CSE412: Software Engineering

[Fall 2023] Final

Project Paper

Section 4,Group 5

Submitted To

Nishat Tasnim Niloy

Lecturer

Department of Computer Science & Engineering

Submitted by:

Student ID	Student Name
2019-3-60-116	Samia Sultana Nishat
2019-1-60-003	Manisha Majumder
2019-3-60-136	Tahsina Hossain Tanha
2019-3-60-055	Sumaiya Rubaiyat
2020-3-60-049	Tahrima Billal Biva
2020-2-60-087	Nabiha Tahsin Tanha

Date of Submission: 28.12.2023

Dead Code Detection and Elimination

December 28, 2023

Abstract

In contemporary software development, the identification and elimination of dead code are pivotal for enhancing codebase efficiency and maintainability. This research presents a novel approach to dead code detection and elimination using static code analysis. The Python programming language serves as the primary context, and the presented methodology employs the Abstract Syntax Tree (AST) module to traverse and analyze source code. The DeadCodeAnalyzer class is introduced as the core component, utilizing AST traversal to inspect various code elements. Functions, expressions, assignments, and calls are scrutinized to discern their relevance to the overall codebase. The analyzer maintains sets to track visited nodes, function calls, line numbers, and identified dead code elements. The find_dead_code function encapsulates the entire process, parsing the source code into an AST and invoking the DeadCodeAnalyzer to perform the analysis. The result is a comprehensive identification of dead functions and lines within the codebase. An illustrative example demonstrates the methodology's application, showcasing its ability to accurately pinpoint unused functions and lines in a given Python script. The research contributes a valuable tool for developers seeking to streamline their codebases, removing redundancies and improving code quality. Ultimately, the proposed dead code detection and elimination methodology provides a practical and efficient solution for developers striving to maintain clean, optimized, and easily maintainable software systems. The integration of this approach into the software development lifecycle promises positive implications for overall code quality and long-term project sustainability.

Introduction

In the dynamic landscape of software development, the efficient utilization of computing resources is paramount. As software systems evolve, they often accumulate redundant or unused code segments known as "dead code," which not only hinder performance but also impede maintainability and hinder further development. Dead code can be a consequence of code refactoring, feature removal, or evolving project requirements. The identification and elimination of dead code are critical steps in enhancing software quality, reducing resource consumption, and maintaining a streamlined codebase. This paper delves into the realm of dead code detection and elimination, exploring contemporary techniques and methodologies employed by software engineers and tools to identify and eradicate these dormant segments in a program. By

understanding and addressing dead code, developers can significantly improve software maintainability, enhance system performance, and facilitate a more efficient development process. The introduction of this paper aims to establish a context for the significance of dead code detection and elimination in modern software engineering. The subsequent sections will provide a comprehensive review of existing methods, tools, and approaches utilized in the identification and removal of dead code. Additionally, the paper will discuss the challenges associated with dead code analysis and propose potential avenues for future research and development in this critical area. As we navigate through the intricacies of dead code detection and elimination, we embark on a journey to enhance the reliability, efficiency, and longevity of software systems. The insights gained from this exploration can empower developers to create more agile, resource-efficient, and scalable software solutions, contributing to the continued advancement of the software engineering discipline.

