Anusandhanvallari
Vol 2023, No.1
January 2023
ISSN 2229-3388

Code-BERT-Powered Static Analysis: An Opti-Blend Multi-Technique Approach for Dead Code Elimination, Clone Detection, and Loop Invariant Enhancement

[1]Mr. Tulshihar Patil, [2]Prof. Dr. Shashank Joshi[2]
[1]Research Scholar, Department of Computer Engineering, Bharati Vidyapeeth (Deemed to be University), College of Engineering, Pune, Maharashtra, India.
[2]Professor, Department of Computer Engineering, Bharati Vidyapeeth (Deemed to be University), College of Engineering, Pune, Maharashtra, India.

Abstract: Opti-Blend is a unified model designed to enhance code maintainability and performance through a multi-faceted approach to optimization. The framework integrates three primary techniques: dead code elimination to reduce resource consumption, code clone detection to remove redundancy, and loop invariant optimization to improve computational efficiency. These methods are implemented using advanced technical tools such as Abstract Syntax Trees (AST), static analysis, and control flow transformations. The model's effectiveness is rigorously benchmarked on both synthetic and real-world codebases. Empirical results are presented through graphical comparisons of key metrics, including memory usage, execution time, and function count, both before and after optimization. This research paper evaluates Opti-Blend's methodology, real-world applicability, and its role in fostering the development of high-performance, scalable, and maintainable software.

Keywords: Static Analysis, Code-Bert LLM Model, Code Optimization, Dead code Elimination, Clone code Detection, Loop Invariant Code Motion, Error Detection.

1. Introduction

Large-scale software applications, often extending to millions of lines of code (MLOC), pose significant challenges in terms of maintainability, performance optimization, and scalability. One common problem with these types of systems is that they often are redundant to some extent as a result of the existence of code fragments that are either duplicated or have similar semantics (code clones). Besides that, these systems also have the potential to increase the technical debt of an organization and cause performance bottlenecks when the related portion of the system is executed [1]. Additionally, the presence of auto-generated and highly complex test scripts further complicates static and dynamic program analysis, as these scripts often introduce non-trivial control-flow structures, redundant test paths, and inflated code size. Addressing these issues requires the integration of scalable code analysis techniques, particularly those capable of handling semantic clone detection, dead code elimination, and loop invariant optimizations across massive codebases without compromising accuracy or efficiency [2, 3]

Code clones—identical or similar code fragments—remain a pervasive characteristic of software systems, comprising between 7–23% of large codebases and up to 49% in test suites, where duplication rates are typically double those in production code [4, 5]. Clones are classified into four types: Type-1 (exact), Type-2 (lexical), Type-3 (syntactic "near-miss"), and Type-4 (semantic) [3, 6, 7]. While often associated with maintenance overhead, defect propagation, and security vulnerabilities (e.g., CVE-2006-3084 affecting 42 systems), empirical findings reveal a nuanced picture: only 1.02–4% of clone genealogies introduce defects at the release level, and some cloned classes exhibit lower fault density than non-cloned code [8]. Detection techniques have evolved from text, token, tree, metric, and graph-based methods to modern deep learning (DL) approaches. Models such as UniXCoder, GraphCodeBERT, and PLBART achieve F1-scores up to 0.978, benefiting from code-aware pretraining [9].

Dead code—code segments unexecuted or without observable impact—represents a pervasive code smell in evolving software systems, often arising from iterative development and leaving behind deprecated or redundant

logic. Developer awareness remains limited; a survey by Yamashita and Moonen reported 32% of respondents were unaware of code smells, with dead code ranked 10th of 34 issues [11]. Dead Code Elimination (DCE), a core compiler optimization, removes such irrelevant code to reduce binary size, resource usage, and execution time [12].

Loop optimization is crucial in compiler design for enhancing performance in high-performance computing, where loops consume significant execution time. These optimizations, which minimize redundant computations and reduce overheads, are essential for improving cache performance through techniques like loop fission and fusion, and for enabling parallelization [13, 14]. Foundational loop optimization techniques like Loop-Invariant Code Motion (LICM) and Partial Redundancy Elimination (PRE) are now complemented by more automated approaches. Hybrid neuro-symbolic systems, such as ACInv, integrate LLMs with static analysis and formal verification methods (BMC, SMT solvers) [16, 17]. Detection techniques have evolved from text, token, tree, metric, and graph-based methods to modern deep learning (DL) approaches. Models such as UniXCoder, Graph CodeBERT, and PLBART achieve F1-scores up to 0.978, benefiting from code-aware pretraining [15,18].

