A SYNOPSIS ON

CODE OPTIMIZATION TOOL

Submitted in partial fulfilment of the requirement for the award of the degree of

BACHELOR OF TECHNOLOGY

In

Computer Science & Engineering

Submitted by:

Utkarsh Joshi	**2261582**
Harshita Joshi	**2261258**
Ravi Mahar	**2261464**
Vikash Bhaisora	**2261605**

Under the Guidance of

Akshay Choudhary

Assistant Professor

Project Team ID: 84

Department of Computer Science & Engineering

Graphic Era Hill University, Bhimtal, Uttarakhand

March-2025

CANDIDATE'S DECLARATION

We hereby certify that the work which is being presented in the Synopsis entitled **"Code Optimization Tool"** in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in Computer Science & Engineering of the Graphic Era Hill University, Bhimtal campus and shall be carried out by the undersigned under the supervision of **Akshay Choudhary**, **Assistant Professor**, Department of Computer Science & Engineering, Graphic Era Hill University, Bhimtal.

Utkarsh Joshi	**2261582**
Harshita Joshi	**2261258**
Ravi Mahar	**2261464**
Vikash Bhaisora	**2261605**

The above-mentioned students shall be working under the supervision of the undersigned on the **"Code Optimization Tool".**

Signature

Supervisor

Signature

Head of the Department

<u>Internal Evaluation (By DPRC Committee)</u>

Status of the Synopsis: Accepted / Rejected

Any Comments:

Name of the Committee Members: **Signature with Date**

1.

2.

Table of Contents

Chapter 1

1.1 Introduction

Code optimization is a crucial phase in compiler design that enhances the efficiency of the generated machine code without altering the program's intended functionality. The primary objective of this phase is to minimize resource consumption—such as execution time, memory usage, and power consumption—while maintaining correctness. Optimized code is essential for modern computing environments, where inefficiencies can cause significant performance bottlenecks, especially in large-scale software systems, embedded systems, and high-performance computing applications.

Modern compilers implement a variety of optimization strategies, categorized into machine-independent and machine-dependent optimizations. Machine-independent optimizations, such as **constant folding, common subexpression elimination, and dead code elimination**, improve the intermediate representation (IR) of the program. Meanwhile, machine-dependent optimizations, like **instruction scheduling and register allocation**, tailor the compiled code for specific processor architectures.

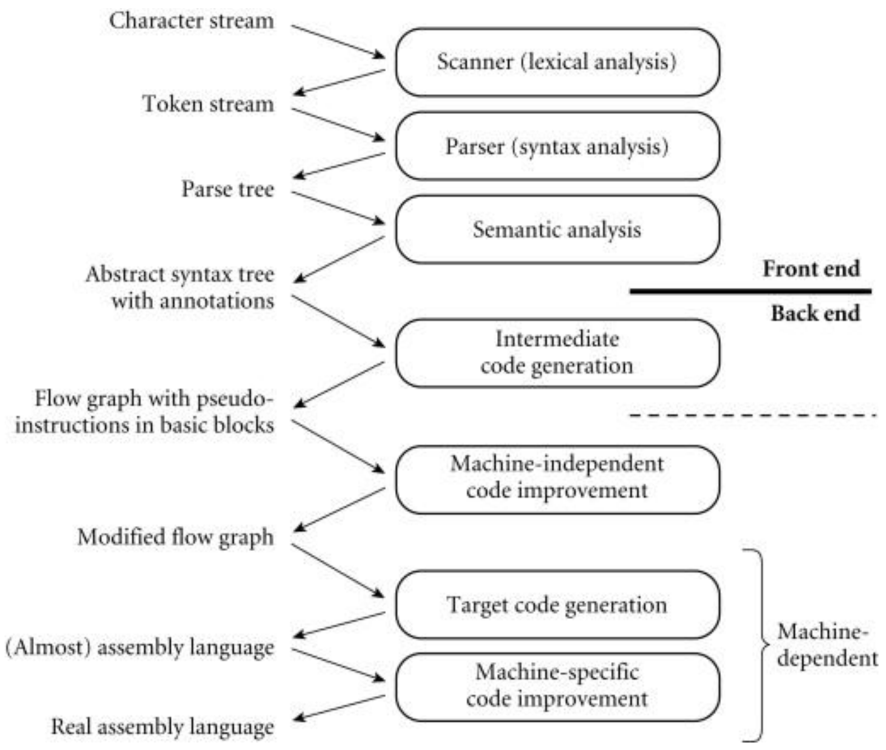
As estimated by A.J. Smith, inefficient code generation can lead to a significant increase in execution time and resource consumption. For instance, redundant computations, unused variables, and poorly structured loops can degrade application performance, making them inefficient and costly. These issues are especially critical in large-scale and real-time systems, where every optimization directly impacts user experience and operational efficiency.

