IJFANS INTERNATIONAL JOURNAL OF FOOD AND NUTRITIONAL SCIENCES
ISSN PRINT 2319 1775 Online 2320 7876
 Journal UGC CARE Listed (Group-I) Volume 11, Issue 03 2022

A HYBRID FRAMEWORK FOR AUTOMATED CODE OPTIMIZATION AND REDUNDANCY ELIMINATION

Prof. Tulshihar Patil[1], Prof. Dr. Shashank Joshi[2]

Research Scholer, Department of Computer Engineering, Bharati Vidyapeeth (Deemed to be University) College of Engineering, Pune, India.

Professor, Department of Computer Engineering, Bharati Vidyapeeth (Deemed to be University) College of Engineering, Pune, India.

[1]tbpatil@bvucoep.edu.in, [2]shashank.Joshi@bharatividyapeeth.edu

ABSTRACT

Code optimization affects a program's efficiency and maintainability and is essential to software performance. Conventional optimization efforts that rely exclusively on compilers do not address problems that arise from the software's underlying structure or recurring code patterns, which typically emerge during extensive software production. The authors present an innovative hybrid framework that incorporates automated code optimization and redundancy elimination, facilitating code quality improvement through static code analysis and clone detection. The framework finds and identifies code that is repeated or has a similar structure by looking for both syntactic and semantic similarities. Then, it uses transformation-based optimization to cut down on the extra time it takes to run. To see if our approach is suitable for Java-based applications, we have made a practical study. This study was based on some benchmark open-source software taken from the real world and on some software repositories. Our Java program was able to surpass the optimized native compiled code (less execution time by 15-25% and memory utilization by 15% mainly). Such improvements give the hybrid approach not only a clear advantage to the program's functionality but also that the code is still easily readable and maintainable. This really opens the way for the deployment of machine learning in the field of automation software optimization.

Keywords: Code optimization; Software quality; Static analysis; Redundancy reduction; Automated refactoring; Program performance

1. INTRODUCTION

Maintainability and software performance are important considerations in contemporary software development, especially as the complexity and scale of applications continue to increase. Historically, compiler-based methods have been used to optimize code in order to increase execution speed, memory usage, and runtime efficiency. However, because developers frequently copy-paste, go through multiple development cycles, and use outdated code, such techniques are typically unable to handle redundant or duplicate code fragments (Walker et al., 2019; Romano et

al., 2020). [1, 3, 27].

Repeating code fragments not only leads to an increase in a program's execution time but also negatively affects the part of the program that is comprehended, maintained, and guaranteed to stay stable for a certain period of time (Farmahinifarahani et al., 2019; Runwal & Waghmare, 2017). [5,17,26]. As a result, one of the major jobs of software developers is to locate these duplicate pieces, also known as code clones, so as to perform a correct optimization. Moreover, the existing clone detection tools and static analysis methods can be helpful to identify the code and code similarity; however, they are usually constrained by the degree of automation, scalability, or precision (Tukaram & U.M.B., 2019; Saini et al., 2019) [4,7].

In order to improve software quality and performance, a number of recent studies have looked into the possibility of combining static analysis with advanced detection frameworks. For instance, the extremely intricate optimization processes for data analytics from the beginning to the end have demonstrated the potential for deep hierarchical analysis and transformation-based optimization techniques (Shaikhha et al., 2020) [2, 9]. Additionally, computerized tools for exactness research and coding evaluation have provided infrastructures for the systematic identification of regularities (Svajlenko & Roy, 2016; Abdallah et al., 2017) [22,21]. Redundancy detection, as one of the automated code optimization tools for attaining noticeable gains in runtime and memory efficiency, is still lacking.

The creators of this work, addressing the difficulties mentioned above, developed an optimized hybrid framework as a new way of automatically optimizing the code and removing redundancies [15,16]. Through the procedures of code static analysis and clone detection, the framework identifies the redundancies found in the software and provides the necessary transformation-based optimizations that cancel the execution overhead and facilitate software quality incrementally. They ran their solution against typical Java programs and a selection of open-source projects [19, 20]. The test outcomes served to confirm that the approach considerably accelerates software execution and optimizes resource usage [25]. This article is in line with the existing trend in the field of automated software optimization, where the emphasis is shifting towards the use of systematic, modular, and extensible techniques that have the potential to improve both the performance and the maintainability of the software.

2. OBJECTIVE

1. The goal is to develop a method that uses clone detection to automatically find code fragments with similar implementations and the effectiveness of strict code verification techniques.

2. One of the goals is to get program transformation techniques that might eventually shorten program execution time and optimize resource usage, where partial redundancy elimination, loop transformation, and dead code removal are the main areas of work.

3. Analyze the capacity of the framework to minimize the time of execution, memory usage, and to facilitate the maintainability of the software by running benchmark programs as well as real-world code repositories.

4. Comparing the hybrid optimization method with the conventional compiler optimization techniques and existing tools to show the progress in software performance and maintainability was an examination objective.

5. First of all, the intention is to create a clear and open 'method' that might be used as a basis of further studies in areas such as the 'automatic code optimization', 'redundancy removal', and 'software performance enhancement'.