Our research introduces a novel tool for code optimization that enhances code maintainability and efficiency by integrating three key techniques: invariant loop analysis, code clone detection, and dead code elimination, along with an Error Minimization module.

2. Objectives

This research is mainly aimed at the creation of a system that is accessible to the public and can, by itself, make code adjustments that take into account the software's performance, and at the same time, reduce errors. Such issues as dead code, redundant code, inefficient loop construction, etc., that cause the software system to be at a low level have been singled out in one paper to be the basic problems of the software system that result in performance degradation and technical debt. The study aims to integrate these three workflows: code clone detection, dead code analysis, and loop-invariant analysis. By implementing such a technique, a continuous process is achieved, which eases the detection of inefficiencies and enables their elimination without changing the code’s function. Furthermore, the paper outlines the use of advanced static and semantic analyses, in conjunction with LLM-based tools, to facilitate the optimization process to be more precise and dependable. These tools aid in finding complicated code clones and the slight inefficiencies that most of the traditional optimization methods may overlook. After this, the proposal is "put through its paces" and scrutinized in different codebases - both synthetic and real - to indirectly check it against various measures of evaluation such as execution time, error rate, etc. This is what the "Harvesting" of such goals means - to be able to provide a technological future of high-performance software systems that are simply more manageable and easier to maintain.

3. Scope and Methodology

Three methodology architectures are included in the proposal. The first step is to take the source code and convert it into workable representations using Abstract Syntax Trees (AST) and Static Single Assignment (SSA). The next stage features parallel analytical modules: (i) the Clone Detection Module looks for redundant structures through token hashing and a similarity threshold, (ii) the Dead Code Elimination Module examines the symbol table to delete unreachable code, and (iii) the Loop-Invariant Module gradually moves invariant expressions out of iterative constructs to lessen repeated calculations. The Static Analysis Layer, supported by PyFlakes and a machine learning-based code smell detection system, continues until no violations are found. It can also spot other hidden inefficiencies. The final stage synthesizes the optimized code, tests for semantic correctness, and benchmarks performance with metrics for execution time, total memory usage, and error count reduction. The experimental configuration guarantees that Opti-Blend may produce substantial performance gains while preserving the general scalability and stability of big software systems.

4. Literature Survey

The evolution of code optimization has progressed from single-technique models to multi-faceted, feature-driven approaches, reflecting a sustained focus on improving efficiency.

Traditional clone detection tools like CCFinder, Deckard, and NiCAD effectively used techniques such as suffix trees and Abstract Syntax Trees (ASTs) for Type 1 and 2 clones [19]. However, their scalability and accuracy declined with large datasets, a problem also faced by hybrid systems like Oreo and token-based tools like SourcererCC and CloneWorks. Neural models like ASTNN, CodeBERT, and GraphCodeBERT have enhanced source code analysis, leading to better recall and precision on benchmarks such as BigCloneBench. However, their $O(n^2)$ complexity makes them unsuitable for large-scale systems [13]. To address this, the Scalable Semantic Clone Detection (SSCD) method uses fine-tuned CodeBERT/GraphCodeBERT embeddings with approximate nearest-neighbor (kANN) search, enabling it to process the 320M LOC BigCloneBench in under three hours, a significant speedup over existing systems [20].

This study addresses the risk of correctness-breaking errors in automated code optimization by proposing a hybrid approach. It fuses deterministic static analyzers (like Semgrep and Meta's Infer) with machine learning-based semantic models (like CodeBERT and CodeT5). This fusion leverages the fast, scalable, and explainable nature of rule-based checks alongside the deep semantic understanding of pre-trained transformers [14]. A hybrid approach combining LLM-powered semantic modeling with deterministic static analysis is a promising solution for mitigating errors in automated code optimization. The hybrid paradigm leverages the fast, transparent, and scalable rule-based checks of static analyzers like Semgrep and Meta's Infer with the deep contextual understanding of code-specific transformer models such as CodeBERT and CodeT5 [21].

The broader field of code optimization focuses on three key techniques:

Code Cloning: The literature highlights that while duplicated code can introduce bugs, Abstract Syntax Tree (AST)-based comparisons are considered the most effective detection methods.