Figure 1.1 illustrates the role of the optimization phase in the compiler design pipeline. The optimization phase typically follows intermediate representation (IR) generation and precedes final code generation. By applying optimizations at this stage, compilers can ensure that the final executable is **highly efficient and tailored to the target hardware architecture**.

To address these challenges, we are developing a **Code Optimization Tool** as part of our project. This tool aims to enhance the efficiency of compiled programs by implementing various optimization techniques at different stages of the compilation process. By focusing on reducing redundant computations, optimizing memory usage, and improving execution speed, our tool will contribute to generating highly efficient machine code.

Our approach involves integrating traditional optimization techniques such as **loop unrolling, dead code elimination, and common subexpression elimination**, alongside modern strategies like **profile-guided optimization (PGO) and instruction scheduling**. The tool will be designed to work seamlessly with existing compiler architectures, providing a flexible and scalable solution for improving software performance.

With this project, we aim to contribute to compiler optimization research and create a practical solution that enhances the execution efficiency of programs across different hardware platforms.



Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis
Abstract syntax tree with annotations
Front end
Back end
Intermediate code generation
Flow graph with pseudo-instructions in basic blocks
Machine-independent code improvement
Modified flow graph
Target code generation
(Almost) assembly language
Machine-specific code improvement
Real assembly language
Machine-dependent

Figure 1.1

1.2 Problem Statement

In modern computing, inefficiently compiled code can significantly impact program execution speed, memory usage, and overall system performance. Poorly optimized code leads to **redundant computations, excessive memory consumption, and inefficient control flow**, increasing execution time and resource utilization. These inefficiencies are particularly problematic in large-scale applications, embedded systems, and high-performance computing, where even minor improvements in efficiency can lead to substantial performance gains.

Traditional compiler optimization techniques exist, but they are often **generic and not tailored for specific applications or architectures**. As a result, programs may still contain unnecessary operations, redundant instructions, and suboptimal memory access patterns, leading to performance bottlenecks. Furthermore, developers often rely on manual optimizations, which are time-consuming, error-prone, and difficult to scale across different architectures.

Key Challenges in Code Optimization:

1. Redundant Computations – Unnecessary repeated calculations slow down execution and waste processing power.
2. Dead Code & Unused Variables – Code that does not contribute to the final output remains in the program, increasing size and memory usage.
3. Inefficient Loop Structures – Poorly optimized loops lead to excessive iterations, impacting execution speed.
4. Suboptimal Register Allocation – Improper assignment of registers results in frequent memory access, slowing down performance.
5. Inefficient Memory Access Patterns – Poorly managed memory access can increase cache misses and degrade performance.
6. Lack of Hardware-Specific Optimizations – Generic compilation strategies do not fully exploit the capabilities of modern processors.

7. Manual Optimization Efforts – Developers often manually optimize code, which is time-consuming, error-prone, and non-scalable.

Chapter 2

Background and Literature Survey

2.1 Introduction

In the present times, research in compiler optimization focuses on improving code efficiency, reducing execution time, and optimizing memory usage. With the rapid advancements in processor architectures and the increasing demand for high-performance computing, modern compilers employ a variety of optimization techniques to generate efficient machine code. These optimizations can be broadly categorized into **machine-independent** and **machine-dependent** techniques.

In this, we review some of the major existing work in the field of **compiler optimization**, highlighting key techniques and their impact on performance. We also discuss recent advancements in optimization strategies and their relevance to our proposed **Code Optimization Tool**.

