Paper1

by Sriram Subramanian

Submission date: 22-Apr-2021 01:19PM (UTC+0530)
Submission ID: 1566431075
File name: Sriram_Subramanian.pdf (1.79M)
Word count: 4064
Character count: 21796

ABSTRACT

Optimization of source code is a key aspect of performance of an application, and scalability and maintenance of code. The C programming language has been ranked the most popular programming language as of 2021 according to the TIOBE rankings. The goal of C is efficiency, and many software systems are still developed in C to achieve high performance. The tool we are developing aims to automatically detect and perform optimizations on C code, and generate logically equivalent C programs with some better or equivalent performance metric. The tool is unique in the sense that it generates optimized C code, and allows the developer to view and compare the changes from the original code. Such a tool is useful in large codebases where it is hard to manually detect and perform such optimizations.

CHAPTER 1

INTRODUCTION

Optimization has always been an important aspect to consider in software development. After having achieved logical correctness, one does and always must look to improve upon the efficiency of the developed code. Optimization can save the expenditure of cpu cycles, memory, power, and help us acquire the desired output in a shorter period of time. Even small improvements in these areas can have a huge impact in a large application, especially in terms of throughput and efficient use of resources. In today's world, optimization can prove to be the key in setting an application apart from another.

The C programming language has been ranked the most popular programming language as of 2021 according to the TIOBE rankings. The goal of C is efficiency, and many software systems are still developed in C to achieve high performance. The C programming language is also regarded by most in the field of computer science as a must-learn programming language.

Most modern and widely used compilers perform multiple optimization, referred to as compiler optimizations. These are especially performed when an optimization level is mentioned explicitly during compilation. Compilers perform both machine independant and machine dependent optimizations. However, these optimizations are conducted on lower level intermediate representations, and then translated to target machine code. The user does not see the optimized code in the high level language.

There are some optimizations that are not implemented in currently popular compilers.

Jon Louis Bentley is an American computer scientist. He is considered a pioneer in the field of computer science as well as an authority on optimization techniques. Bentley's rules for optimizing work is one of his formulations that have been widely appraised and moreover, known to provide improvements in performance when applied. Only a few of Bentley's rules are implemented in current day compilers.

Bentley's rules for optimizing work mostly focus on reducing the number of comparisons in code, decreasing the memory footprint and reducing the amount of work done at run time. Performing these optimizations however, may lead to larger code files and decreased readability.

CHAPTER 2

PROBLEM STATEMENT

Software tools currently available in the market perform static and dynamic analyses but do not refactor code to result in better performance. Compilers do not implement many optimization techniques that can prove to be very useful and detrimental. Moreover, compilers do not output optimized source code, disallowing further experimentation by the user and better insight into optimization techniques.

We propose to provide a solution by developing an automated tool for the optimization of C source code.

We propose to develop a tool that **generates optimized source code** as the output, given input source.

We propose to try and implement at least 10 of Jon Bentley's rules for optimizing work, few of which aren't implemented in current day compilers.

We propose to develop a graphical user interface which would be convenient for the user as well as informative and productive.

We propose to provide user flexibility in choosing the optimizations techniques to perform.

We propose to provide performance metrics and underline the contrast in performance between un-optimized and optimized versions through the interface.

CHAPTER 3

LITERATURE SURVEY

Here, we present the knowledge that gave us better insight into the field and helped guide us throught the processes of design and development.

3.1 A Survey Of Compiler Optimization [1]

Compiler Optimization can be broadly classified as:

1. Machine Dependent optimizations
2. Architecture Dependent optimizations
3. Architecture Independent optimizations

3.1.1. Machine Dependent Optimization:

- Are usually local optimizations, and are applied on small parts of code.
- These optimizations reduce the time and memory consumed by the program.
- Mckeeman proposes a post processing technique (window technique).
- Bagwell describes several coding tricks that can be implemented on almost any machine.

3.1.2. Architecture Dependent Optimization:

Three characteristics of machines:

a) Has n accumulators:-

use lesser number of registers for the update by:

i) looking ahead and generating RHS of expression first.

ii) using the deferred store technique, eliminates storing of partial results within loops.

b) Parallel independent instruction execution:-

i) Reordering of instructions can result in ability to execute in parallel.

ii) Utilization of independent execution units is higher.

iii) Inherent parallelism present in the original program would dictate the improvements attained by the reordering of instructions.

c) Arithmetic and logical operations executed on multiple data streams.

Takes advantage of inherent parallelism in procedural code, 3 steps:

i) Determine data dependencies.

ii) Examine flow within loops.

iii) Determine expression ordering.

3.1.3. Architecture Independent optimization:

These are global optimization techniques.

1. Common subexpression elimination.
2. Elimination of Dead variables.
3. Code Motion
4. Constant propagation

3.1.4 Analysis Of Source:

1. Frequency Analysis

   Original Fortran compiler:

   - Analysis of source program, division into basic blocks
   - Tabulation of basic block predecessors
   - The Monte Carlo simulation technique used to find relative frequency of execution for each basic block.
   - Optimizations carried out on blocks from lower to higher frequencies.

2. Matrix Analysis
   - To avoid lengthy Monte Carlo techniques, Prosser proposes an approach using boolean matrices.
   - Predecessor information is used to construct a connection matrix.
   - Basic blocks which are part of loops are determined by using repeated multiplication of the boolean matrices.

- Boolean-matrix multiplication can also be reduced to O(n^2) making it more practical.

3. Graph-Theoretic Analysis
    - Busam proposes and reviews a three pass optimizing compiler.
    - In the first pass all operations are encoded and recorded in a table.
    - The second pass scans code in reverse and removes common subexpressions as well as performs code motion.
    - Third pass deals with assignment to registers and generation of code.

3.2 General Possible Optimizations In C [2]

1. Loop unrolling
2. Avoid calculations in loop - Most compilers are good at identifying loop invariant computations.
3. Avoid pointer dereference in loop - It is a loop invariant action, so a temporary variable can be used to store this result and this can be done outside the loop.
4. Use Register variables as counters of inner loops - Accessing these variables can save time as accessing register variables is quicker than accessing ones in memory.
5. Loop jamming - combine adjacent loops which loop over the same range of the same variable
6. Loop inversion - Assuming the loop is insensitive to direction. Some machines have a special instruction for decrement and compare with 0.
7. Mathematical optimizations

    i) Avoid integer division where possible because it is slow

    ii) Use shift operators instead of * and / for powers of 2

    iii) Strength reduction - Use an alternative cheaper operator than a costly one where possible
8. Switch instead of if-else when possible, especially when case labels can be contiguous because that creates a jump table which is fast.
9. Prefer int to char or short - C performs all operations of char with an integer. So if an operation with char is performed several times, it amounts to more number of overhead conversions on the backend.
10. Short-circuit evaluation

11. Dead Code Elimination

3.3 Comparison Of Popular Compilers [3]

The most popular compilers for C are GCC (from GNU), Clang and MSVC.

GCC : Free software. It is widely used and is the official compiler for GNU and linux systems, also used in the creation of other operating systems.

Clang: Low-level Virtual Machine (LLVM) contains a series of modularized compiler components and tool chains. Optimizations can be done at compile time, run time as well as idle time. Clang is known for providing better optimization performance when compared to GCC.