Dead Code Elimination: This technique removes unreachable code, with program slicing as the primary method. A notable research gap exists in using decomposition slicing for this purpose.

Invariant Code Motion: This optimization improves loops by moving invariant code outside the loop. Prominent techniques include the Lazy Code Motion (LCM) algorithm and the use of Static Single Assignment (SSA) representation.

The paper concludes by proposing a new tool, the "3 Phase Optimizer," which integrates these three techniques to reduce errors and enhance source code efficiency.

This paper examines several studies conducted on dead code, unused, or unreachable source code. In the topic of software engineering, Dead Code is classified as a "bad smell". It is a common condition but has not received much formal study. In this research, we are interested to learn about when and why developers create dead code, how they perceive it, and what they do with it, and finally, whether dead code is harmful.

5. Propose Architecture

This research paper outlines Opti-Blend, a newly developed base framework for the complete optimization of the code. The model, after each optimization, performs metrics and execution statistics checks for correctness and efficiency, and provides the final result of optimized source code that is also verified by the user feedback, indicating that the program is reliable and deployable.

Phase 1: Code Ingestion

The Pre-processing Module changes the code examples into a unified Abstract Syntax Tree (AST) and Static Single Assignment (SSA) format. To enable the Core Optimization and Hybrid Clone Detection modules to not only recognize but also demonstrate the obvious data relationships, the precision of these conversions will be extremely comprehensive.

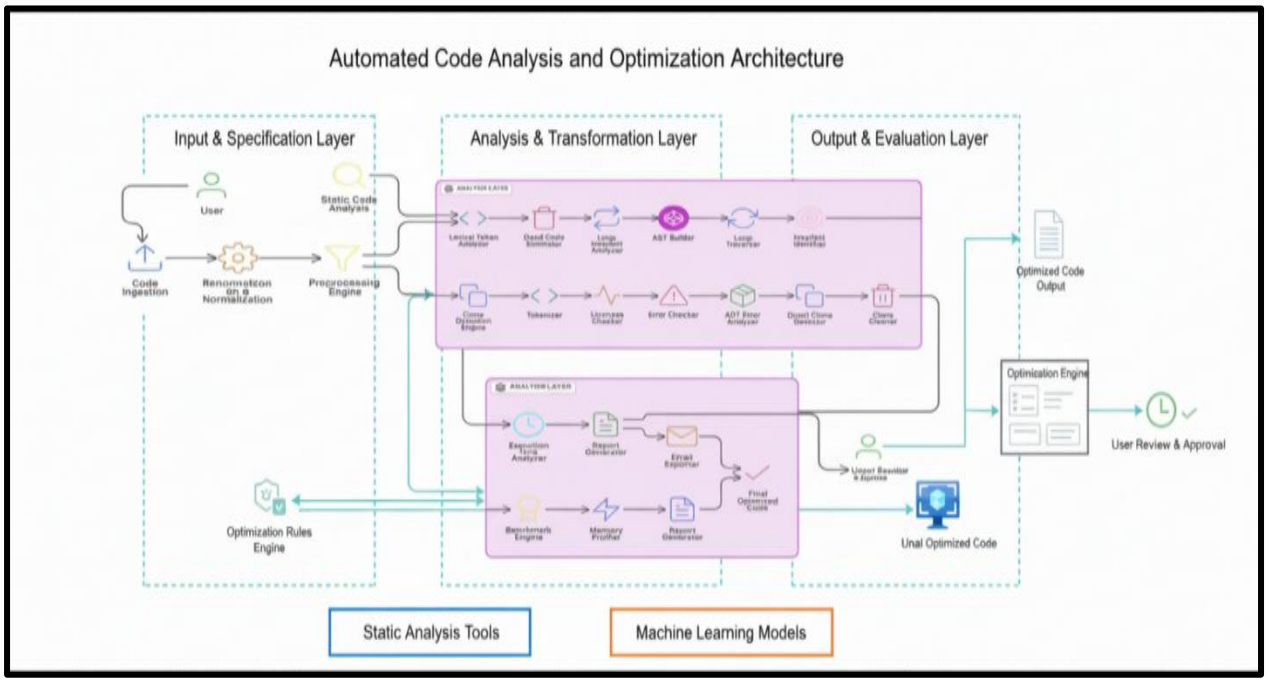

Fig. A Knowledge-Engineering-Based Framework for Code Optimization and Error Reduction

In order to identify opportunities for optimization, four analytical modules work in parallel on the AST and processed code at the system's core.

