
}
if (size == 0) { {
printf("sum : %f\n", sum1);
}
} else { { {
// tail recursion eliminated
int par_size_1b728cc193614adf9e36b62180df9
size;
double par_sum1_1b728cc193614adf9e36b62180
= sum1;
size =
par_size_1b728cc193614adf9e36b62180df9
1;
sum1 =

```
    if (size == 0)
    {
        printf("sum : %f\n",sum1);
    }
    else sum(size - 1, sum1 + size);
}

int main()
{
    int size = 100000;
    sum(size, 0);
    return 0;
}
```

```
{
 label_1b728cc193614adf9e36b62180df9ee8:{
    }
    if (size == 0) { {
            printf("sum : %f\n", sum1);
    }
    } else { { {        // tail recursion eliminated
                int par_size_1b728cc193614adf9e36b62180df9
                    size;
                double par_sum1_1b728cc193614adf9e36b62180
                    = sum1;
                size =
                    par_size_1b728cc193614adf9e36b62180df9
                    1;
                sum1 =
                    par_sum1_1b728cc193614adf9e36b62180df9
                    par_size_1b728cc193614adf9e36b62180df9
```

RUN

OUTPUTS &METRICS

ubuntu@ubuntu: ~CCodeOptimizer (Non-Optimized Output)

File Edit View Search

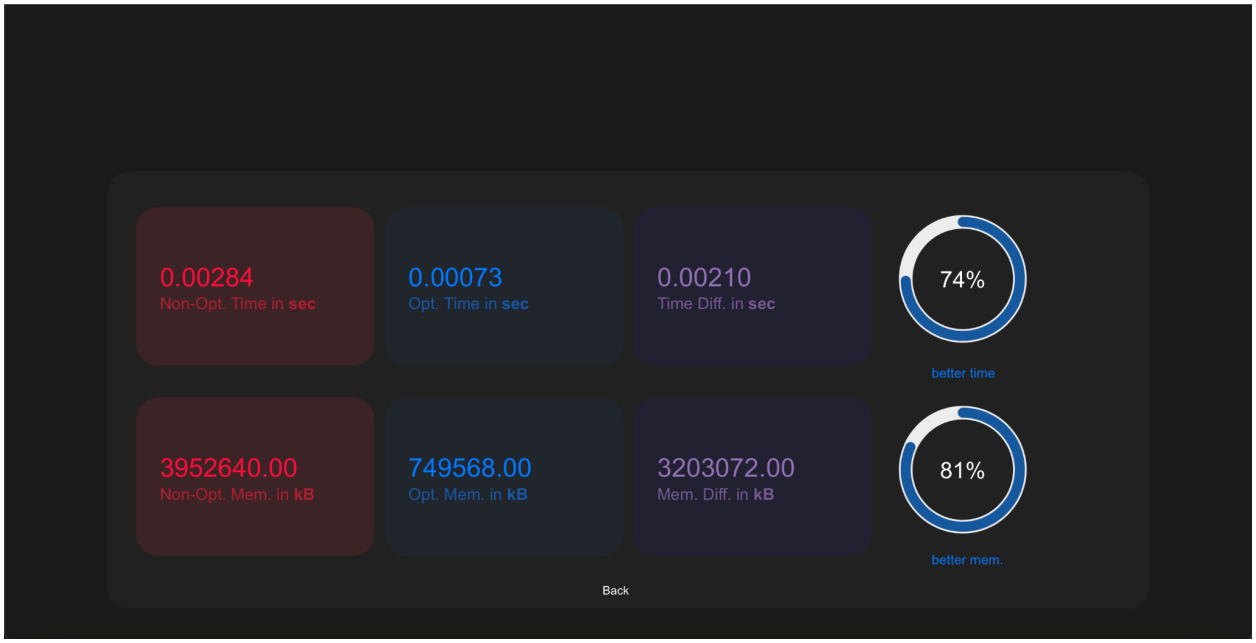
sum : 5000050000.000000

ubuntu@ubuntu: ~CCodeOptimizer (Optimized Output)