MVSC: Closed, proprietary .

Comparison in terms of error handling (highlighting the pros of each):

GCC	Clang	MSVC
With the -O2 flag, Out of bounds error can be detected.	Cannot detect out of bounds even with -Warray-bounds or /Wall.	Cannot detect out of bounds even with -Warray-bounds or /Wall.
Doesn't detect the error.	Reports error for code that attempts to perform string concatenation using + operator.	Doesn't detect the error.
Unclear errors.	Unclear errors.	In case of if-else statements without braces for multi-lined body of if, and in the case of a returning function with a missing return statement, the errors are clear.

table 1.

Comparison in terms of compilation process:

GCC	Clang
Compilation process is as follows: read the source file -> preprocess the source file -> convert it into an IR -> optimize -> generate an assembly file -> the assembler generates an object file.	The process of generating assembly files is omitted in the process of generating object files. The object file is generated directly from the IR.
GCC IR is not so concise.	The data structure of LLVM IR is more concise => occupies less memory during compilation, supports faster traversal. Clang and LLVM are advantageous in terms of the compilation time. Clang reduces the single-thread compilation time by 5% to 10% compared with GCC. Therefore, Clang offers more advantages for the construction of large projects.

table 2.

Comparison in terms of execution performance:

GCC	Clang
Always advantageous in terms of performance optimization. On a code with no hotspots, 1% to 4% performance advantage over Clang and LLVM for most programs at the O2 and O3 levels.	Clang and LLVM are conservative in loop optimization and thus not advantageous in performance.
Growth not as rapid as Clang.	For AI-related programs, benchmarking showed a 3% better performance compared to GCC. It is fast improving.
GCC optimizes the vectors at the O3 level. Except for vectorized programs, GCC does not greatly improve the performance at the O3 level compared with that at the O2 level.	Clang and LLVM optimize the vectors at the O2 level, so benchmarking has shown it to perform better when hotspots involve vectorized regions.

table 3.

3.4 Similar Existing Tools

- Compiler Explorer is an open source software/website that can compile and show the assembly created by a wide variety of compilers, platforms and settings.
- Some static code analysis tools:
    1. PVS-Studio- detects some typing errors like copy-paste errors, it helps find bugs in the code and is capable of detecting some security flaws and loopholes in the code. However, it is no replacement for profiling tools. [4]
    2. Provides technical debt estimation and code visualization [5]
    3. Helix QAC- The tool gives good diagnostics. It is preferred in cases where there is a strict requirement to meet specific coding standards. It has features like prioritizing

issues, identifying critical problems on the basis of baselines or other parameters, and suggesting fixes for all of these. [6]

3.5 Optimization Of Code Through Hardware Resource Policies [7]

In any model, multiple hardware resource management strategies are applied to resources which are similar and associated with certain processors after which the instruction is executed in the said processor.

Then the method selects the substantially optimum hardware resource management strategy amongst the above two, which will be applied to those kinds of resources similar to the ones seen above for carrying out the given instruction block execution as ideally as possible.

Stages:

1. Start
2. For a particular processor , a cache management strategy(optimistic) is applied to the cache associated with it.
3. The instruction block is executed in the processor and the context is retrieved.
4. For another processor a pessimistic cache management strategy is applied to the cache associated with it.
5. The instruction block is executed in the second processor and the context is retrieved.
6. The page which indicates the optimum strategy is retrieved along with the context of execution of the instruction block.
7. End

CHAPTER 4

SYSTEM REQUIREMENTS SPECIFICATION

4.1 Introduction

The purpose of this document is to provide a comprehensive description 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 optimization techniques, as well as implementing Jon Bentley's rules to optimize code.

4.1.1 Project Scope

The tool we propose to build will be able to convert C source code into an optimized version by applying some common optimization techniques, as well as implementing Jon Bentley's rules to optimize code. The input to the tool will be C source code, and the generated output will be optimized C source code.

- The purpose of this tool is to achieve a performance boost in programs, while showing the optimizations on the source code that led to the same.

- The benefits of the tool are that :

    1. programmers can compare the input and output source codes, to understand how the code was optimized
    2. optimizations are guaranteed, unlike when passed through a compiler directly
    3. It proves for a reduction in computational power usage
    4. Better insight into programming practices that can lead to performance boost

- The objectives are:

    1. Implement a minimum of ten rules of optimizations
    2. Test against sample programs, for validating automated optimization

3. Test for logical equivalence of input and output programs, by passing both through the gcc compiler
4. Profiling the input and output codes to measure the change in metrics such as compile time and run time

- The goals are:

1. Automated source code optimization
2. Profiling to check for improvement in performance

4.2 Product Perspective

The idea for this tool originated from Jon Bentley's rules and Alexandrascu's guidelines. The uniqueness of this tool lies in the fact that it performs a source-to- source transformation of the code, showing the optimizations. The fact that the generated output is source code provides for better insights and experimentations by the programmer.

4.2.1 Product Features

There are three primary features of this tool:

1. View the optimized source code: The tool will allow the programmer to view the source code after optimization in a readable, indented format.
2. Profiling: We will use the valgrind tool to point out the optimizations applied to certain parts of code. Along with this, the change in metrics for the input and output source codes will be highlighted.
3. The optimizations will be grouped into multiple levels, allowing the user to customize them as per their preference.

4.2.2 Operating Environment

- Linux/Unix based distributions with 1GB of RAM

4.2.3 General Constraints, Assumptions and Dependencies

Dependencies:

- python3.6 or higher
- clang-tidy
- indent for Linux and gindent for MacOSx
- valgrind and kcachegrind
- gcc compiler

4.2.4 Risks

During this phase, we are implementing the general optimisation guidelines, as well as some guidelines for optimization proposed by Jon Bentley. We are operating under the assumption that the modifications we are introducing in the source code are not conflicting with each other, and are indeed going to boost the performance of the code. This is a potential risk, because the performance boost achieved by a rule could be lowered when another rule is applied, which is not being checked, at this stage.

4.3 Functional Requirements

Sequence of actions:

- Provide an interface for the user to specify the chosen optimization level, as well as the input source file.
- Check the program for syntax and semantic errors by passing it through a compiler.
- In the case of errors, immediate termination.
- Pre-processing the input source code for further stages.
- Running the source code through the lexer and parser to generate an intermediate representation.
- Perform optimizations while and after parsing.
- Generate output source code from the intermediate representation.
- Output profiling results.

4.4 External Interface Requirements

4.4.1 User Interfaces

- Online GUI
- Generated source code and profiled results are produced within second scale latencies
- Error messages are displayed on the GUI

4.4.2 Hardware Requirements

- Monitor
- Keyboard
- Basic computer peripherals

4.4.3 Software Requirements

Operating Systems:

- Unix/Linux based operating systems

Tools:

- ply3.11
- clang-tidy LLVM10.0.0
- gcc
- python3.6 or higher
- indent

Python Libraries:

- copy
- re
- uuid
- collections.abc
- sys

- math

4.5 Non-Functional Requirements

4.5.1 Performance Requirements

- The entire input program must fit into memory.
- The output source code must display significant improvement in performance metrics as compared to the input source code when compiled and run.
- The output must be generated within second scale latencies.