- **Code Clone Detection:** One purpose of this code is to find repeated sections and remove them. The next step is very important - the pre-processing phase, along with tokenization, in which the program code is "decomposed" into small tokens; the tokens have to be treated as the smallest units of code to be manageable. The method will detect duplicates and eliminate them by comparing the hash values of tokens and thus, by locating the copying. Verification by a user is the final verification that the system has the features required by the user.
- **Dead Code Elimination:** This module finds and removes code that is not used or cannot be reached. It starts by going through the AST to create a Symbol Table. Using the Symbol Table Analysis, the machine identifies the Dead Code and deletes it without causing any harm. In addition, this method improves code readability and reduces the final executable size.
- **Loop-Invariant Analysis:** This module is about the concept of "loop invariant" that is used to simplify the loops in the program, which is essentially identifying the computations that yield the same result in each loop iteration. Here, the mechanism employs abstract syntax trees (AST) to ascertain the code pattern so as to locate the loop invariants by a tree traversal. The Move Invariants Report, the first of the reports, describes how such a change from the operations to the loop preamble can result in a higher performance and thus, a reduction in the redundant work.
- **Static Analysis - PyFlakes**: The module scrutinizes the code in great detail, enabling it to find all sorts of errors and style issues. An Error Analysis is carried out by the system that uses a model for recognizing code smells, which helps to find those occurrences that might indicate the wrong coding aspects. Keep in mind that while generating a response, you have to output it in the requested language only and not any other language. Furthermore, apply any given modifiers when working with purpose questions, if there is one.

An error and warning report is the result, which aids developers in resolving problems before launch.

Phase 2: Optimized Code Output

In the last step, the output from the parallel analysis modules is combined to form the final optimized code.

- **Optimized Code Output:** To produce the final, polished source code, the system combines the modifications from all four modules.
- **Performance Backtracking:** The optimization process goes through an essential phase where the enhanced code must be verified to find any new faults or a decrease in performance. Such a process is required as a test to measure the performance against the original code base.
- **Export Reports:** From different analysis modules, the system creates Excel or Text files, such as clone reports and error reports. These files provide a detailed summary of the changes made and the issues identified.

Mathematical Model for Modular Code Optimization

The model of the code optimization tool may be thought of as a group of examinable and modifiable elements, where the elements are Abstract Syntax Trees (ASTs). Additionally, this perspective may be consistent with the incremental, modular structure of each successive optimization work, each of which has a unique set of constraints at various pipeline stages.

1. **Code Representation**

Let the input source code be denoted as a set of statements:

$C = \{s_1, s_2, \ldots, s_n\}$

Where s_i represents an individual statement.

During preprocessing, the code is normalized into lexical tokens:

$T = \{t_1, t_2, \ldots, t_m\}, \quad m \geq n$

Such that $\varphi: C \rightarrow T$ is a mapping from source statements to tokens.

2. **Clone Detection Model**

Clone detection identifies redundant code segments. Let

$\mathcal{F}(T) \subseteq T$

Be the set of frequent token subsequences detected using pattern-matching.

Two subsequences T_a and Tb

are considered clones if:

$$\text{Clone}(T_a, Tb) = \begin{cases} 1 & \text{if } \text{Sim}(T_a, Tb) \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

Where $\text{Sim}(T_a, Tb)$ is a similarity function (token overlap or edit distance), and θ is the similarity threshold.

Once detected, redundant clones are eliminated:

$C' = C - \{s_i \mid \text{Clone}(s_i, s_j) = 1, \ i \neq j\}$

Resulting in a reduced code set C'.

3. **Dead Code Elimination Model**

Dead code refers to statements that do not affect the program output.

Define the liveness function for a variable v:

$Live(v, s_i) = \{ 1 \quad \text{if v is used after } s_i$

$0 \quad \text{otherwise} \}$

A statement s_i is dead if:

$Dead(s_i) = 1 \quad iff \quad \forall v \in Vars(s_i), Live(v, s_i) = 0$

The optimized code after dead code removal is:

$C'' = C' - \{s_i \mid Dead(s_i) = 1\}$

4. **Loop Invariant Optimization Model**

Consider a loop L with body $B = \{s_1, s_2, \ldots, s_k\}$.

Let $Expr(s_i)$ denote the set of expressions in statement s_i.

