PROJECT REQUIREMENTS SPECIFICATION

Automated Tool for 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

TABLE OF CONTENTS

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.

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

- The objectives are
  - implement a minimum of ten rules of optimizations
  - test against sample programs, for validating automated optimization
  - test for logical equivalence of input and output programs, by passing both through the gcc compiler
  - profiling the input and output codes to measure the change in metrics such as compile time and run time

- The goals are
  - automated source code optimization

- ○ profiling to check for improvement in performance

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.

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.

2.2. Operating Environment

- Linux/Unix based distributions with 1GB of RAM

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

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.

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. External Interface Requirements

4.1. User Interfaces

- Unix/Linux CLI

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

4.2. Hardware Requirements

- Monitor
- Keyboard
- Basic computer peripherals

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

5. Non-Functional Requirements

5.1. Performance Requirement

- 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.

5.2. Safety Requirements

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

Appendix A: Definitions, Acronyms and Abbreviations

1. AST - Abstract Syntax Tree
2. CLI - Command Line Interface
3. ply - Python Lex and Yacc

Appendix B: References

- https://pypi.org/project/ply/
- Linux man page
- https://www.gnu.org/software/indent/