4.5.2 Safety Requirements

- The output source code must be logically equivalent to the input source code.

CHAPTER 5

SYSTEM DESIGN

5.1 Design Considerations

5.1.1 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.

5.1.2 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 be potentially solved using cloud services with ease.

5.1.3 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.

5.2 High Level System Design

5.2.1 Component Diagram:

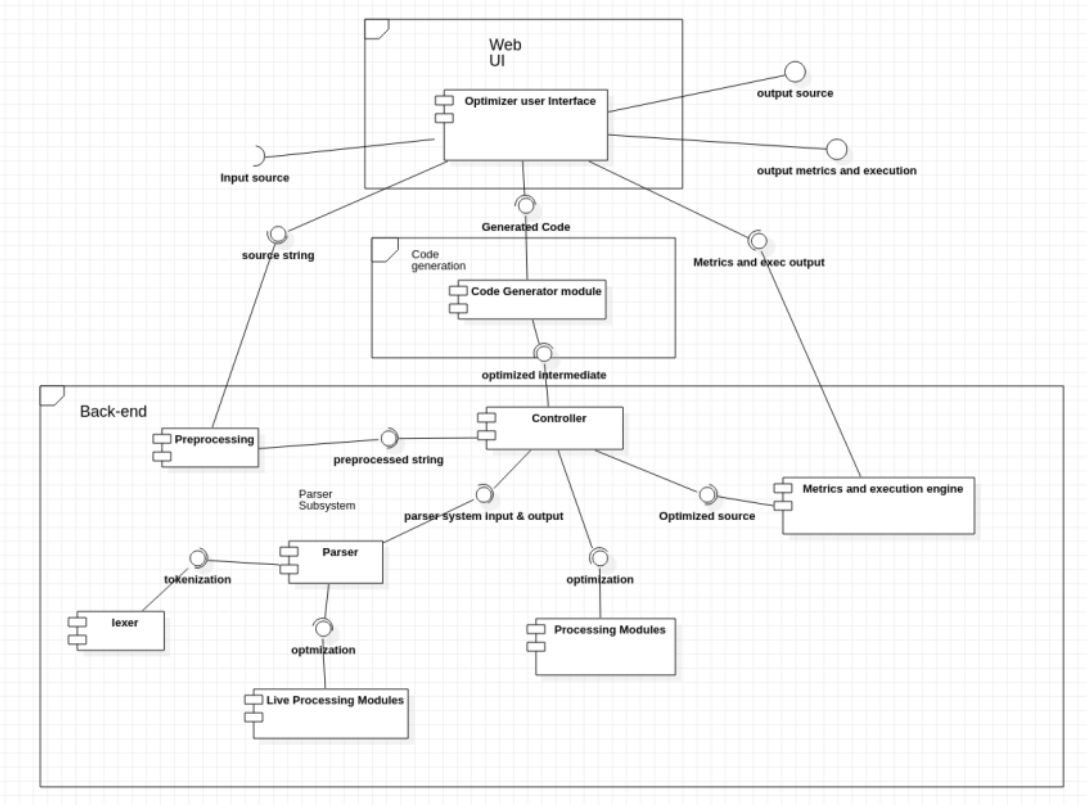

fig 1

5.3 Design Description

5.3.1 Class diagram :

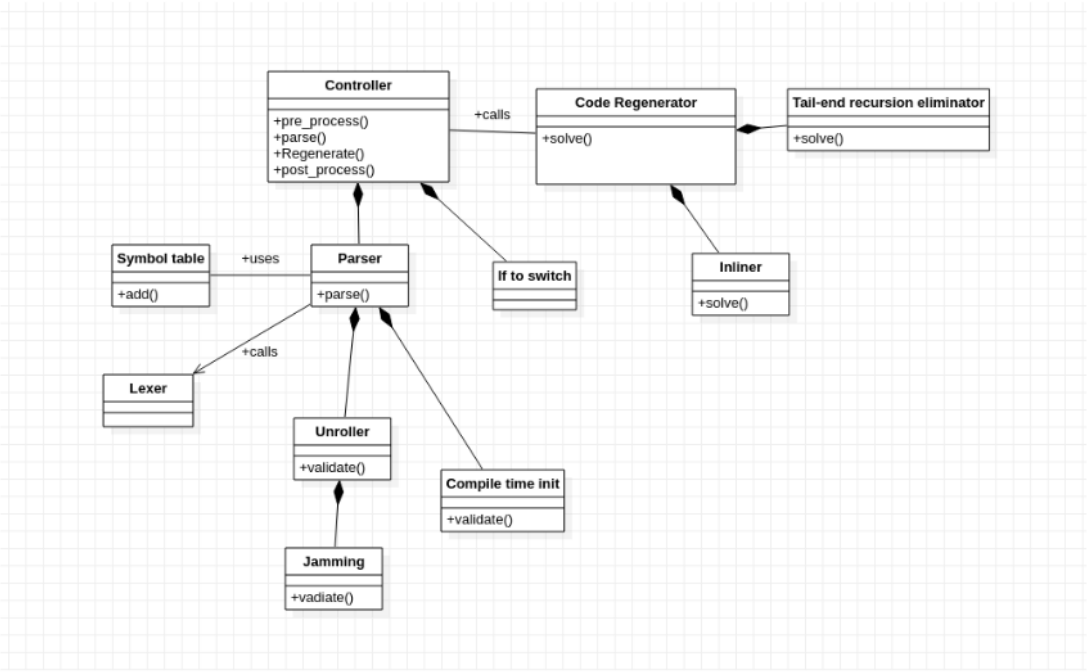

fig 2

5.3.2 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.