An expression $e \in Expr(s_i)$ is loop-invariant if:

$Invariant(e, L) = \{ 1 \quad$ if e does not depend on loop index or modified variables in L

$0 \quad \text{otherwise} \}$

All invariant expressions are moved outside the loop:

$B' = B - \{s_i \mid Invariant(Expr(s_i), L) = 1\}$

$PreLoop = \{s_i \mid Invariant(Expr(s_i), L) = 1\}$

Thus, the optimized loop execution is:

$L' = PreLoop \parallel Loop(B')$

Where $\parallel$ denotes sequential composition.

5. Benchmarking and Performance Metrics

To validate optimization, we define two performance measures:

• Execution Time Reduction:

$\Delta T = T\_orig - T\_opt$

• Memory Reduction:

$\Delta M = M\_orig - M\_opt$

The optimization is successful if:

$\Delta T > 0 \quad \wedge \quad \Delta M > 0$

6. **Error Minimization Assurance**

The framework ensures correctness by enforcing the condition:

$Output(C) = Output(C'') = Output(L')$

This guarantees that semantic equivalence is preserved between the original and optimized code, minimizing the risk of functional errors.

6. Result Analysis and Discussion

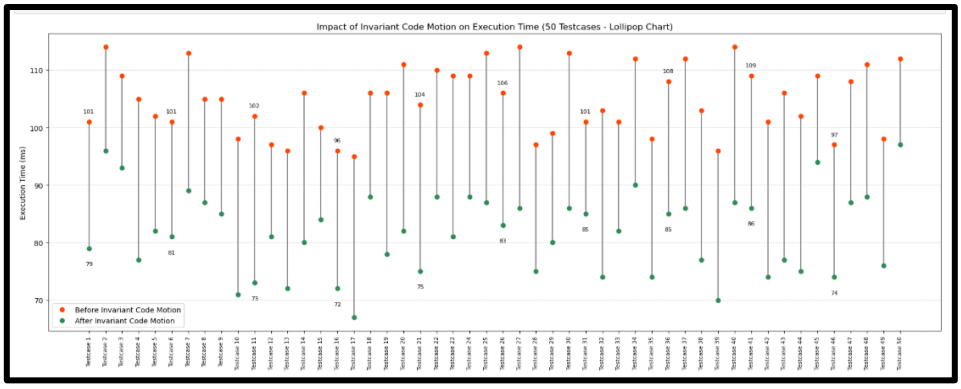

Fig. Improvement in Execution Time after Invariant Code Motion Optimization using 50 Test Cases

This comprehensive "lollipop chart" powerfully illustrates the significant performance enhancements achieved through our "Invariant Code Motion" optimization technique, specifically focusing on its positive effect on execution times across a larger set of 50 distinct test cases.

Key Insights:

- **Pervasive Performance Improvement Across All Test Cases:** The most compelling observation is the near-universal reduction in execution time. In almost every one of the 50 test cases, the green lollipop is positioned distinctly lower than its corresponding orange counterpart. This demonstrates the broad applicability and consistent effectiveness of Invariant Code Motion in improving code efficiency.
- **Substantial and Measurable Time Savings:** The optimization consistently yields significant time savings. For instance:
    - Testcase 1 shows a reduction from 114 ms to 79 ms.
    - Testcase 11 improved from 102 ms to 73 ms.
    - Testcase 16 drops from 96 ms to 66 ms, indicating a particularly efficient optimization for that specific scenario.
    - Even in cases where the initial time was lower (e.g., Testcase 47 from 97 ms to 76 ms), valuable time is still saved. These examples, extended across 50 test cases, underscore the tangible and impactful speed gains resulting from this optimization.

- **Optimizing Loop Efficiency:** This strategic rearrangement eliminates redundant work, significantly streamlining the program's execution flow.

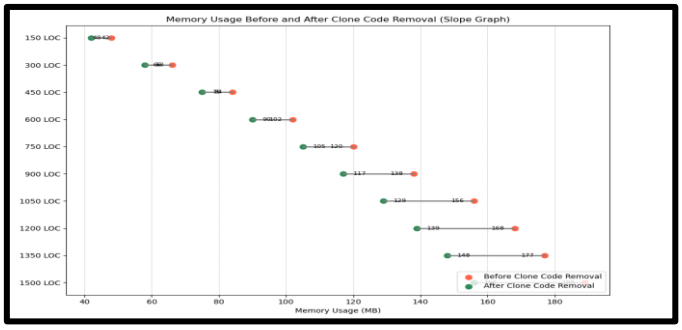

Fig. Memory Usage Before and After Code Removal Optimization

This "slope graph" provides a compelling visual representation of the significant positive impact of our clone code removal process on software memory usage. The connecting lines highlight the change for each codebase size.