2.2 Compiler Optimization Techniques

Various studies have been conducted to enhance compiler optimization processes. The primary techniques used for optimization include:

2.2.1 Machine-Independent Optimizations

These optimizations are applied at the **intermediate representation (IR)** level and do not depend on the target machine architecture. Some key techniques include:

- Constant Folding: Replacing expressions with known constant values at compile time (e.g., replacing 3 + 5 with 8).
- Common Subexpression Elimination (CSE): Identifying and eliminating redundant expressions that are computed multiple times.
- Dead Code Elimination: Removing code that does not contribute to the final output of the program.
- Loop Optimizations: Techniques such as loop unrolling, loop fusion, and loop invariant code motion help reduce loop overhead and improve efficiency.

2.2.2 Machine-Dependent Optimizations

These optimizations depend on the **underlying hardware architecture** and are performed at the **assembly level** or during final code generation. Some examples include:

- **Register Allocation:** Efficiently using processor registers to minimize memory access.
- **Instruction Scheduling:** Reordering instructions to reduce pipeline stalls and improve execution efficiency.
- **Cache Optimization:** Optimizing memory access patterns to minimize cache misses and improve performance.

2.3 Existing Work in Code Optimization

2.3.1 Classic Compiler Optimizations

Smith et al. (1995) proposed various optimization strategies to enhance compiler performance. Their research demonstrated that **eliminating redundant computations and improving register allocation** could significantly boost execution efficiency.

Aho, Sethi, and Ullman (2006), in their book **"Compilers: Principles, Techniques, and Tools"**, discussed fundamental compiler optimization strategies, including **constant propagation, inline expansion, and strength reduction**. These techniques are widely implemented in modern compiler frameworks such as **LLVM and GCC**.

2.3.2 LLVM and GCC Optimization Frameworks

LLVM (Low-Level Virtual Machine) and GCC (GNU Compiler Collection) are two of the most widely used compiler infrastructures today.

- **LLVM Optimization Passes:** LLVM implements various optimization passes, such as **global value numbering (GVN), loop unrolling, and tail call elimination**, which enhance the efficiency of compiled code.
- **GCC Optimization Levels (-O1, -O2, -O3, -Ofast):** GCC provides multiple optimization levels, each offering different trade-offs between performance and compilation time.

Research by Chris Lattner (2004) on **LLVM's optimization pipeline** showed that a well-structured intermediate representation enables better optimization strategies compared to traditional compiler architectures.

2.4 Recent Advancements in Code Optimization

2.4.1 Profile-Guided Optimization (PGO)

Recent studies suggest that **profile-guided optimization (PGO)** can significantly improve performance by optimizing code based on runtime execution behaviour. Microsoft and Google have successfully integrated PGO into their compilers (MSVC and Clang), reducing execution time by up to **30%** in some cases.

2.4.2 Machine Learning-Based Optimizations

Recent research explores the use of machine learning models for predicting optimal compiler optimization sequences. For example:

- DeepTune (2018): A deep-learning-based system that predicts better compiler optimizations based on code patterns.
- Auto-Vectorization: AI-driven methods are being used to optimize loops for SIMD (Single Instruction, Multiple Data) execution, improving performance on modern processors.

2.5 Relevance to Our Code Optimization Tool

The insights from existing literature provide a strong foundation for our proposed **Code Optimization Tool**. Our tool will integrate both **traditional optimization techniques** (constant folding, dead code elimination) and **advanced strategies** (profile-guided optimization, instruction scheduling) to enhance compiler performance.