5.4 State diagram

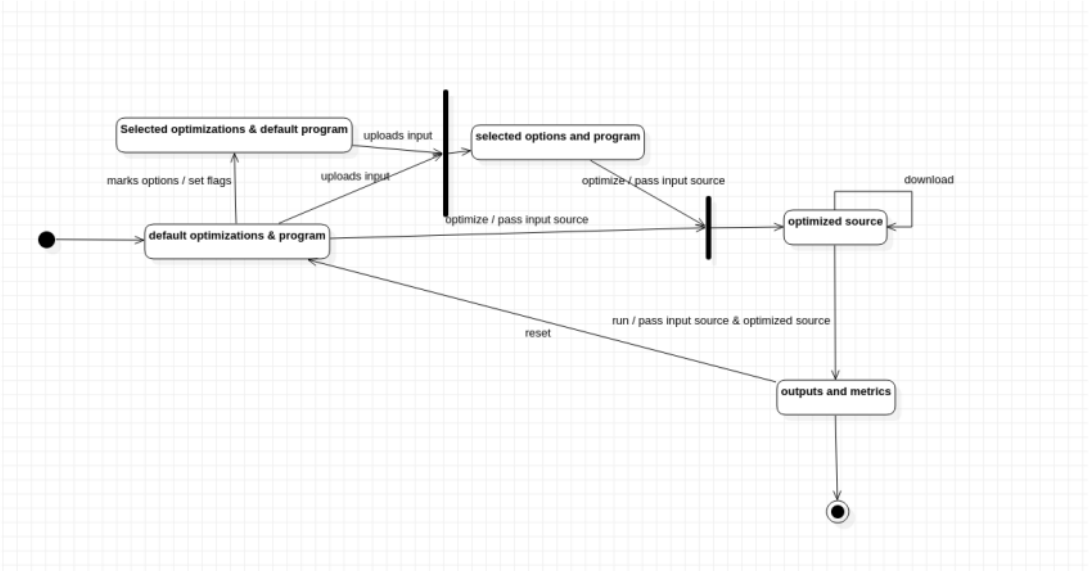

fig 3

5.5 User Interface

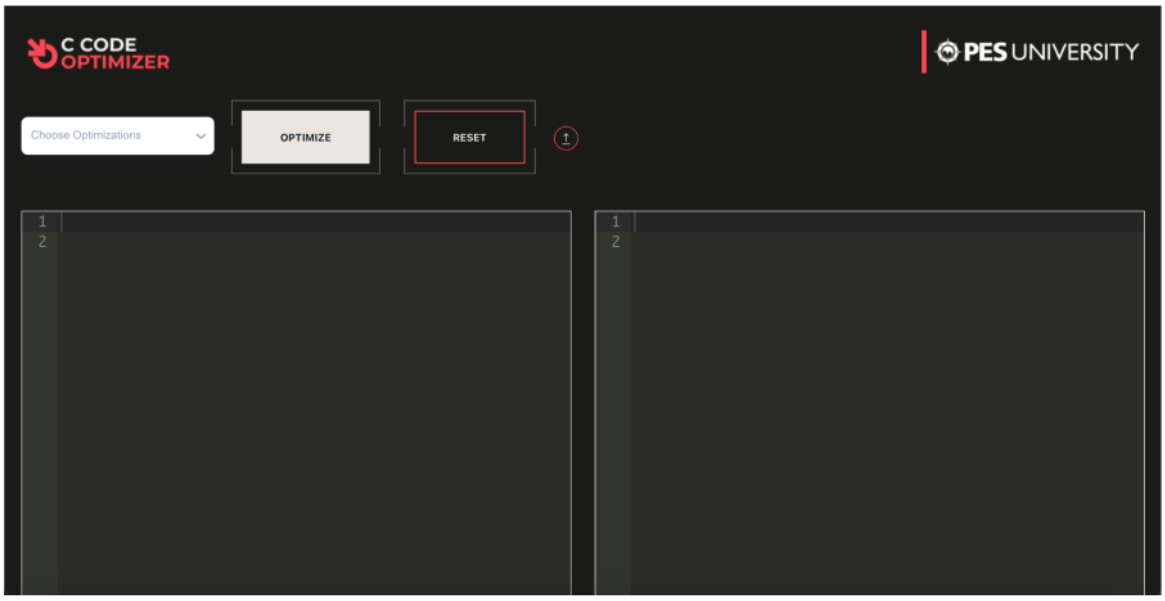

fig 4.1

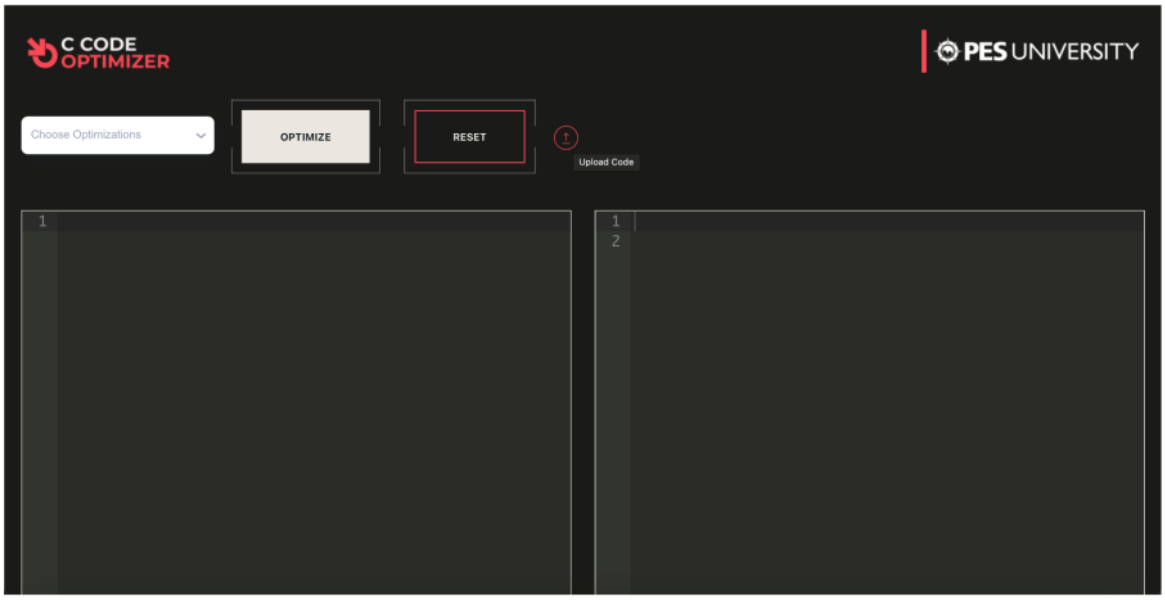
C CODE OPTIMIZER
PES UNIVERSITY
Choose Optimizations
OPTIMIZE
RESET
Upload Code

fig 4.2

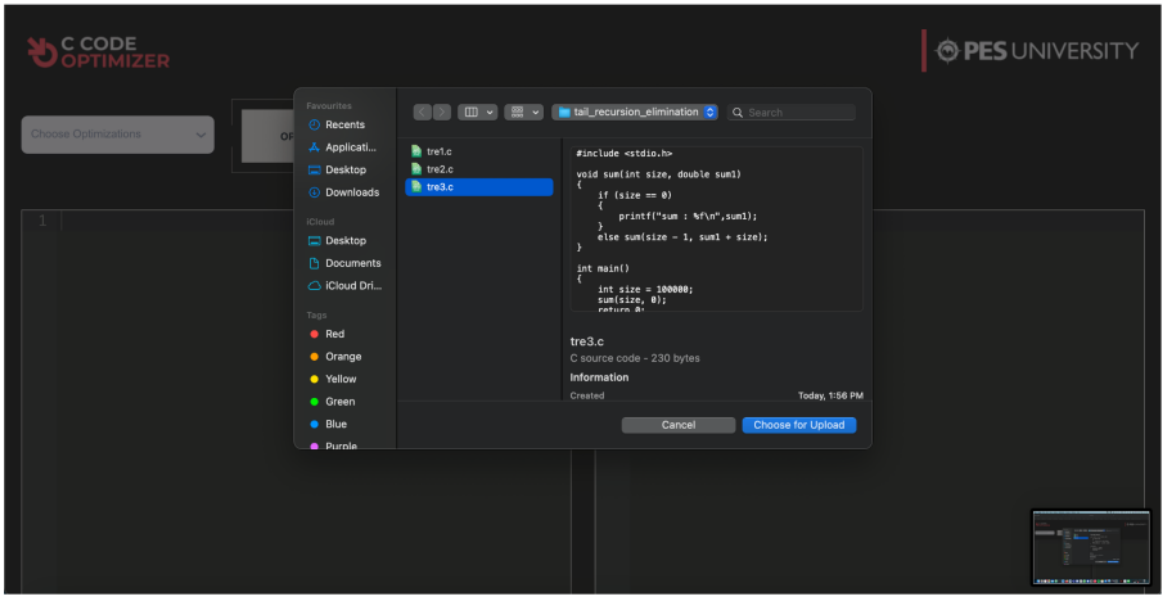
C CODE OPTIMIZER
PES UNIVERSITY
Choose Optimizations
Favourites
Recents
Applicati...
Desktop
Downloads
iCloud
Desktop
Documents
iCloud Dri...
Tags
Red
Orange
Yellow
Green
Blue
Purple
tail_recursion_elimination
Search
tre1.c
tre2.c
tre3.c
#include <stdio.h>
void sum(int size, double sum1)
{
if (size == 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

