DEAD CODE ELIMINATION TOOL

Mini Project Study Plan

CS3501 - Compiler Design

SDG 13: Climate Action

Submitted By:
Pragadeeswaran T

814723104112

Computer Science and Engineering

2023 – 2027

1. PROJECT OVERVIEW

Dead Code Elimination (DCE) is a compiler optimization technique that removes code segments that do not affect program outcomes. This project aims to develop a tool that identifies and eliminates dead code, thereby reducing executable size, improving performance, and reducing energy consumption, which aligns with SDG 13: Climate Action by promoting energy-efficient software.

1.1 Project Objectives

- Understand the theoretical foundations of dead code elimination in compiler optimization
- Implement data flow analysis techniques including liveness analysis and reaching definitions
- Develop a working prototype that identifies unreachable code, unused variables, and redundant computations
- Demonstrate measurable improvements in code efficiency and energy consumption
- Align the project with environmental sustainability through SDG 13

1.2 Expected Outcomes

- A functional dead code elimination tool capable of analyzing source code

- Comprehensive documentation including literature review, technical design, and user manual
- Performance analysis demonstrating code optimization benefits
- Environmental impact assessment aligned with SDG 13

2. DETAILED WEEKLY STUDY PLAN

Week 1-2: Foundation and Literature Review

Topics to Study:

- Compiler phases: lexical analysis, syntax analysis, semantic analysis, code generation, and optimization
- Overview of compiler optimization techniques
- Introduction to dead code and its types:
    - Unreachable code (control flow based)
    - Unused variables and definitions
    - Redundant computations
    - Dead stores
- Control Flow Graph (CFG) concepts
- Basic blocks and flow graph representation

Learning Activities:

- Read Chapter 8 and 9 from "Compilers: Principles, Techniques, and Tools" by Aho, Sethi, and Ullman
- Study research papers on dead code elimination techniques from IEEE and ACM Digital Library
- Review existing compiler optimization tools such as GCC optimization flags (-O1, -O2, -O3) and LLVM optimization passes
- Analyz case studies of dead code in real-world software
- Document all findings in a structured literature review format

Deliverable: Literature Review Report (5-7 pages) covering theoretical foundations, existing techniques, and proposed approach

Week 3: Data Flow Analysis - Part 1

Topics to Study:

- Introduction to data flow analysis framework
- Reaching definitions analysis
- Use-definition (UD) chains
- Definition-use (DU) chains
- Forward data flow analysis

- Backward data flow analysis
- Iterative algorithms for solving data flow equations

Learning Activities:

- Work through manual examples of reaching definitions on sample code
- Study the iterative algorithm for computing data flow information
- Practice constructing Control Flow Graphs from given code snippets
- Understand GEN and KILL sets for different types of analyses
- Solve textbook exercises on data flow analysis

Deliverable: Study notes with at least 5 worked examples demonstrating data flow analysis concepts

Week 4: Data Flow Analysis - Part 2

Topics to Study:

- Liveness analysis (live variable analysis)
- Available expressions analysis
- Very busy expressions
- Constant propagation
- Worklist algorithms for efficiency

Learning Activities:

- Implement basic Control Flow Graph construction algorithm in Python/Java/C++
- Code the liveness analysis algorithm
- Test the implementation on simple code examples with known outputs
- Debug and validate the correctness of your implementation
- Document the algorithms with comments and explanations

Deliverable: Working implementation of CFG construction and liveness analysis with test cases

Week 5: Dead Code Detection Algorithms

Topics to Study:

- Mark-and-sweep algorithm for dead code detection
- Unreachable code detection using CFG traversal (depth-first search, breadth-first search)
- Dead variable elimination using liveness information
- Dead store elimination techniques
- Conservative vs. aggressive optimization strategies

Learning Activities:

- Implement unreachable basic block detection algorithm
- Develop algorithm to identify unused variables using liveness analysis results
- Create comprehensive test suite with various dead code patterns
- Test on edge cases such as loops, conditionals, and nested structures
- Validate detection accuracy manually

Deliverable: Dead Code Detection Module with documented algorithms and test results

Week 6: Tool Development and Integration

Topics to Study:

- Lexical analysis and tokenization
- Parsing techniques (top-down, bottom-up)
- Abstract Syntax Tree (AST) construction and manipulation
- Parser generators: ANTLR, Yacc, Lex, PLY
- Command-line interface design