Key Contributions of Our Tool:

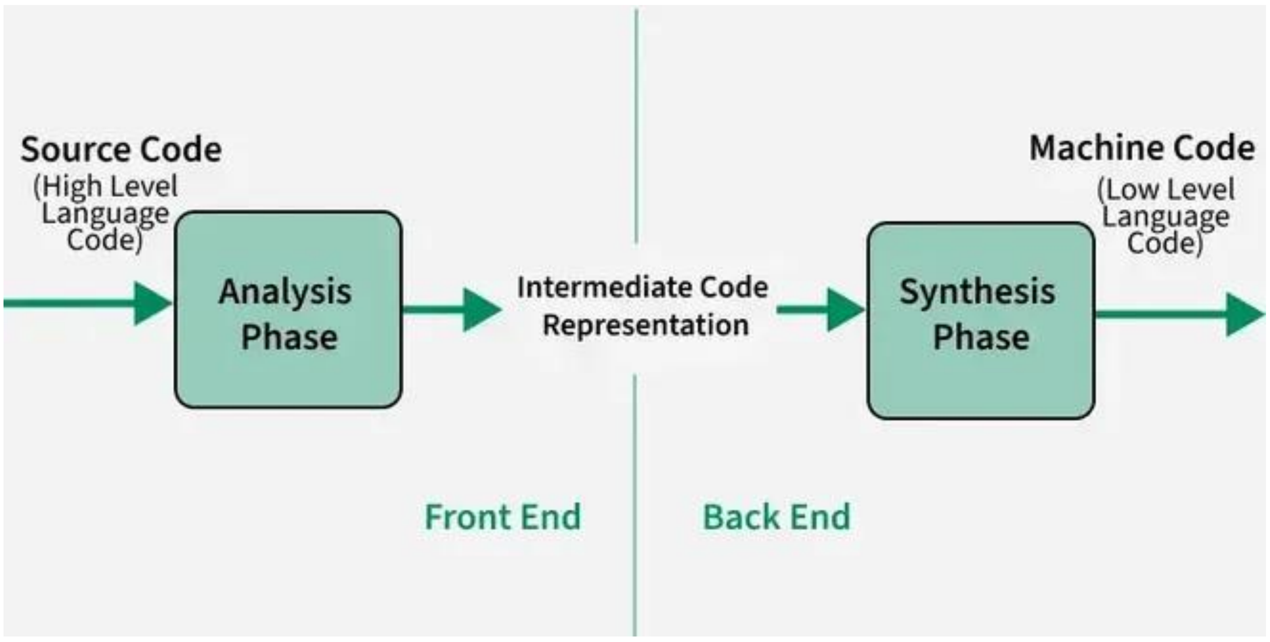
✓ Implements **machine-independent and machine-dependent optimizations** for better efficiency.
✓ Enhances loop optimizations and memory access patterns to improve execution speed.
✓ Provides a modular design that allows integration with existing compiler frameworks like **LLVM and GCC**.
✓ Potential for future extensions, such as **AI-driven compiler optimizations**.

Chapter 3

3.1 Objectives

The objectives of the proposed **Code Optimization Tool** are as follows:

1. Enhance Code Efficiency
    a. Develop an optimization tool that reduces redundant computations, minimizes execution time, and optimizes memory usage without altering program functionality.
2. Implement Key Optimization Techniques
    a. Integrate various machine-independent (constant folding, dead code elimination, loop optimization) and machine-dependent (register allocation, instruction scheduling) optimization techniques to improve overall compiler performance.
3. Improve Loop and Memory Optimization
    a. Optimize loop structures to reduce unnecessary iterations and enhance performance.
    b. Implement efficient memory access strategies to minimize cache misses and improve execution speed.
4. Ensure Scalability and Compatibility
    a. Design the tool to be modular and compatible with existing compiler frameworks like LLVM and GCC, ensuring easy integration and scalability.
5. Enable Profile-Guided Optimization (PGO)
    a. Implement profile-guided optimization (PGO) to analyse runtime behaviour and apply optimizations dynamically based on actual execution patterns.