fig 4.3

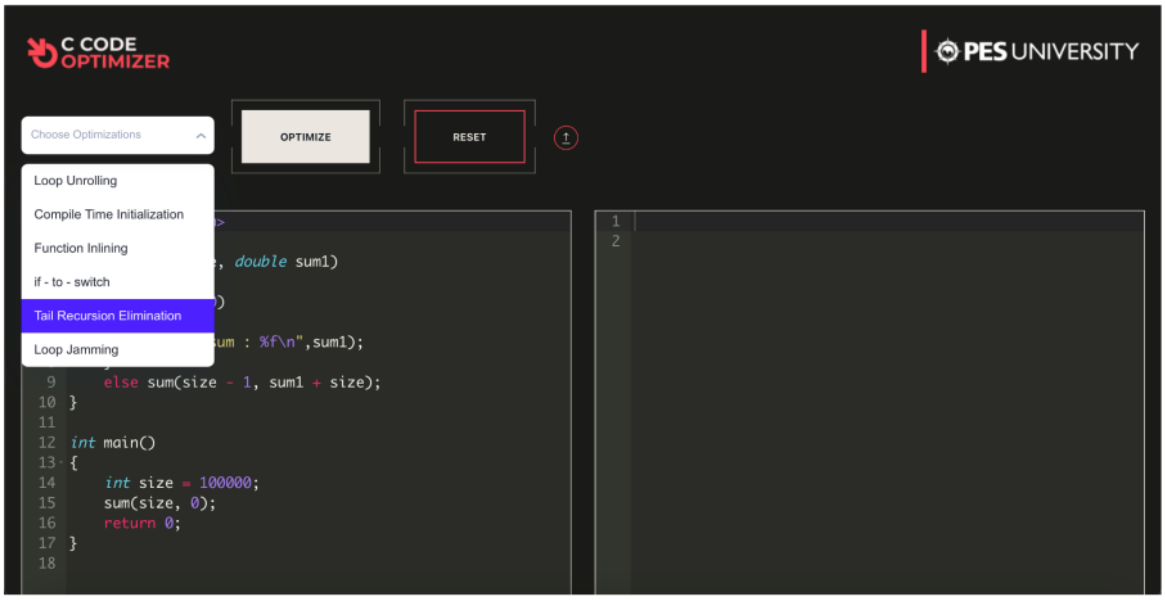
C CODE OPTIMIZER
PES UNIVERSITY
Choose Optimizations
OPTIMIZE
RESET
Loop Unrolling
Compile Time Initialization
Function Inlining
if - to - switch
Tail Recursion Elimination
Loop Jamming
, double sum1)
um : %f\n",sum1);
else sum(size - 1, sum1 + size);
}
int main()
{
int size = 100000;
sum(size, 0);
return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

fig 4.4

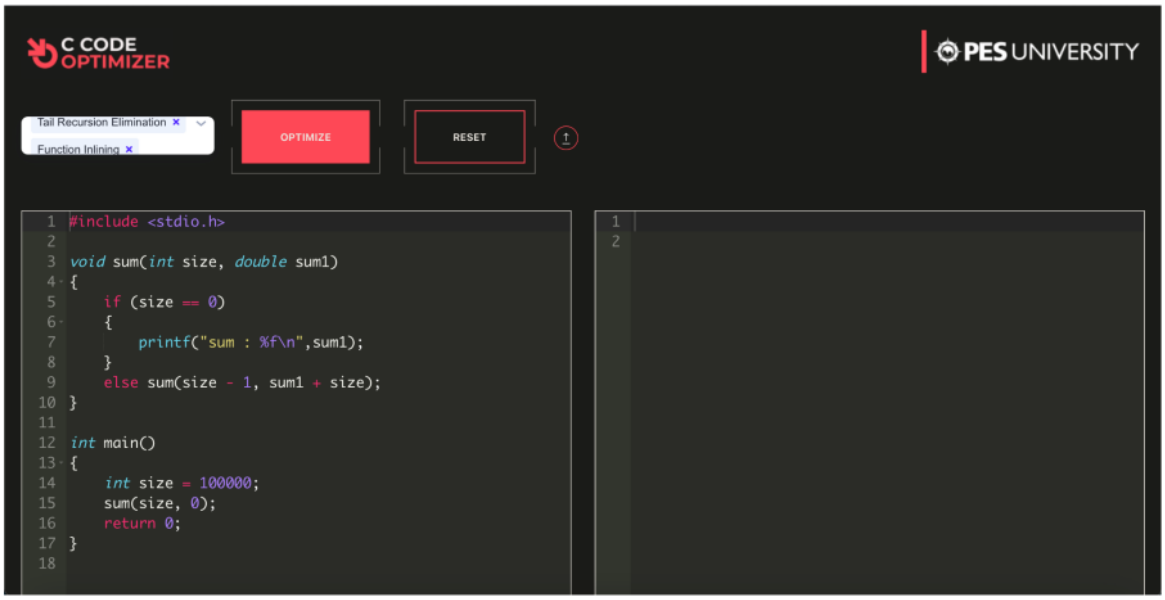
C CODE OPTIMIZER
PES UNIVERSITY
Tail Recursion Elimination
Function Inlining
OPTIMIZE
RESET
#include <stdio.h>
void sum(int size, double sum1)
{
if (size == 0)
{
printf("sum : %f\n",sum1);
}
else sum(size - 1, sum1 + size);
}
int main()
{
int size = 100000;
sum(size, 0);
return 0;
}