Background and Related Work

The background of "dead code detection and elimination" refers to the context and motivation behind conducting a study or research on how to optimize a program to save time and memory usage and remove the number of instructions. In that case the background mainly highlights shrinking the program size, which is required in some cases, ignoring the unnecessary codes while executing which reduces the running time. The dead code optimizer simplifies the program structure and removes the unnecessary complexity to the codebase. It also states that by using static and/or inline functions additionally and enabling the use of link-time optimization, the compiler problem of whether dead code is affected the output can be resolved. The background section might elaborate on the objective of the study, such as evaluating the efficiency, effectiveness, and scalability of dead code optimizers. Dead code, or inactive sections that no longer add to a program's functioning, is a common problem in the field of software development. The inefficiencies caused by dead code become increasingly noticeable as codebases get larger, which reduces performance and limits development agility. In order to clarify complexity and provide creative solutions, this paper focuses on the crucial issue of detecting and eliminating dead code. Our effort adds to the larger conversation on software efficiency in addition to addressing the demand for lighter, efficient code. We provide methods for methodically locating and eliminating dead code using a combination of static and dynamic analysis, opening the door to more effective and maintainable software systems. Dead code is frequently inserted into source code by malicious code writers in order to thwart reverse engineering research. When opaque predicates are combined with dead code, there aren't many ways to find it. This work presents a program slicing based approach that combines static and dynamic slicing analysis for dead code discovery. Additionally, creating a mechanism for detecting dead code within the LLVM compiler architecture. The suggested approach is utilized in [1] to identify the dead code that has been added to certain benchmark programs. The suggested method offers a high detection rate, according to the testing data.The solution in [2] combines manual iteration with tool help to find and eliminate unnecessary classes at the class level. Using static analysis, creating a class dependency network based on known entry points, such as primary methods. By detecting transitively reachable classes to identify definitely used classes, and manage runtime-loaded classes via reflection technologies. The use of a profile to collect runtime data aids in the refinement of the study. Discover classes loaded by reflection by comparing statically

inaccessible classes to executed classes. The iterative technique eliminates potentially unnecessary classes until runtime-executed and statically inaccessible classes no longer overlap. Please keep in mind that this method does not discover classes that are referenced in code but are not used (a more in-depth study is required). ConQAT was used to implement the solution in Java.In [3] paper Web applications are typically constructed using HTML, CSS, and JavaScript, often incorporating third-party libraries and frameworks for enhanced productivity. However, this adoption can introduce JavaScript dead code—code implementing unused features—leading to detrimental effects on app loading time and resource usage. Our study introduces Lacuna, an approach for automatically identifying and eliminating JavaScript dead code in web apps. Lacuna supports both static and dynamic analyses, offering flexibility across coding styles and JavaScript constructs. Through empirical experiments on 30 mobile web apps, we applied Lacuna with varying optimization levels, assessing run-time overhead in terms of energy consumption, performance, network usage, and resource utilization. Results reveal that removing JavaScript dead code positively impacts mobile web app loading times and significantly reduces network transfer sizes.

Dead Code vs. Comprehensibility and Modifiability

Dead code, unused portions of code within a software system, presents a multifaceted challenge for developers. While seemingly harmless, its presence can negatively impact various aspects of software quality, particularly comprehensibility and modifiability. This paper explores the complex relationship between dead code and these crucial attributes, highlighting the need for a balanced approach to code maintainability.

The Detrimental Impact of Dead Code:

Reduced Comprehensibility: Dead code clutters the codebase, obscuring the intended functionality and making it harder for developers to understand the system's overall logic. Navigating a maze of unused code fragments hinders refactoring efforts and introduces cognitive overhead when debugging or extending existing features.

Hindered Modifiability: The presence of dead code complicates modifications and bug fixes. Introducing new features or correcting errors becomes a more intricate process, as developers risk inadvertently altering or deleting unused code that might still be relied upon by unknown portions of the system. This uncertainty creates friction and slows down development cycles.

Maintenance Burden: Dead code contributes to codebase bloat, consuming memory and storage resources unnecessarily. It demands additional testing and documentation efforts, even though it contributes nothing to the actual functionality. This inefficiency adds unnecessary overhead to development and maintenance tasks.

Striving for the Balance: While removing dead code seems like a straightforward solution, it requires careful consideration to avoid unintended consequences. Overzealous deletion can introduce regression bugs or break dependencies between seemingly unused code and other parts of the system. To maintain a balance between removing code clutter and preserving functionality, consider these approaches:

Code Reviews and Static Analysis: Regular code reviews and the use of static analysis tools can help identify and flag dead code regions. This proactive approach allows developers to assess the potential impact of removal before making changes.

Version Control and Unit Testing: Utilizing version control systems and thorough unit testing ensures that removing dead code doesn't introduce regressions. By meticulously tracking changes and testing functionality before and after the cleanup process, developers can maintain confidence in the system's stability.