Learning Activities:

- Choose implementation language (Python recommended for rapid development, Java/C++ for performance)
- Select parser library or build custom lexer/parser
- Integrate CFG construction module with dead code detection module
- Develop command-line interface for tool usage
- Implement input/output file handling
- Create basic error handling and reporting mechanisms

Deliverable: Working prototype (alpha version) with end-to-end functionality

Week 7: Testing and Validation

Learning Activities:

- Create comprehensive test suite covering:
    - Simple programs with obvious dead code
    - Complex programs with nested structures
    - Edge cases and boundary conditions
    - Programs with no dead code (negative tests)
- Test on benchmark programs from standard test suites
- Measure and document performance improvements:
    - Code size reduction (lines of code, executable size)
    - Execution time improvement
    - Memory usage reduction

    - Energy consumption reduction (if measurement tools available)
- Debug identified issues and refine algorithms
- Compare results with existing compiler optimization tools
- Perform accuracy validation against manual analysis

Deliverable: Comprehensive Test Report with performance metrics, graphs, and comparison analysis

Week 8: SDG 13 Integration and Documentation

Learning Activities:

- Calculate energy savings achieved through code optimization
- Research carbon footprint of software execution in data centers and personal devices
- Document environmental impact using metrics such as:
    - Power consumption reduction (Watts)
    - Energy saved per execution (Joules)
    - Estimated carbon emission reduction (grams CO_2)
    - Projected impact at scale
- Prepare comprehensive project report following academic standards
- Create professional presentation slides
- Develop user manual with installation instructions and usage examples
- Write technical documentation explaining system architecture and algorithms

Deliverable: Final Project Report (20-30 pages), Presentation Slides, User Manual, and Technical Documentation

Week 9-10: Final Refinement and Presentation

Learning Activities:

- Code cleanup: remove debug code, improve naming conventions, add comments
- Code optimization: improve algorithm efficiency, reduce memory usage
- Final round of testing and bug fixes
- Prepare demonstration scenarios showcasing tool capabilities
- Create demo video (optional but recommended)
- Rehearse presentation multiple times
- Prepare for Q&A session by anticipating potential questions
- Submit all final deliverables according to college guidelines

Deliverable: Final presentation, polished source code, complete documentation, and all supporting materials

3. KEY CONCEPTS TO MASTER

Concept	Description	Priority
Control Flow Graph (CFG)	Graph representation of all possible execution paths through a program, with nodes representing basic blocks and edges representing control flow	High
Basic Blocks	Maximal sequences of consecutive instructions with single entry point and single exit point, no branches except at the end	High
Liveness Analysis	Data flow analysis technique for determining which variables are live (may be used in the future) at each program point	High
Reaching Definitions	Analysis to determine which variable definitions may reach a given program point without being overwritten	High
Unreachable Code	Code segments that can never be executed under any input conditions due to control flow constraints	Medium
Dead Store	Assignment to a variable whose value is never subsequently read or used before being overwritten	Medium
Constant Propagation	Optimization technique that replaces variables with their known constant values to enable further optimizations	Low
Use-Definition Chain	Data structure linking each variable use to all possible definitions that may reach it	Medium

4. RECOMMENDED STUDY RESOURCES

4.1 Primary Textbooks

1. **Compilers: Principles, Techniques, and Tools (Dragon Book)**
   Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
   Focus: Chapters 8 (Code Generation) and 9 (Machine-Independent Optimizations)
   Key Topics: Data flow analysis, optimization techniques, dead code elimination
2. **Engineering a Compiler**
   Authors: Keith Cooper and Linda Torczon
   Focus: Chapters on Data Flow Analysis and Optimization
   Key Topics: Iterative data flow algorithms, SSA form, advanced optimization
3. **Modern Compiler Implementation in Java/C/ML**
   Author: Andrew W. Appel
   Focus: Implementation-oriented approach with practical examples

4.2 Online Resources

- **Stanford CS143 Compilers:** Lecture notes and assignments available online
- **MIT OpenCourseWare:** Computer Language Engineering course materials
- **LLVM Documentation:** https://llvm.org/docs/ - Study optimization passes
- **GCC Internals:** https://gcc.gnu.org/onlinedocs/gccint/ - Understanding real-world compiler optimization
- **Research Papers:** ACM Digital Library, IEEE Xplore - Search for "dead code elimination" papers