fig 4.5

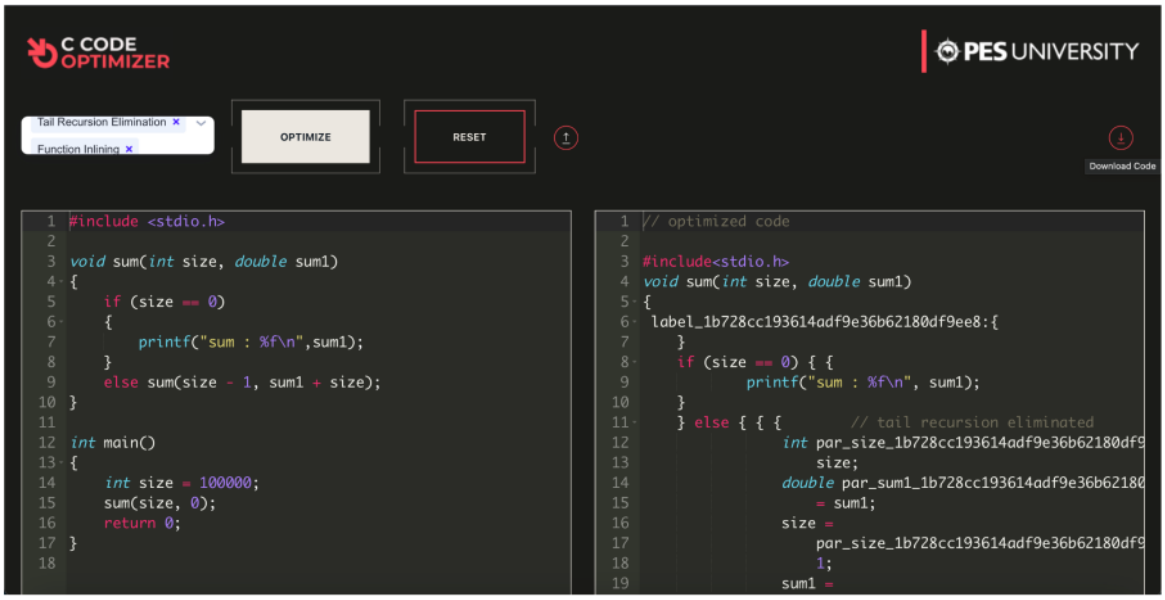
C CODE OPTIMIZER
PES UNIVERSITY
Tail Recursion Elimination
Function Inlining
OPTIMIZE
RESET
Download Code
#include <stdio.h>
void sum(int size, double sum1)
{
if (size == 0)
{
printf("sum : %f\n",sum1);
}
else sum(size - 1, sum1 + size);
}
int main()
{
int size = 100000;
sum(size, 0);
return 0;
}
//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

fig 4.6

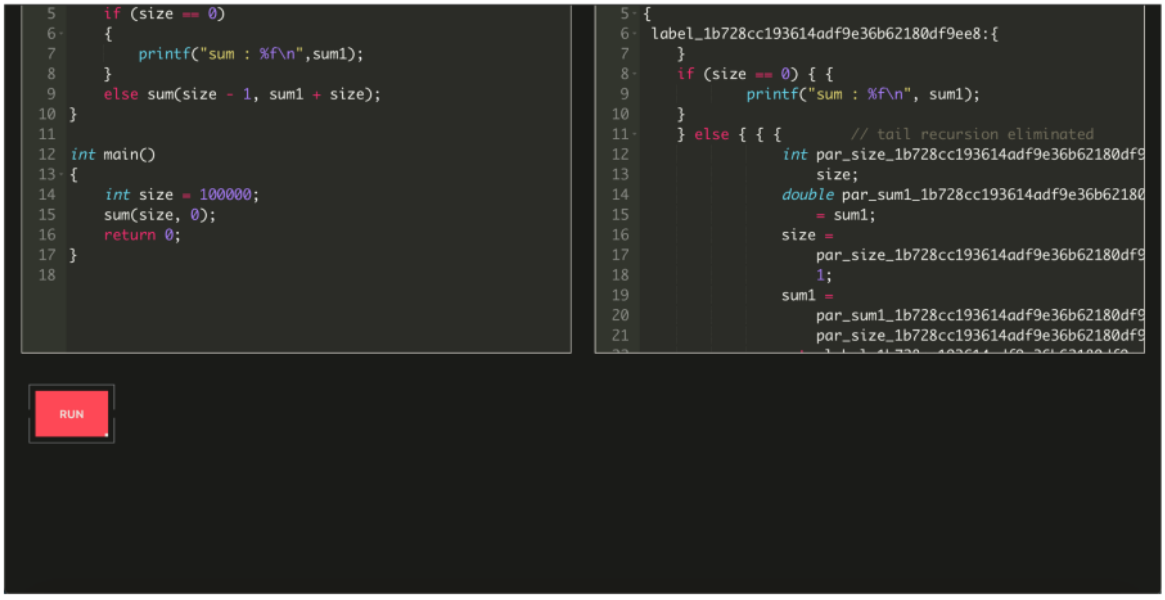
if (size == 0)
{
printf("sum : %f\n",sum1);
}
else sum(size - 1, sum1 + size);
}
int main()
{
int size = 100000;
sum(size, 0);
return 0;
}
{
label_1b728cc193614adf9e36b62180df9ee8:{
}
if (size == 0) { {
printf("sum : %f\n", sum1);
}
} else { { {
// tail recursion eliminated
int par_size_1b728cc193614adf9e36b62180df9
size;
double par_sum1_1b728cc193614adf9e36b62180
= sum1;
size =
par_size_1b728cc193614adf9e36b62180df9
1;
sum1 =
par_sum1_1b728cc193614adf9e36b62180df9
par_size_1b728cc193614adf9e36b62180df9
RUN

fig 4.7

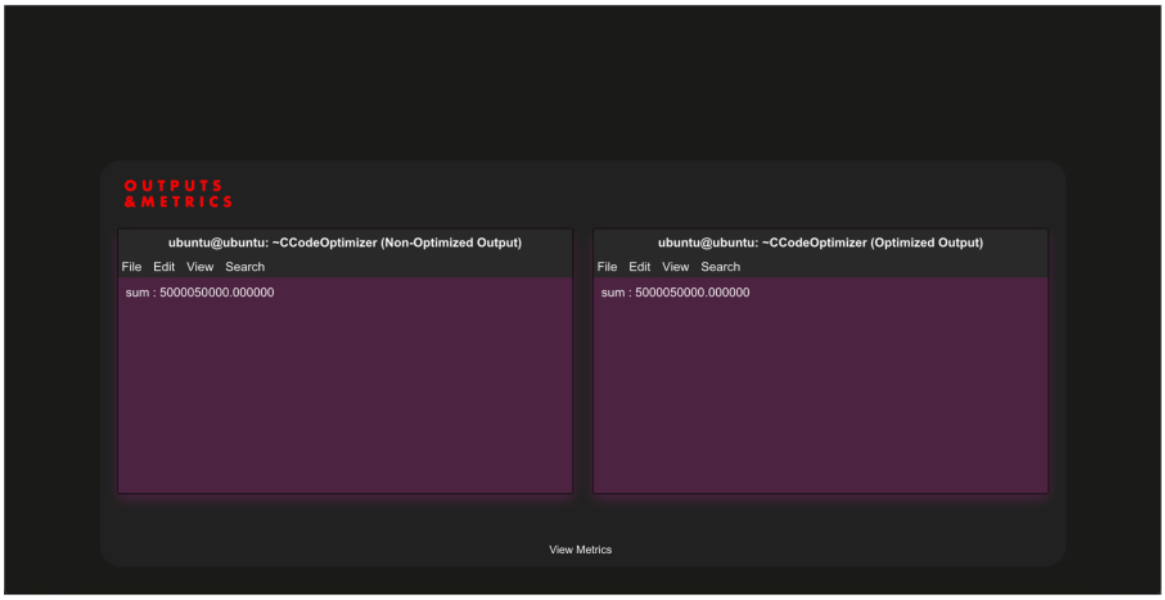
OUTPUTS
&METRICS
ubuntu@ubuntu: ~CCodeOptimizer (Non-Optimized Output)
File Edit View Search
sum : 5000050000.000000
ubuntu@ubuntu: ~CCodeOptimizer (Optimized Output)
File Edit View Search
sum : 5000050000.000000
View Metrics