Source Code
(High Level Language Code)
Analysis Phase
Intermediate Code Representation
Synthesis Phase
Machine Code
(Low Level Language Code)
Front End
Back End

Chapter 4

Hardware and Software Requirements

4.1 Hardware Requirements

Sl. No	Name of the Hardware	Specification
1	RAM	Min: 8 GB Recommeded:16gb or higher
2	Storage	Min: 256 GB SSD Recommended: 512 GB SSD or higher
3	Processor:	Min: Intel Core i5 (10th Gen) / AMD Ryzen 5 5000 series Recommended: Intel Core i7/i9 (12th Gen) or AMD Ryzen 7/9
4	Operating System:	Linux (Ubuntu 20.04+ / Fedora / Arch Linux) [Preferred] Windows 10/11 (for compatibility testing)

4.2 Software Requirements

Sl. No	Name of the Software	Specification
1	Programming Languages	Python (3.8 or later) – The primary language for developing the optimization tool. C/C++ – Required for integrating with low-level compiler frameworks like LLVM.
2	Libraries and Frameworks	LLVM & llvmlite – For intermediate representation (IR) generation and applying optimization passes. Python AST (ast module) – To analyze and transform Python source code. cProfile – For analyzing function execution time.

Chapter 5

5.1 Possible Approaches

5.1.1 Abstract Syntax Tree (AST)-Based Optimization

- **Approach:** The tool will first parse the source code into an **Abstract Syntax Tree (AST)** and traverse it to detect redundant computations, dead code, and loop inefficiencies.
- **Implementation:**
  - Python provides a built-in ast module for AST manipulation.
  - For C programs, pycparser will be used to analyze the AST.
- **Optimizations Implemented:**
  - **Constant Folding:** Evaluating constant expressions at compile time.
  - **Dead Code Elimination:** Removing code that has no effect on the final output.
  - **Strength Reduction:** Replacing expensive operations with equivalent cheaper ones (e.g., replacing x * 2 with x << 1).

5.1.2 Intermediate Representation (IR)-Based Optimization

- **Approach:** The tool will convert source code into **LLVM IR**, apply optimization passes, and generate optimized bytecode.
- **Implementation:**
  - Use llvmlite to generate and optimize IR.
  - Apply built-in LLVM optimization passes like **constant propagation**, **common subexpression elimination (CSE)**, and **loop unrolling**.
- **Optimizations Implemented:**
  - **Common Subexpression Elimination (CSE):** Removing redundant expressions to save computation.
  - **Loop Invariant Code Motion (LICM):** Moving loop-invariant computations outside loops.
  - **Instruction Reordering:** Reordering instructions to improve pipeline efficiency.

5.1.3 Profile-Guided Optimization (PGO)

- Approach: The tool will analyze runtime execution patterns using profiling tools and apply optimizations based on real execution data.
- Implementation:
  - Use Python's cProfile and line_profiler to gather execution statistics.
  - Apply targeted optimizations based on profiling data (e.g., inlining frequently executed functions).
- Optimizations Implemented:
  - Function Inlining: Reducing function call overhead by replacing function calls with actual code.
  - Branch Prediction Optimization: Reordering conditionals based on execution frequency.

5.1.4 Just-In-Time (JIT) Compilation Optimization

- Approach: Instead of static optimizations, apply runtime optimizations using Just-In-Time (JIT) compilation.
- Implementation:

    - Use Pyjion or Numba to compile Python code dynamically with optimizations.
    - Convert frequently executed code paths into optimized machine code.
- Optimizations Implemented:
    - Adaptive Optimization: Dynamically recompiling hot functions for better performance.
    - Loop Unrolling and Vectorization: Using SIMD instructions for faster execution.

5.2 Detailed Algorithms

5.2.1 Constant Folding Algorithm

Constant folding **evaluates constant expressions at compile time** to reduce runtime computation.

Algorithm:

1. Parse the source code into an **AST**.
2. Traverse the AST to find **arithmetic expressions** with constant values.
3. Evaluate the constant expression and replace it with its computed value.
4. Update the AST with the optimized expression.

Example:

Before: x = 3 * 5 # Evaluated at runtime

After: x = 15 # Computed at compile time

5.2.2 Dead Code Elimination Algorithm

Dead code elimination **removes variables and statements that do not affect the program's output**.

Algorithm:

1. Parse the source code into an **AST**.
2. Identify **variables and statements** that are never used.
3. Remove such statements from the AST.
4. Rewrite the optimized code.

Example:

```
Before: x = 10

y = 20  # 'y' is never used, so it is removed

print(x)

After: x = 10

print(x)
```

5.2.3 Common Subexpression Elimination (CSE) Algorithm

CSE **eliminates redundant computations** by storing already computed expressions.