4.3 Tools and Software

- **ANTLR (ANother Tool for Language Recognition):** Parser generator for building lexers and parsers
- **PLY (Python Lex-Yacc):** Python implementation of lex and yacc parsing tools
- **LLVM:** Modular compiler infrastructure for reference and comparison
- **Python/Java/C++:** Choose based on comfort level and performance requirements
- **Git/GitHub:** Version control and code repository management
- **Visual Studio Code / PyCharm / Eclipse:** Integrated development environments

4.4 Video Lectures

- Stanford CS143 lectures on YouTube
- MIT OCW lectures on compiler design
- NPTEL lectures on Compiler Design by IIT professors

5. IMPLEMENTATION MILESTONES

Milestone	Target Week	Success Criteria
Literature Review Complete	Week 2	Comprehensive document covering DCE theory, existing techniques, and proposed approach with at least 10 references
CFG Construction Working	Week 4	Tool successfully generates accurate Control Flow Graphs from sample programs with visualization capability
Liveness Analysis Implemented	Week 4	Algorithm correctly computes live variable information, validated against manual analysis

Dead Code Detection Functional	Week 5	Tool accurately identifies unreachable code, unused variables, and dead stores in test programs
Alpha Version Complete	Week 6	End-to-end working prototype with command-line interface, capable of analyzing simple programs
Testing Complete	Week 7	Comprehensive test suite executed, performance metrics collected, comparison with existing tools documented
Documentation Complete	Week 8	All reports, manuals, and presentations finalized and reviewed
Final Submission	Week 10	All deliverables submitted, demonstration prepared, ready for evaluation

6. SDG 13 ALIGNMENT: CLIMATE ACTION

6.1 Connection to Climate Action

Dead code elimination directly contributes to Sustainable Development Goal 13 (Climate Action) through multiple pathways:

- **Energy Efficiency:** Optimized code requires fewer CPU cycles to execute, reducing energy consumption in data centers, servers, and end-user devices. Even small improvements, when scaled globally, result in significant energy savings.
- **Reduced Carbon Footprint:** Lower energy consumption translates to reduced carbon emissions, especially in regions where electricity is generated from fossil fuels. Smaller executables also require less storage and network bandwidth, further reducing environmental impact.
- **Resource Optimization:** Efficient code extends hardware lifespan by reducing computational load, decreasing heat generation, and minimizing wear on components. This reduces electronic waste and the environmental cost of manufacturing new hardware.
- **Scalability Impact:** When applied to large-scale systems serving millions of users (cloud services, mobile applications, web platforms), even minor optimizations multiply into substantial energy and cost savings.

6.2 Quantifying Environmental Impact

Metrics to Calculate:

- Power consumption reduction (Watts) before and after optimization
- Energy saved per program execution (Joules or kWh)

- Estimated carbon emission reduction using regional electricity carbon intensity (grams CO_2 per kWh)
- Projected annual energy savings if deployed at scale
- Equivalent environmental benefit (e.g., trees planted, kilometers not driven)

6.3 Project Activities for SDG 13 Integration

1. **Energy Measurement:** Use tools like Intel Power Gadget, PowerTOP, or specialized hardware to measure actual power consumption before and after code optimization
2. **Carbon Calculation:** Research regional electricity carbon intensity and calculate CO_2 emissions reduction based on energy savings
3. **Green Computing Research:** Study principles of sustainable software engineering and incorporate relevant concepts into project documentation
4. **Impact Documentation:** Create detailed section in final report demonstrating environmental benefits with charts, graphs, and quantitative analysis
5. **Scaling Analysis:** Extrapolate individual program improvements to hypothetical large-scale deployment scenarios

7. FINAL DELIVERABLES CHECKLIST

Ensure all the following items are completed before final submission:

Source Code Package:

- ☐ Complete, well-commented source code
- ☐ README file with project overview and setup instructions
- ☐ Requirements.txt or equivalent dependency list
- ☐ Sample input files for testing
- ☐ Build/installation scripts if applicable

Documentation:

- ☐ Literature Review Report (5-7 pages)
- ☐ Technical Design Document explaining system architecture and algorithms
- ☐ User Manual with installation and usage instructions
- ☐ Developer Documentation for future maintenance and enhancement

Testing Materials:

- ☐ Comprehensive test suite with diverse test cases
- ☐ Test execution results and logs
- ☐ Performance analysis report with metrics and graphs
- ☐ Comparison analysis with existing tools

SDG 13 Documentation:

- ☐ Environmental impact assessment with calculations
- ☐ Energy consumption measurements and analysis
- ☐ Carbon footprint reduction estimates
- ☐ Green computing principles applied

Final Report:

- ☐ Complete project report (20-30 pages) covering all aspects
- ☐ Proper formatting with title page, table of contents, page numbers
- ☐ All sections: Introduction, Literature Review, Methodology, Implementation, Results, SDG Analysis, Conclusion
- ☐ References in standard citation format (IEEE/APA)
- ☐ Appendices with code samples, additional figures, raw data

Presentation Materials:

- ☐ Professional presentation slides (15-20 slides)
- ☐ Demonstration plan with example scenarios
- ☐ Demo video (optional but recommended, 3-5 minutes)
- ☐ Backup materials in case of technical difficulties

8. EVALUATION CRITERIA

Understanding how your project will be evaluated helps you focus efforts appropriately. The typical evaluation breakdown is:

Criteria	Weight	Key Assessment Points
Technical Implementation	35%	Algorithm correctness, code quality, functionality completeness, efficiency, proper use of data structures, error handling
Documentation	20%	Completeness, clarity, technical depth, proper formatting, quality of explanations, code comments
Testing and Validation	20%	Test coverage, variety of test cases, performance measurements, accuracy validation, comparison analysis
SDG 13 Integration	10%	Quality of environmental impact analysis, meaningful calculations, understanding of green computing principles
Presentation	10%	Clarity of explanation, demonstration effectiveness, ability to answer questions, presentation professionalism

Innovation and Extra Features	5%	Novel approaches, additional features beyond requirements, creative solutions, tool usability enhancements

9. RISK MANAGEMENT

Anticipating potential challenges and having mitigation strategies ensures project success:

Risk	Impact	Mitigation Strategy
Complex parser development consuming excessive time	High	Use existing parser libraries (ANTLR, PLY) instead of building from scratch. Focus on core optimization algorithms rather than frontend complexity.
Insufficient understanding of data flow analysis	High	Allocate extra time during Weeks 3-4 for studying. Work through numerous examples. Seek help from faculty advisor early. Use online tutorials and videos.
Limited availability of good test cases	Medium	Start collecting test programs from Week 1. Use standard benchmark suites. Write own test cases with known expected outputs. Ask peers to share test programs.
Time constraints due to other coursework	Medium	Follow weekly schedule strictly. Start early. Focus on essential features first. Keep advanced features as optional enhancements. Use time management techniques.
Tool integration and compatibility issues	Medium	Test integration early (Week 5-6). Have backup plan for standalone tool if integration fails. Keep modules loosely coupled for easier debugging.
Difficulty measuring energy consumption	Low	Use available software tools (Intel Power Gadget, etc.). If direct measurement not possible, use theoretical analysis and published data for estimates.
Bugs in implementation affecting results	Medium	Test incrementally as you develop. Use version control to track changes. Implement logging for debugging. Have peer review your code.

10. DAILY STUDY SCHEDULE

10.1 Recommended Daily Time Allocation

Consistent daily effort is more effective than sporadic intensive sessions. Aim for **2-3 hours daily**:

- **Theory Study (45-60 minutes):**
  - Textbook reading and note-taking
  - Understanding concepts and algorithms
  - Watching video lectures if needed
  - Reviewing previous day's work
- **Practical Implementation (60-90 minutes):**
  - Writing code
  - Testing and debugging
  - Running experiments
  - Integrating modules
- **Documentation (30 minutes):**
  - Writing daily progress notes
  - Updating project report sections
  - Commenting code
  - Preparing presentation materials

11 .FUTURE SCOPE

- Integration of machine learning to predict redundant code automatically.
- Applying DCE techniques in large-scale cloud environments to reduce energy usage.
- Development of tools that track real-time compiler optimization impact on energy metrics.

12. CONCLUSION

The **Dead Code Elimination Tool** plays an essential role in optimizing compiler performance and reducing software inefficiency. Studying this process enhances understanding of compiler design while highlighting the connection between software optimization and environmental sustainability under **SDG 13 – Climate Action**.