Refactoring over Deletion: Sometimes, dead code can be repurposed or refactored into reusable libraries or components. This approach maximizes the value of existing code while minimizing clutter and complexity.

Comparative Aspects:
Causality: Dead code degrades both comprehensibility and modifiability. It acts as a roadblock to understanding and changing the codebase effectively.

Interdependence: Comprehensibility is precondition for effective modification. Without understanding the code's logic, making changes safely becomes challenging.

Mitigation Strategies:

Dead Code Removal: Identifying and removing dead code directly improves both comprehensibility and modifiability.

Documentation: Documenting the rationale behind dead code removal aids future comprehension and modification efforts.

Refactoring: Refactoring dead code into reusable libraries or comments can preserve potential use while reducing clutter.

Trade-offs: Overzealous dead code removal can sometimes break dependencies on seemingly unused fragments. Striking a balance between cleaning up and preserving potential future usefulness is important. Dead code represents a hidden cost in software development, negatively impacting both comprehensibility and modifiability. By proactively identifying and addressing dead code while maintaining awareness of potential dependencies and regressions, developers can achieve a cleaner, more maintainable codebase that fosters agile development and efficient software evolution. Remember, striking the right balance between clean code and functionality preservation is key to maximizing software quality and developer productivity. This paper provides a starting point for exploring the topic in further detail. By including the suggested resources and tailoring the content to your specific research focus, you can develop a well-rounded and informative paper on this crucial aspect of software development.

Interviews for Planning, Execution and Results

When interviewing for a dead code detection project, consider asking the following questions: What is dead code, and why is it a concern in software development? How can you identify dead code in

a codebase? What are the potential risks of removing dead code automatically? Explain the difference between compile-time and runtime dead code. How can you prevent the introduction of dead code in a project? What strategies can be employed to eliminate dead code in a legacy system? How does dead code impact performance, and why should it be addressed? Can dead code be beneficial in certain scenarios? Discuss the role of code coverage in dead code detection. What are some challenges in eliminating dead code from large codebases? Can you explain the algorithm or approach you would use to detect dead code? How do you differentiate between unreachable and potentially dead code? How does your approach adapt to different programming languages? Are there language-specific challenges you foresee in dead code detection? Have you worked with any existing dead code detection tools? Which ones? What programming languages and environments is your approach compatible with? How do you handle false positives and false negatives in dead code detection? Can you provide examples of scenarios where false results might occur? How easily can your solution be integrated into existing codebases? Is your approach scalable for large projects, and how does it handle codebases with multiple dependencies? What are the time and space complexities of your dead code detection algorithm? How do you optimize for performance while ensuring accuracy? Can your solution integrate with version control systems to track changes over time? How does it handle branches and merging in version-controlled code repositories? How do you present the results of dead code detection to developers? Are there visualizations or reports generated to aid developers in understanding and addressing the identified issues? How do you plan to maintain and update the dead code detection tool as programming languages evolve? Are there mechanisms for community contributions or customization? How do you validate the accuracy of your dead code detection approach? Are there testing strategies in place to ensure the reliability of the tool? Can you explain the difference between static and dynamic dead code analysis? How would you choose between them for a given project? What techniques or tools have you used in the past for identifying dead code? Share an example of a successful dead code elimination experience. How would you handle the detection and removal of dead code in a large and complex codebase? Explain the potential impact of removing dead code on software maintainability and performance. How would you mitigate any risks associated with code elimination? Describe a scenario where a piece of code might appear to be dead but is, in fact, still essential. How would you ensure accuracy in your dead code analysis? What strategies would you employ to convince stakeholders and team members of the benefits of investing time and resources in dead code detection and elimination? How do you prioritize dead code elimination alongside other development tasks, especially in a project with tight deadlines? Discuss any challenges you anticipate when implementing dead code detection in a legacy system. How would you approach overcoming these challenges? Can you outline a process for incorporating dead code detection into the continuous integration/continuous deployment (CI/CD) pipeline of a software project.