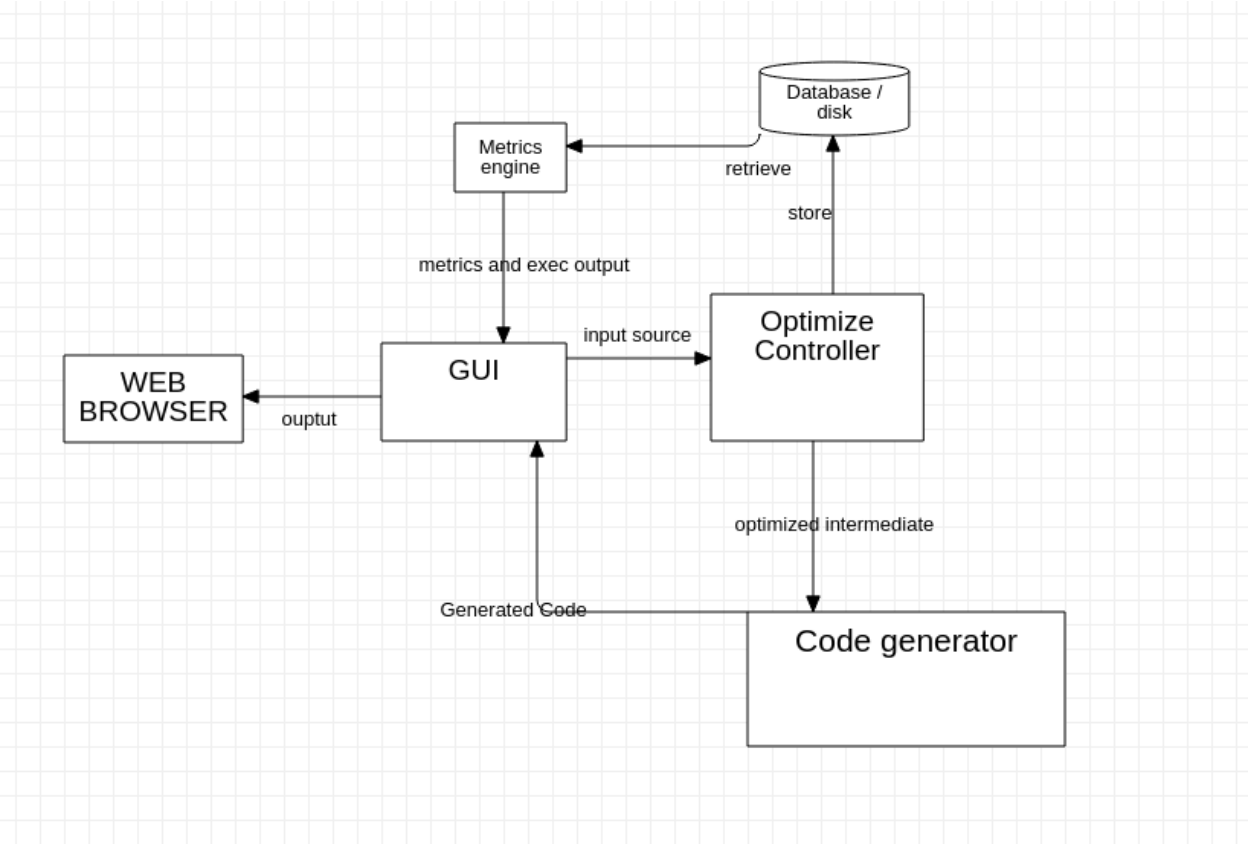
File Edit View Search

sum : 5000050000.000000

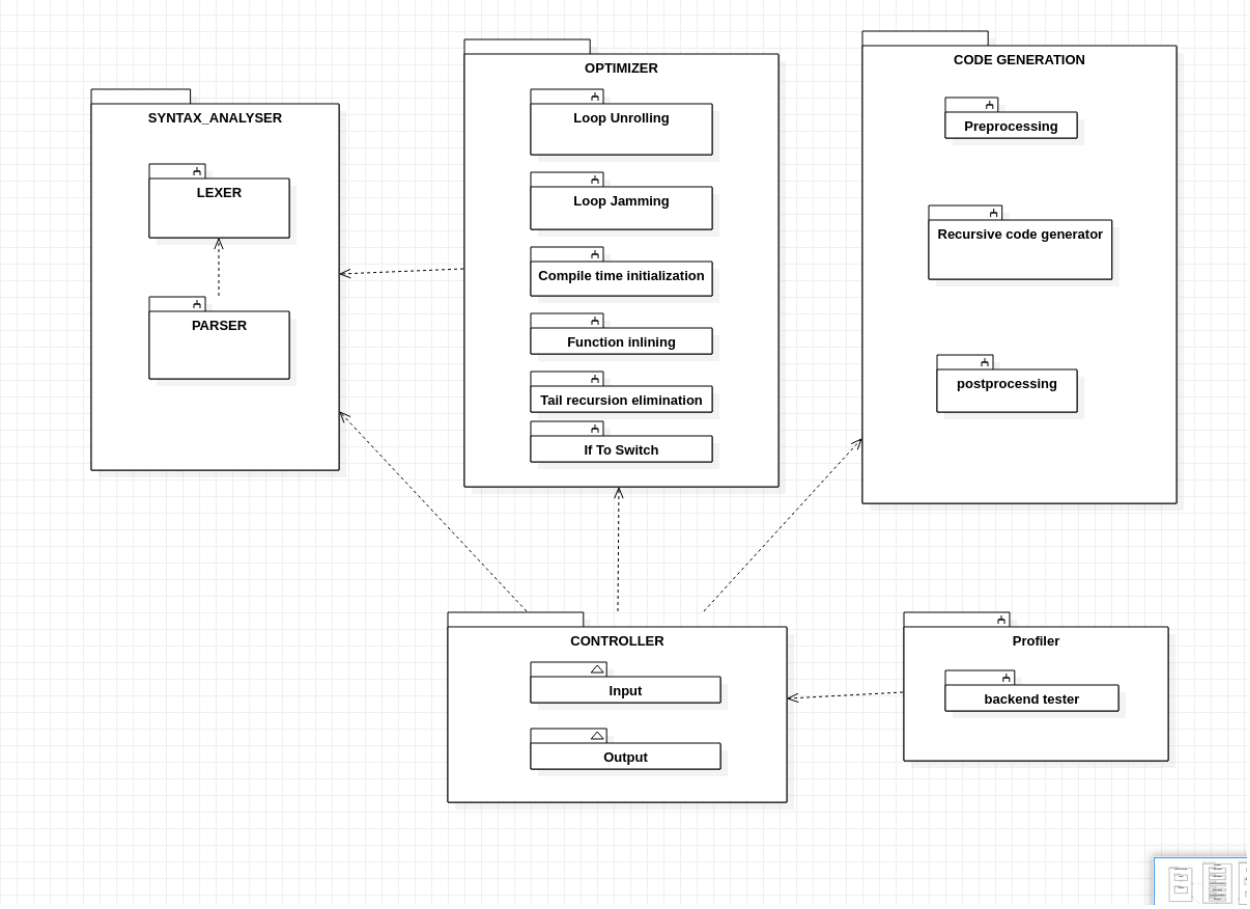
View Metrics


0.00284
Non-Opt. Time in sec
0.00073
Opt. Time in sec
0.00210
Time Diff. in sec
74%
better time
3952640.00
Non-Opt. Mem. in kB
749568.00
Opt. Mem. in kB
3203072.00
Mem. Diff. in kB
81%
better mem.
Back

7. External Interfaces



8. Packaging Diagram



9. Help

1. A page in the UI will be reserved for providing information about the interface itself and on how to use it.
2. A catalogue of all possible actions that the user can perform will be provided.
3. A catalogue of all the optimization options that a user can select will be provided.

10. Design Details

10.1 Novelty - The tool we are developing is completely new and there is no existing tool with the same set of features.

10.2 Innovativeness - Completely innovative in terms of implementation, given that there is no existing open source codebase for these optimizations.

10.3 Interoperability - The UI can be accessed by the user through any system as long as it has a modern web browser.

10.4 Performance - The tool is expected to deliver results in second scale latencies for the average input program size.

10.5 Reliability - The tool will not change the underlying logic of the input program provided that the user inputs syntactically and semantically correct programs.

10.6 Maintainability - The inherent code base is easy to maintain as it is modular and an object based model.