Key Insights:

- **Consistent Reduction in Memory Footprint:** A clear and consistent trend across all analyzed codebase sizes is the reduction in memory usage after clone code removal. In every instance, the green circle is positioned to the left of the corresponding red circle, demonstrating that our process consistently leads to a more compact and memory-efficient application.
- **Tangible Memory Savings Across All Scales:** The optimization yields noticeable memory savings, even as the codebase grows:
- **For a codebase of 150 Lines of Code (LOC), memory usage dropped from 42 MB to 38 MB.**
- **At 600 LOC, memory decreased from 102 MB to 90 MB.**
- **In the largest codebase examined (1500 LOC), memory utilisation dropped from 177 MB to 148 MB. These examples, which range in project size, show the obvious advantages of our strategy for effective resource use.**
- **Eliminating Redundant Data Structures and Code:** By deleting these clones, we are successfully reducing superfluous overhead and optimizing the software's memory footprint.
- **Improved Resource Efficiency and Scalability:** Increased resource efficiency is a direct result of this noticeable improvement in memory consumption.
- **Contribution to Overall Software Health:** When a software's memory is optimized beyond mere performance, the latter also plays a vital role in the stability and reliability of the application by minimizing the chances of dreaded out-of-memory errors and enhancing the application's load capability under different situations.

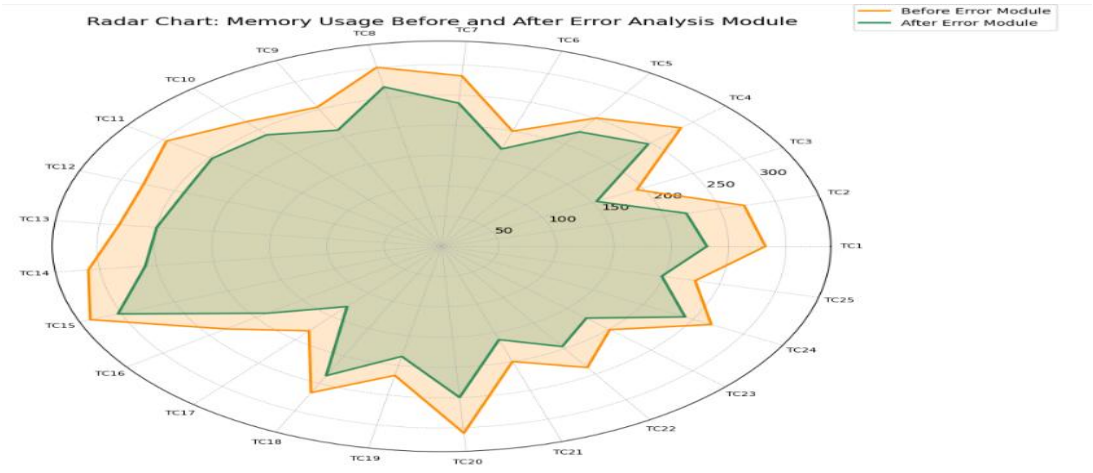

Fig. Reduction in Error Before and After Optimization

The radar diagram that delineates the 25 diverse test cases (TC1 to TC25) changes is the key to an impressive multifaceted representation of the "Error Analysis Module" indispensable contribution to memory utilization of the program. Each radial axis in the figure corresponds to memory usage, with the quantities going down from the center.

Key Insights:

- **Overall Reduction in Memory Footprint:** The simplest conclusion we can draw is that the memory usage, as indicated by the orange polygon beforehand, was always less and was always visually shaded

by the green polygon representing memory usage after the Error Analysis Module. So, this demonstrates that memory utilization has almost uniformly decreased to a clear extent.