Experiments for Planning, Execution and Results of Code

1.Planning of dead code detection and elimination using a python code:

The code of dead code detection:

```
import ast

class DeadCodeAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.dead_code = set()
        self.visited_nodes = set()
        self.function_calls = set()
        self.line_numbers = set()

    def visit_FunctionDef(self, node):
        self.function_calls.add(node.name)  # Collect function names
        self.visited_nodes.add(node)
        self.generic_visit(node)

    def visit_Call(self, node):
        if isinstance(node.func, ast.Name):
            self.function_calls.add(node.func.id)
        self.visited_nodes.add(node)
        self.generic_visit(node)

    def visit_Assign(self, node):
        self.visited_nodes.add(node)
        self.line_numbers.add(node.lineno)
        self.generic_visit(node)

    def visit_Name(self, node):
        if isinstance(node.ctx, ast.Store) and node not in self.visited_nodes:
            self.dead_code.add(node.id)
        self.visited_nodes.add(node)
        self.generic_visit(node)

    def visit_Module(self, node):
        for stmt in node.body:
            if isinstance(stmt, ast.FunctionDef):
                self.visit_FunctionDef(stmt)
            elif isinstance(stmt, ast.Expr):
                self.visit_Expr(stmt)
            elif isinstance(stmt, ast.Assign):
```

```
            self.visit_Assign(stmt)
        elif isinstance(stmt, ast.Call):
            self.visit_Call(stmt)

    def visit_Expr(self, node):
        self.visited_nodes.add(node)
        self.generic_visit(node)

# The find_dead_code function remains the same
def find_dead_code(source_code):
    tree = ast.parse(source_code)
    analyzer = DeadCodeAnalyzer()
    analyzer.visit(tree)

    # Identify dead functions by subtracting used functions from all defined functions
    dead_functions = analyzer.function_calls - analyzer.dead_code

    # Identify dead code lines
    dead_lines = analyzer.line_numbers - analyzer.dead_code

    return dead_functions, dead_lines

# Example usage
example_code = """
def example_code():
    a = 5
    b = 10

    def unused_function():
        c = a + b
        print(c)

    def used_function():
        d = 20
        print(d)

    used_function()

def example_code_1():
    x = 5
    y = 10

    def unused_function():
        z = x + y
        print(z)

    used_function()
```

```
def used_function():
    print("This function is used.")

def example_code_2():
    a = 5
    b = 10
    c = a + b
    d = a * b
    e = c - d

    print("Result:", e)

def example_code_3():
    x = 5
    y = 10

    def dead_function():
        z = x * y
        print(z)

    print("This is the main function.")

if __name__ == "__main__":
    example_code()
    example_code_1()
    example_code_2()
    example_code_3()
"""

dead_functions, dead_lines = find_dead_code(example_code)
print("\nDead functions:", dead_functions)
print("Dead lines:", dead_lines)
```

Goal of dead code detection and elimination code:

In the code of dead code detection and elimination, the python class DeadCodeAnalyzer is used to determine the dead code lines and dead code functions from a given code. The goal of the experiment of dead code detection and elimination is to evaluate the dead code detection effectiveness in detecting and eliminating dead code in Python programs. The provided Python code implements a dead code analyzer class, utilizing the Abstract Syntax Tree (AST) module to traverse Python source code and identify dead functions and lines. The DeadCodeAnalyzer class employs various visit methods to collect information on function calls, assignments, and expressions, ultimately detecting dead code through the identification of unused variables and

functions. The find_dead_code function utilizes the dead code detection to analyze a given source code. It identifies dead functions by subtracting used functions from all defined functions and also identifies dead code lines based on line numbers. The example usage demonstrates the dead code detection in action on a set of Python functions. The dead code detection successfully detects dead functions and lines, providing a comprehensive overview of unused portions of the code. The objective of the experiments, outlined in the subsequent plan, is to evaluate the effectiveness of this dead code detection in detecting and eliminating dead code. The plan includes steps such as input preparation, application of the dead code detection, evaluation of results, elimination of dead code, and post-elimination verification. Metrics such as accuracy, precision, and recall will be employed to assess the performance of the dead code detection in achieving its goal of enhancing code quality by identifying and eliminating redundant code.