Algorithm:

1. Convert the source code into an **Intermediate Representation (IR)**.
2. Identify duplicate expressions appearing **multiple times** in the IR.
3. Store the first computed result in a temporary variable.
4. Replace all subsequent occurrences with the stored variable.

Example:

Before: a = (x + y) * (x + y)

After:

t = (x + y)

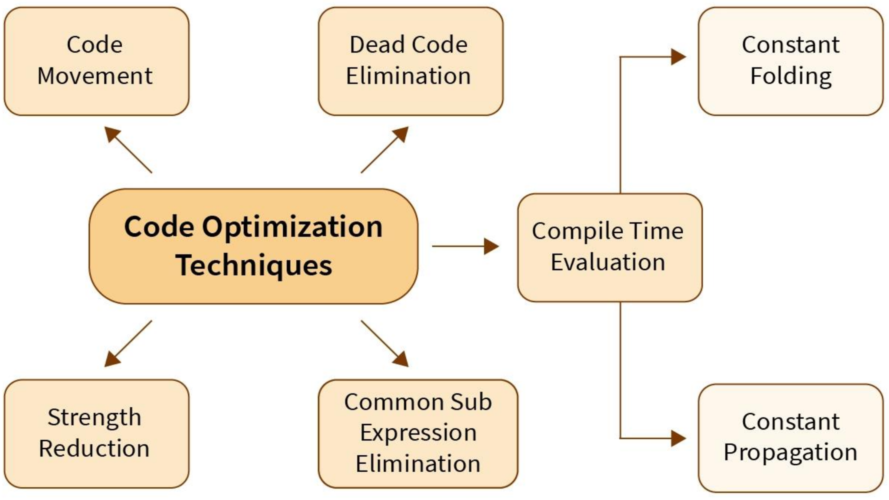
a = t * t

5.2.4 Loop Unrolling Algorithm

Loop unrolling **reduces loop overhead by executing multiple iterations per loop cycle**.

Algorithm:

1. Identify **simple loops with a fixed number of iterations**.
2. Unroll the loop by **manually replicating** the loop body for multiple iterations.
3. Reduce the number of loop iterations accordingly.



References

1. **Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D.** (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley.
    a. A foundational textbook covering compiler design and various optimization techniques.
2. **Muchnick, S. S.** (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
    a. Discusses advanced code optimization techniques, including intermediate representations and control flow analysis.
3. **Cooper, K. D., & Torczon, L.** (2011). *Engineering a Compiler (2nd Edition)*. Morgan Kaufmann.
    a. Explores compiler optimizations, register allocation, and instruction scheduling.
4. **Allen, R., & Kennedy, K.** (2002). *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann.
    a. Focuses on data dependence analysis and optimization techniques for modern processors.
5. **Srikant, Y. N., & Shankar, P.** (2009). *The Compiler Design Handbook: Optimizations and Machine Code Generation (2nd Edition)*. CRC Press.
    a. Covers both theoretical and practical aspects of compiler optimization strategies.
6. **Smith, A. J.** (1982). *Cache Memories*. ACM Computing Surveys, **14**(3), 473-530.
    a. Examines memory optimization techniques, crucial for improving execution speed in modern compilers.
7. **Frances Allen**. (2002). *Optimizing Compilers for Parallel Computing*. ACM Transactions on Programming Languages and Systems, **24**(3), 225–250.
    a. Discusses compiler techniques for optimizing parallel programs.
8. **LLVM Project**. (2024). *LLVM Compiler Infrastructure*. Retrieved from https://llvm.org
    a. Provides insights into IR-based optimizations using LLVM.
9. **Python AST Module**. (2024). *Python Abstract Syntax Tree Documentation*. Retrieved from https://docs.python.org/3/library/ast.html
    a. Reference for static analysis and code transformation using Python's AST module.
10. **LlvmLite Library**. (2024). *Lightweight LLVM Binding for Python*. Retrieved from https://github.com/numba/llvmlite

- Used for implementing intermediate representation (IR)-based optimizations in Python.