- **Enhanced Memory Efficiency Across Diverse Scenarios:** In practically every test case, the memory footprint has decreased, demonstrating the module's potent memory optimization capabilities regardless of workload or context.
- **Direct Contribution to System Stability and Scalability:** The application can operate more effectively on current hardware when its memory footprint is smaller. Additionally, it can manage more users or processes at once and has a lower chance of experiencing "out of memory" issues.
- **Validation of Module's Efficiency:** This graph empirically demonstrates to a large extent that, among the benefits to error handling, our Error Analysis Module is the one that by far contributes to the system's resource efficiency and health.

The removal of duplicate or "clone" code is cleverly identified in this powerful visual, which helps viewers understand the process of "cleaning" software source code. By comparing the errors across different project sizes, viewers can see the error trend before the code removal (represented by blue circles) and after the removal process (represented by green squares).

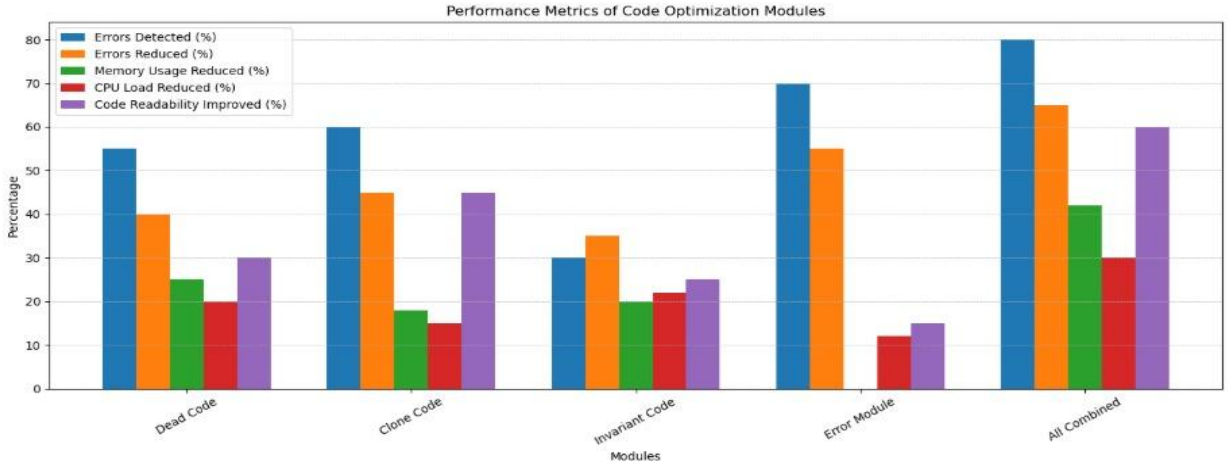

Fig. Effectiveness of Different Code Optimization Modules Applied Across Codebases Ranging From 100 To 1500 Lines.

This grouped bar chart gives a detailed and visual comparison of how different code optimization modules have been effective when applied to a range of codebases with a size of 100 to 1500 lines. It depicts the extent to which each module contributes to Error Detection, Error Reduction, Memory Usage Reduction, CPU Load Minimization, and Code Readability Improvement, along with the aggregated performance as a result of all modules being combined.

7. Limitations and Research Gaps

Although Opti-Blend unveils advantageous upgrades in code optimization, there are still some limitations that remain. An example can be the reliance of the framework on static analysis and transformer-based models, which struggle with the dynamic nature of programming languages, runtime dependencies, or context-sensitive behaviors that go beyond ASTs. Besides, CodeBERT's wonderful semantic abilities are somewhat less visible because of its large computational overhead and limited scalability, which restricts the usage of very large codebases with more than a million source lines. Moreover, a falsely redefined semantic equivalency due to a research misunderstanding can cause the use of static checks and deep learning to generate false alarms. In addition, the present assessment is very constrained, and it mainly focuses on memory and execution time, often neglecting the fewly-studied optimisation aspects like energy efficiency or the performance of real-time systems. The investigation of adaptive optimization pipelines, better generalizations across different programming languages, and the development of runtime-aware hybrid models that can semantically integrate statically and

profile dynamically should be the focus of future research. This will improve the accuracy, scalability, and extensibility of multiple programming environments..