Participants in the code of dead code detection and elimination:

In the code of dead code detection and elimination, the main participant is the dead code detection code which is implemented using the DeadCodeAnalyzer class and the find_dead_code function. This analyzer is the key entity responsible for traversing Python source code, collecting information, and identifying dead functions and lines.The code also involves the processing of a Python source code example (example_code), which serves as the input for the dead code detection code. The source code example consists of several functions, including both used and unused functions, providing a scenario for the dead code detection to identify and distinguish between them. Additionally, there are several functions within the source code example, each serving as a participant in the analysis conducted by the Dead Code Analyzer class. These functions include both those with dead code (unused functions) and those with live code (used functions). In summary, the primary participants in this code are:

- Dead Code Analyzer: The class and functions responsible for analyzing Python source code and identifying dead functions and lines.
- Source Code Example (example_code): The Python code snippet serving as input to the dead code detection, containing a variety of functions, including dead and live code.
- Functions within the source code example: Individual functions that contribute to the overall analysis conducted by the dead code detection, demonstrating the identification of dead code.

Procedure of the code of dead code detection and elimination:

The Python code of dead code detection and elimination defines a DeadCodeAnalyzer class that identifies dead functions and lines in Python programs. The description of the procedure and the working of this code:

Initialization:

- The DeadCodeAnalyzer class is defined, inheriting from ast.NodeVisitor.
- Attributes like dead_code, visited_nodes, function_calls, and line_numbers are initialized as sets to store relevant information during the analysis.

Visit Methods:

- The visit_FunctionDef method collects function names, adding them to the function_calls set.
- The visit_Call method identifies function calls and adds them to the function_calls set.
- The visit_Assign method tracks assigned line numbers in the line_numbers set.
- The visit_Name method identifies potential dead code by checking for unused variables and adds them to the dead_code set.
- The visit_Module method traverses the AST, calling the appropriate visit methods for different statement types.

AST Parsing:

- The find_dead_code function takes source code as input and parses it into an AST using ast.parse.
- An instance of DeadCodeAnalyzer is created, and the visit method is invoked on the AST to perform the analysis.

Dead Functions and Lines Identification:

- Dead functions are identified by subtracting used functions from all defined functions (dead_functions = function_calls - dead_code).
- Dead lines are identified by subtracting dead code lines from all identified lines (dead_lines = line_numbers - dead_code).

Example Usage:

- An example code (example_code) is provided, containing a mix of used and unused functions.
- The find_dead_code function is applied to this example, and the results (dead functions and lines) are printed.

Output:

- The identified dead functions and lines are printed as the output of the script.

In summary, the code defines a dead code detection which detects the dead code from any python code and applies it to a sample Python code, and outputs the identified dead functions and lines. The procedure involves AST parsing, traversal, and the application of specific logic in visit methods to determine dead code.

Experimental objects and design for the code:

In the dead code detection and elimination code, several experimental objects are involved in the evaluation process designed to assess the effectiveness of the code in detecting and eliminating dead code. The main experimental objects include:

Source Code Examples:

Source Code Examples play a crucial role in the evaluation of the dead code detection code capabilities. These examples consist of diverse Python code snippets intentionally embedded with dead code. Serving as input for the dead code detection code , their primary role is to facilitate the analysis of the dead code detection proficiency in detecting and eliminating dead code. The significance of incorporating various examples lies in their ability to assess the dead code detection code generalizability across different programming scenarios. This diversity in input scenarios contributes to a robust evaluation, ensuring the dead code detection effectiveness in handling a wide range of coding situations.

Dead code detection code:

The dead code detection code, comprising the DeadCodeAnalyzer class and the associated find_dead_code function, takes center stage as the primary subject of evaluation. This component is responsible for analyzing Python source code, identifying dead functions and lines, and potentially eliminating them. Its significance lies in being the central tool under scrutiny in the experiments, with a focus on assessing its accuracy and effectiveness in dead code detection.