fig 4.8

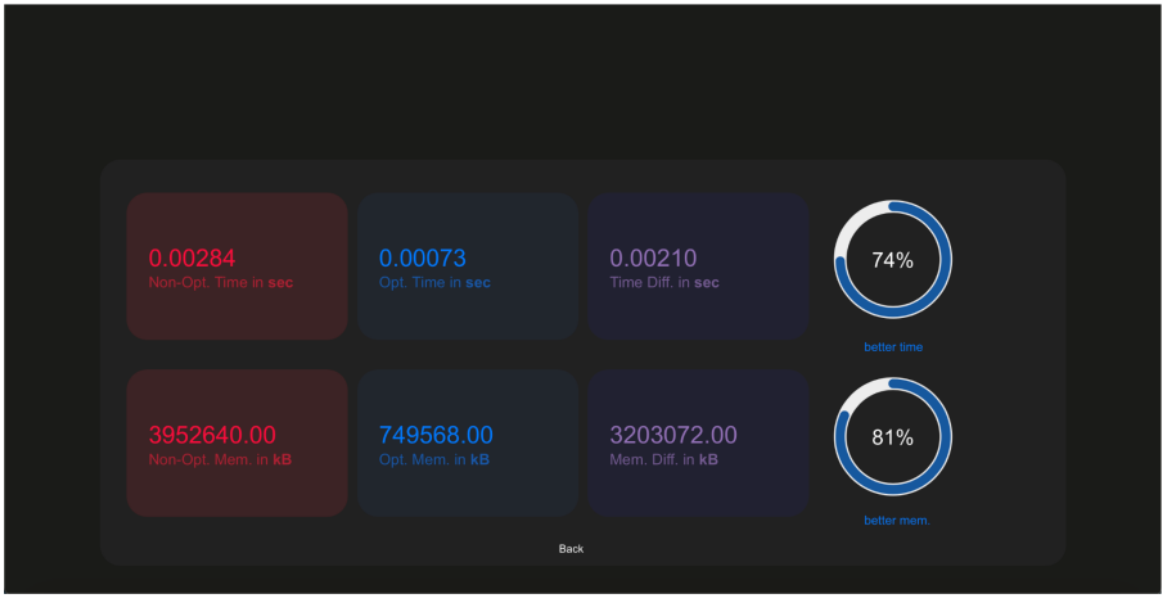
0.00284
Non-Opt. Time in sec
0.00073
Opt. Time in sec
0.00210
Time Diff. in sec
74%
better time
3952640.00
Non-Opt. Mem. in kB
749568.00
Opt. Mem. in kB
3203072.00
Mem. Diff. in kB
81%
better mem.
Back