10.7 Portability - Clients can access UI from any system of choice in the presence of a modern browser. The server functions best in a unix/linux based environment due to component dependencies. However, substitutes for the components are available and the server will function in other systems.

10.8 Reusability - Code base consists of a few modules that can prove to be useful in other applications.

Appendix A: Definitions, Acronyms and Abbreviations

Abbreviations:

OS - Operating Systems

UI - User Interface

Definitions:

Bentley's rules - refers to the set of rules formulated by John Bentley which are known to improve performance of programs when applied.

Appendix B: References

https://www.geeksforgeeks.org/basic-code-optimizations-in-c/
https://www.keil.com/support/man/docs/armclang_intro/armclang_intro_fnb1472741490155.htm
https://medium.com/@riddhipandya_82734/code-optimization-techniques-266c2199e62b

https://alibabatech.medium.com/gcc-vs-clang-llvm-an-in-depth-comparison-of-c-c-compilers-899ede2be378
https://easyaspi314.github.io/gcc-vs-clang.html
https://en.wikipedia.org/wiki/High-level_design
https://tallyfy.com/uml-diagram/

Appendix C: Record of Change History

#	Date	Document Version No.	Change Description	Reason for Change
1.	07 - 04-21	2	Changed component diagram	addition of a new component.
2.	10-04 -21	3	Appended to interface listing	New interface

Appendix D: Traceability Matrix

Project Requirement Specification Reference Section No. and Name.	DESIGN / HLD Reference Section No. and Name.
3. Functional Requirements	3. High level system design - component diagram
5. Non-functional requirements	11. Design details