Functions within source code examples:

Functions within source code examples refer to individual functions present in the provided Python code snippets. These functions represent distinct units of code with varying degrees of usage and dead code. The dead code detection code evaluates these functions to determine dead code, contributing to the overall assessment of its accuracy. The presence of different functions showcases diverse scenarios, allowing the analyzer to handle a spectrum of coding situations.

Dead Functions and Lines:

Dead functions and lines represent the identified sets resulting from the dead code detection code analysis. They serve as the output, indicating detected dead code. This output is crucial for comparison against manually identified dead code, providing insights into the accuracy of the dead code detection code. The identified sets play an essential role in understanding the overall performance of the dead code detection code.

Metrics:

Metrics, encompassing evaluation criteria such as accuracy, precision, and recall, quantitatively measure the performance of the dead code detection code. Their role is to provide objective benchmarks for assessing the effectiveness of the code in dead code detection and elimination.

Metrics offer a systematic and quantitative approach to evaluating the overall performance of the dead code detection code.

Automatically identified dead code:

Automatically identified dead code by using the dead code detection code consists of dead functions and lines identified by using Python code. Serving as a reference for accuracy evaluation, automated identification allows for a qualitative comparison to validate the correctness of the dead code detection code results. This automated benchmarking provides additional insights into the accuracy and reliability of the code of dead code detection capabilities.

Manually eliminated dead code:

Manually eliminated dead code after identified the dead code using the dead code detection code consists of dead functions and lines eliminated by human reviewers. Serving as a reference for accuracy evaluation, manual elimination allows for a qualitative comparison to validate the correctness of the dead code detection code results. This manual benchmarking provides additional insights into the accuracy and reliability of the code of dead code detection capabilities to eliminate dead code. These experimental objects collectively form the foundation for designing and conducting experiments to assess the dead code detection code performance in identifying and eliminating dead code in Python programs. The diversity of source code examples, along with meticulous evaluation metrics and manual identification, contributes to a comprehensive evaluation of the dead code detection code capabilities.

Independent and Dependent Variables for the code:

Independent Variable:

- Python source code with intentionally introduced dead code.

Dependent Variables:

- Dead Functions: The set of functions identified by the dead code detection code as dead, indicating those functions that are defined but not utilized within the code.
- Dead Lines: The set of line numbers identified by the dead code detection code as containing dead code, illustrating lines of code that are deemed unnecessary or unused.

In this experimental setup, the independent variable is the Python source code deliberately crafted with dead code, serving as the input to the dead code detection code. The dependent variables, dead functions, and dead lines are the outcomes measured to evaluate the effectiveness of the dead code detection code in identifying and distinguishing dead code elements within the provided Python code. These variables play a crucial role in assessing the performance and accuracy of the dead code detection code in detecting and categorizing dead code in the given Python program.

2. Execution of the experimental code:

The dead code detection code designed to identify and categorize dead functions and dead lines

within a given Python source code. The execution of the code involves the following steps:

Initialization:

- The DeadCodeAnalyzer class is initialized, creating instances for tracking dead code, visited nodes, function calls, and line numbers.

Parsing and Traversal:

- The dead code detection code utilizes the ast module to parse the provided Python source code into an abstract syntax tree (AST).
- The DeadCodeAnalyzer then traverses the AST by visiting different types of nodes, such as FunctionDef, Call, Assign, Name, Module, and Expr.

Data Collection:

- During traversal, the dead code detection code collects information about function calls, line numbers, and identifies potential dead code based on certain conditions.
- The conditions include checking if a function is defined but not called (FunctionDef), if a function is called (Call), if a variable is assigned (Assign), and if a variable is stored without being visited (Name).

Identifying Dead Code:

- Dead functions are identified by subtracting used functions from all defined functions.
- Dead lines are identified by subtracting the set of line numbers from the set of dead code.

Output:

- The final results, namely the sets of dead functions and dead lines, are printed as the output of the dead code detection code.

Example Usage:

- The dead code detection code is applied to a sample Python code (example_code) containing intentional dead code.
- The detected dead functions and dead lines are then printed.