fig 4.9

5.6 External Interface diagram

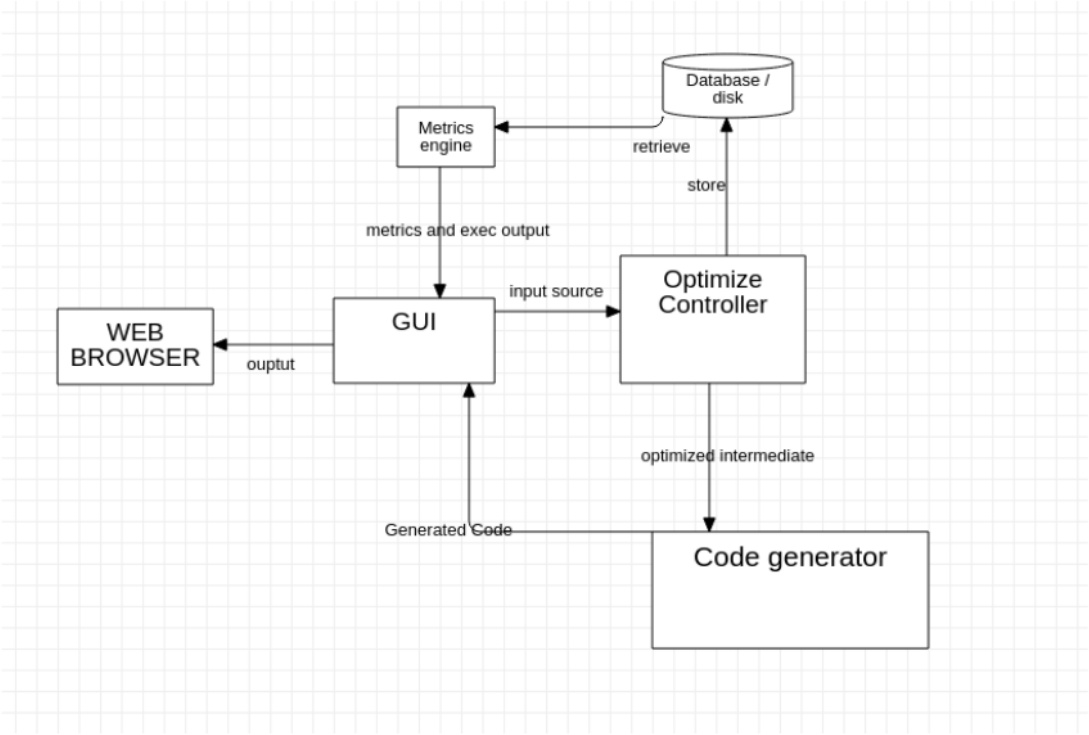

fig 5

5.7 Package Diagram

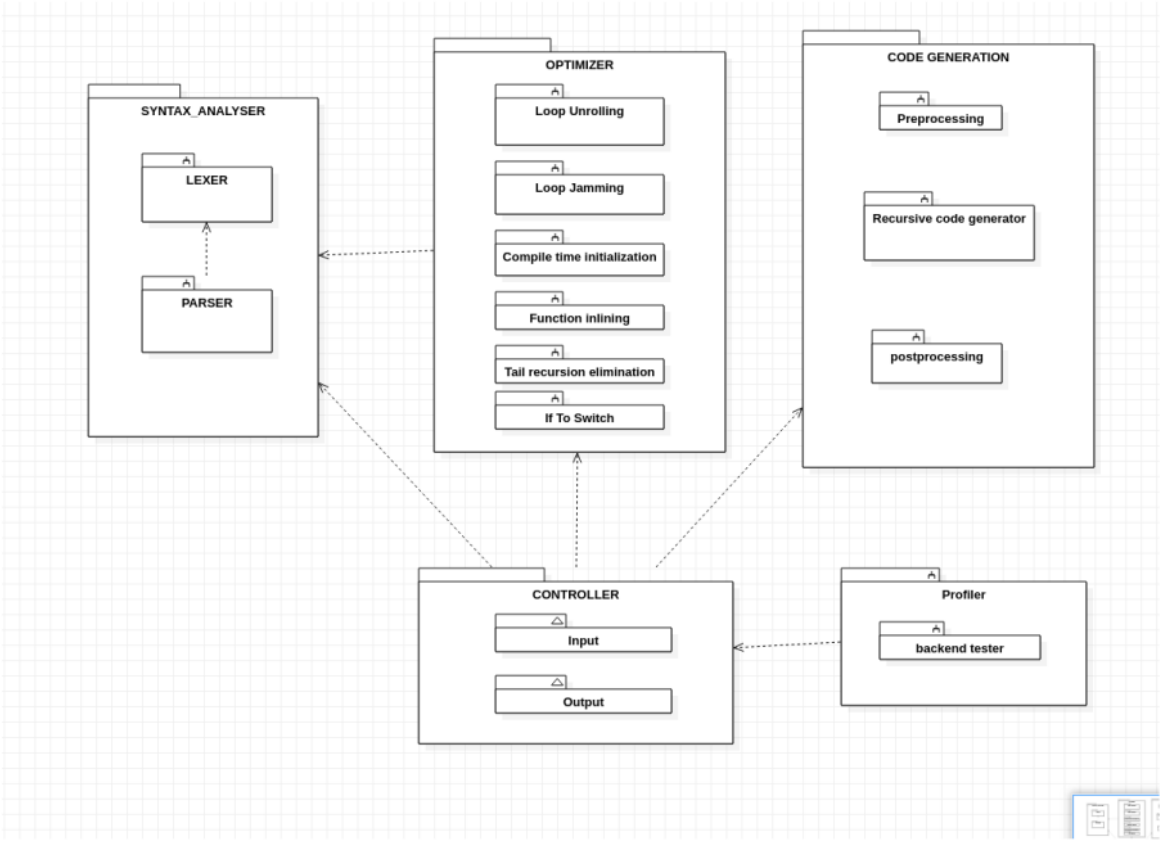

fig 6

5.8 User Help considerations:

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.

5.9 Design Details :

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

- Innovativeness - Completely innovative in terms of implementation, given that there is no existing open source codebase for these optimizations.
- Interoperability - The UI can be accessed by the user through any system as long as it has a modern web browser.
- Performance - The tool is expected to deliver results in second scale latencies for the average input program size.
- Reliability - The tool will not change the underlying logic of the input program provided that the user inputs syntactically and semantically correct programs.
- Maintainability - The inherent code base is easy to maintain as it is modular and an object based model.
- 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.
- Reusability - Code base consists of a few modules that can prove to be useful in other applications.

CHAPTER 6

PSEUDOCODE

Pseudocode for controller.py :

1) input_prg ← read(input file)
2) input_flags ← read(flags)
3) Preprocess(input_program)
4) parse(input_prg) ← PARSE_TREE (Perform Loop based optimizations while parsing if Flags are set)
5) tail_rec_eli_solve(PARSE_TREE) (Performs optimizations on the parse tree)
6) fn_inline_solve(PARSE_TREE) (Performs optimizations on the parse Tree)
7) output_program = solve(PARSE_TREE) (Performs optimizations on the parse Tree)
8) make_compile_time_inits(output_program)
9) Postprocess(output_program)

Pseudocode for Code generator module (solve function) :

Optimized Parse Tree → Solve Function → Optimized Source Code

```
solve (start_index, end_index, PARSE_TREE, output_program) :
    if start_index=end_index // base case 1
        stop recursion
    if type (PARSE_TREE[start_index]) = 'string'  // base case 2
        if (PARSE_TREE[start_index] is a keyword)
            output_program.append (PARSE_TREE[start_index] + space_char)
        else
            output_program.append (PARSE_TREE[start_index])
    if type (PARSE_TREE[start_index]) = 'int'   // base case 3
        output_program.append (string(PARSE_TREE[start_index])
    if type (PARSE_TREE[start_index]) = 'list')
```

solve (0, length(PARSE_TREE[start_index]), PARSE_TREE, output_program)

solve (start_index+1, end_index, PARSE_TREE, output_program) // continuing the recursion

CHAPTER 7

CONCLUSIONS FOR CAPSTONE PROJECT PHASE- I

The progress that has been achieved so far is :

1. Completed research on several topics in the field of optimization
2. Completed manual assertion of performance boost provided by implementing Bentley's rules
3. Completed development of overall structure of the tool and assembly of skeletal components
4. Completed development of lexical analyser, parser as well as controller modules
5. Began development of a robust front end and graphical user interface
6. Began partial implementations for six of Bentley's rules for optimizing work
7. Began rigorous testing of implemented rules on several test cases
8. Began systematic documentation of test cases.
9. Began development of **code generator** module
10. Began rigorous benchmarking of optimization applicable source codes.

CHAPTER 8

PLAN OF WORK FOR CAPSTONE PROJECT PHASE-2

The expected progress to be achieved in the next phase of the capstone project:

1. Refining and completely implementing all of the 6 ongoing rules being implemented.
2. Hopefully implementing 4 more of applicable Bentley's Rules.
3. Enhance the existing Graphical User Interface and extend its functionality.
4. Refining and optimizing Code generator to achieve faster and more readable output.
5. Exploring the possibility of integrating already existing helper tools to enhance performance
6. Conduction of rigorous unit and integration testing.
7. Conduction of user acceptance testing.
8. Refining and refactoring of the tester module.
9. Providing a catalogue of popular use cases on which optimizations prove to be highly impactful.
10. Continued organised and systematic documentation of the progress.

REFERENCE / BIBLIOGRAPHY

[1] Paul B Schneck, A Survey Of Compiler Optimization Techniques, https://ntrs.nasa.gov/api/citations/19730021416/downloads/19730021416.pdf

[2] Michael E. Lee, Optimization of Computer Programs in C, http://icps.u-strasbg.fr/~bastoul/local_copies/lee.html

[3] Medium, gcc vs clang-llvm, https://alibabatech.medium.com/gcc-vs-clang-llvm-an-in-depth-comparison-of-c-c-compilers-899ede2be378

[4] PVS-Studio Analyser, https://www.viva64.com/en/pvs-studio/

[5] CPP:Depend, https://www.cppdepend.com/

[6] Helix QAC, https://www.perforce.com/products/helix-qac

[7] Bran Ferren, W.Daniel Hills, Nathan P, Clarence T, Lowell L Jr., Optimization Of Code Through Hardware Resource Policies

APPENDIX : DEFINITIONS ACRONYMS AND ABBREVIATIONS

Abbreviations:

OS - Operating Systems

UI - User Interface

AST - Abstract Syntax Tree

Definitions:

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

Paper1

ORIGINALITY REPORT

5%	4%	2%	3%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	content.yudu.com Internet Source	2%
2	www.termpaperwarehouse.com Internet Source	1%
3	www.scribd.com Internet Source	1%
4	Paul B. Schneck. "A survey of compiler optimization techniques", Proceedings of the annual conference on - ACM 73 ACM 73, 1973 Publication	<1%
5	alibabatech.medium.com Internet Source	<1%
6	Submitted to Georgia Institute of Technology Main Campus Student Paper	<1%
7	onlinelibrary.wiley.com Internet Source	<1%
8	Paul B. Schneck. "A survey of compiler optimization techniques", Proceedings of the	<1%

annual conference on - ACM'73, 1973
Publication

9	www.fao.org Internet Source	<1%

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	On		