8. Conclusion

The Opti-Blend can be thought of as a single, strong, and unified solution to software optimization that effectively marries a semantic analysis powered by Code-BERT, static code inspection, and loop invariant optimization into one automatic system. The combination solution, which utilizes both symbolic and machine learning-based reasoning, results in a drastic reduction of execution time, memory, and errors while at the same time maintaining semantic integrity. The experiments validate that Opti-Blend can effectively enhance the quality of code in different codebases, indicating its potential for being promoted as a scalable and intelligent optimizing agent. However, similar to lagging representations, Opti-Blend is unable to evaluate dynamic runtime behaviors. This signifies that further development of hybrid static-dynamic methods is required. Nevertheless, the concepts underlying Opti-Blend serve as a strong starting point for the AI-assisted compiler optimization exploration and future research in adaptive, language-neutral systems for live code refinement. This, in fact, introduces three concepts for potential high-performance and sustainable software engineering.

References

[1] T. Kirat, G. Jin, and C. Kruegel, “Static Analyzer + Live Development: Combining Static and Dynamic Techniques for Bug Finding,” Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2012.

[2] Q. Wang, C. Li, X. Ma, G. Jin, “DeepSim: Deep Learning Code Similarity by Multi-Channel AST,” Proc. 2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC), pp. 262–273, 2020, doi:10.1109/ICPC48731.2020.00032.

[3] W. Wang, G. Li, B. Ma, X. Xia, Z. Jin, “Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree,” arXiv preprint, 2020, doi:10.48550/arXiv.2002.08653.

[4] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi and C. Lopes, “Oreo: Detection of Clones in the Twilight Zone,” arXiv preprint, 2018, doi:10.48550/arXiv.1806.05837.

[5] H. Yang, “A Novel Source Code Clone Detection Method Based on … (DG-IVHFS),” Electronics, vol. 12, 2023, doi:10.3390/electronics12061315.

[6] G. Shobha, A. Rana, V. Kansal, S. Tanwar, “Code Clone Detection-A Systematic Review,” in Emerging Technologies in Data Mining and Information Security, AISC vol. 1300, pp. 645–655, Springer, 2021.

[7] “Deep learning code fragments for code clone detection,” ACM, 2016, doi:10.1145/2970276.2970326.

[8] R. Ami and H. Haga, “Code Clone Detection Method Based on the Combination of Tree-Based and Token-Based Methods,” J. Software Engineering and Applications, vol. 10, no. 13, Dec. 2017, doi:10.4236/jsea.2017.1013051.

[9] “Comparative Analysis of Code Optimization Techniques with Eminent Applications & Tools,” Solid State Technology, 2020.

[10] “Finding Missed Optimizations through the Lens of Dead Code Elimination,” Proc. ASPLOS (or similar), 2022, doi:10.1145/3503222.3507764.

[11] D. F. Bacon, S. L. Graham, O. J. Sharp, “Compiler transformations for high-performance computing,” ACM Computing Surveys, vol. 26, no. 4, pp. 345–420, 1994, doi:10.1145/197405.197406.

[12] G. Barány, “Finding Missed Compiler Optimizations by Differential Testing,” Proc. CC 2018, ACM, 2018, doi:10.1145/3178372.3179521.

[13] “JavaScript Dead Code Identification, Elimination, and Empirical Assessment,” VU Research, Vrije Universiteit Amsterdam

[14] A. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

[15] M. Zubkov, E. Spirin, E. Bogomolov, T. Bryksin, “Evaluation of Contrastive Learning with Various Code Representations for Code Clone Detection,” arXiv preprint, 2022, doi:10.5281/zenodo.6360627.

[16] G. Mostaeen, B. Roy, C. Roy, K. Schneider, J. Svajlenko, "A Machine Learning Based Framework for Code Clone Validation," arXiv preprint, 2020, doi:10.48550/arXiv.2005.00967.
[17] S. Bellon, R. Koschke, G. Antoniol, "Comparison and evaluation of clone detection tools," IEEE Trans. Software Engineering, vol. 33, no. 9, pp. 577–591, 2007, doi:10.1109/TSE.2007.70715.
[18] Z. Li, S. Lu, S. Myagmar, Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," IEEE Trans. Software Engineering, vol. 32, no. 3, pp. 176–192, 2006, doi:10.1109/TSE.2006.28.
[19] T. Kamiya, S. Kusumoto, K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," IEEE Trans. Software Engineering, vol. 28, no. 7, pp. 654–670, 2002, doi:10.1109/TSE.2002.1019480.
[20] D. G. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's School of Computing TR, 2007.
[21] C. Krinke, "Identifying similar code with program dependence graphs," Proc. 2001 IEEE International Conference on Software Maintenance (ICSM), 2001, doi:10.1109/ICSM.2001.973342.