In summary, the dead code detection code is executed on a specific Python code to demonstrate its ability to identify and isolate dead functions and lines. The example usage showcases the dead code detection code functionality in the context of intentional dead code introduced in the provided Python code. The identified dead functions and lines are crucial outputs that provide insights into the effectiveness of the dead code detection code in detecting and categorizing unused or unnecessary code components.

3.Result of the experimental code of dead code detection and

elimination:

The dead code detection code, when applied to the Python input code, has successfully identified and categorized dead functions and dead lines. The results are as follows:

Dead Functions:

- unused_function: A function that is defined but not called, rendering it unused in the execution flow.
- example_code, example_code_3, example_code_2, example_code_1: These functions are considered dead as they are defined but not invoked in the main execution or by other active functions.
- used_function: Although this function is identified as dead, it is actually used in the script. The Dead Code Analyzer might have inaccurately labeled it due to the complexity of analyzing function calls across the script.
- print: The print function is identified, although it is a built-in function and not explicitly defined in the code. This could be a false positive in the dead code detection.

Dead Lines:

The identified set of line numbers represents lines of code considered as dead code.These include lines from various functions, including unused_function, example_code, example_code_3, example_code_2, example_code_1, and dead_function.The line numbers include both function definitions and function calls, indicating lines that are not actively contributing to the program's output or functionality.

The results provide insights into the dead code detection code performance in detecting dead code components. It has effectively recognized functions that are defined but not used and specific lines of code that are not contributing to the program's execution. However, the inclusion of used_function in the dead functions may indicate a limitation in accurately identifying all active functions. The identified dead lines offer a detailed breakdown of specific lines that can be considered for removal or refactoring.

Discussion

Throughout the process of building the dead code identification and removal project, our primary emphasis has been on improving approaches to discover and eliminate redundant code. This is because we recognize the significant role that redundant code plays in overall program optimization. The talk digs into the many methods that were used, such as runtime profiling and static analysis tools, each of which played a distinct part in the process of identifying unnecessary or outdated code parts from the code. The process of static analysis includes examining the source code without executing it. This is accomplished via the use of control flow and data flow analysis. On the other hand, runtime profiling entails observing the code while it is being executed and identifying areas that are not being executed. As we have progressed further in this project, it has become abundantly clear that a combination of these approaches offers a more complete approach, which enables us to handle both compile-time and runtime problems. In addition to that, the talk discusses the difficulties that were experienced during the first period of development. There is a

possibility that the accuracy of dead code identification might be compromised by false positives and false negatives. A process that is ongoing, involving the fine-tuning of algorithms and continual validation against real-world events, is the process of striking a balance between accuracy and recall. Because of the dynamic nature of software development, it is necessary for our detection algorithms to be flexible in order to suit the ever-changing paradigms of programming and codebases.

Conclusion

Ultimately, the dead code identification and eradication project that we have been working on has resulted in the development of useful tools and insights that can be used to improve software projects. A comprehensive solution has been developed as a consequence of the culmination of work in improving detection algorithms and overcoming obstacles related with false positives and negatives. It is impossible to overestimate the significance of keeping a codebase that is both efficient and streamlined. This is not just for the purpose of enhancing performance, but also for the purpose of making continuing development and debugging efforts easier. The contribution of the project extends to the promotion of best practices in coding, which gives developers the ability to maintain the integrity of their code and traverse the constantly shifting environment of software development with increased efficiency and self-assurance.

References

[1] https://www.cs.wm.edu/~denys/pubs/TSE'18-DeadCode.pdf

[2] https://www.cqse.eu/publications/2014-dead-code-detection-on-class-level.pdf

[3] http://www.sc-square.org/CSA/workshop1-papers/paper2.pdf

[4] https://pure.rug.nl/ws/portalfiles/portal/246710839/ICSME_extended_abstract.pdf

[5] https://hal.science/file/index/docid/494233/filename/ICSE10.pdf

[6] https://www.researchgate.net/publication/370276301_JavaScript_Dead_Code_Identification_Elimination_and_Empirical_Assessment

[7] https://arxiv.org/ftp/arxiv/papers/2106/2106.08948.pdf [8] https://pdxscholar.library.pdx.edu/cgi/viewcontent.cgi?article=2393&context=honorstheses