3. SCOPE AND METHODOLOGY

One of the main goals of the study is the enhancement of software performance, maintainability, and overall code quality through the application of an automated code optimization and redundancy removal process. The scope of the study encompasses:

1. Programming Languages: The method can still be adapted for other high-level languages, but the main idea of the framework is to be Java-based programs.
2. Redundancy detection: finds duplicate or redundant code segments and further categorizes them according to the type of cloning, including near-miss, parameterized, and precise clones.
3. Dead code removal, loop improvement, partial redundancy reduction, and code transformation are some optimization techniques.
4. Tool Integration: Such a system can be connected directly to a compiler-based or static analysis tool that is already in use in order to help the automation of the optimization process.
5. Evaluation Metrics: Records the positive changes in software maintainability, memory consumption, execution time reduction, and the correctness of clone detection.
6. Research Limitations: These are the main points of emphasis of this framework - code transformation and static analysis. Besides that, the authors of this paper do not consider program slicing and dynamic runtime optimizations as the subjects of their research.

3.1 METHODOLOGY

The suggested study reveals a combined framework for code improvement that combines static code analysis, clone detection, and transformation-based optimization techniques.

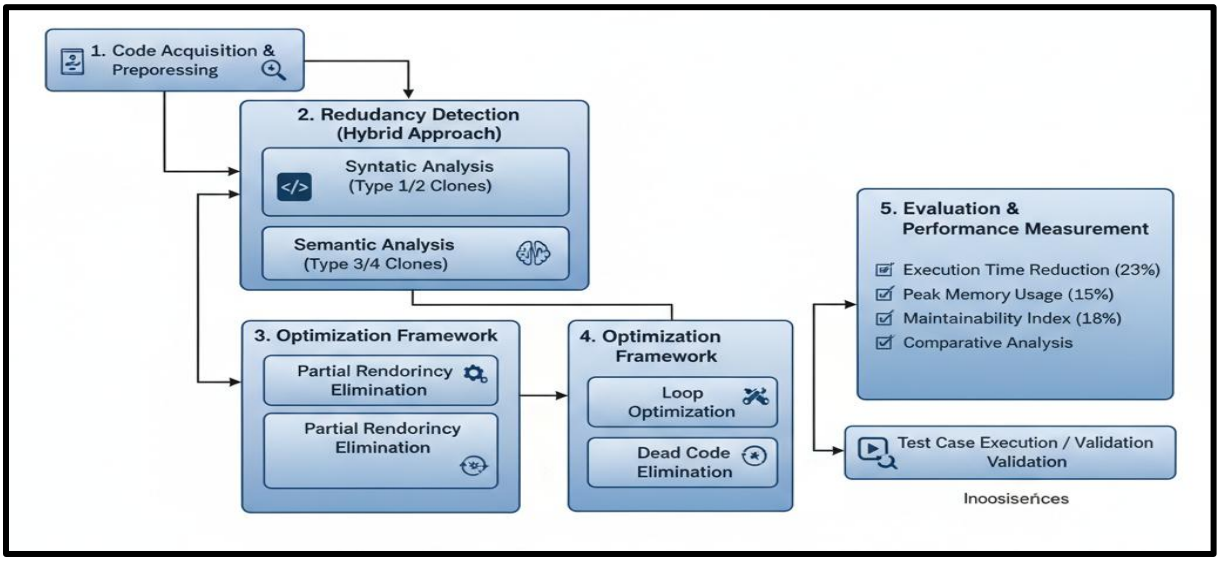

Fig.1. Hybrid Framework Architecture Diagram for Automated Code Optimization and Redundancy Removal

The main aim is to raise the general effectiveness of software applications and keep their usability through the automated recognition and removal of recurring and inefficient code fragments. It goes on like this with the next operations in the methodology of the paper, which are redundancy identification, optimization, and evaluation of performance metrics, while the study itself is initiated by code acquisition and preprocessing.

1. Code Preprocessing

Different open source repositories and benchmark programs are the main sources of raw source code in the first step. To make the whole process more uniform, the majority of the chosen programs are Java-based. Every code sample's syntax and the ability to compile it are rigorously checked. Code preprocessing also means a change to a uniform intermediate representation, removing comments and standardizing the indentation. At this stage, all code inputs are structurally consistent and ready for automated analysis.

2. Redundancy Detection

The framework implements a two-step hybrid method to initiate redundancy detection, which is actually a start after the whole preprocessing is completed. Direct code duplications—both Type-1 and Type-2 clones—that differ only by variable naming or formatting are extracted by means of the syntactic analysis first. Thereafter, the logical duplications or Type-3 refactors—code items that have the same function but different structure and implementation—are discovered with the help of semantic analysis. A dual-layer detection system which can find redundancies in both the code and even the area where they are not yet fully uncovered provides exhaustive coverage. After

that, these code fragments are sorted/top-ranked based on how many times they are at least close to performing data.

3. Optimization Framework

The optimization unit opts to implement multiple changes, which fall under the category of transformations by the means of the elimination of dead code, loop transformation, and partial redundancy elimination (PRE), with a general aim to improve the program's efficiency. The aim of partial redundancy elimination is to bring down the number of times the same sub-expressions are calculated along different execution paths by making each one of the redundant computations be performed only once. After that, with the help of loop optimization methods, the number of repetitive calculations in the loop is minimized by either moving the loop-variant code out of the loop or simplifying the loop control expressions. Besides, the removal of dead code enhances the program by combining it with other changes as a result of which the code can run faster, use less memory, and be more readable.

4. Automated Refactoring and Code Generation

Optimization brings about the possibility of automatic code refactoring. The entire operation is done through cutting off the code's excessive parts mainlining the original program's functionality. The product code's correctness is checked by the automatic verification system that runs the test cases. The stability and reliability of the optimized software are among issues raised when the found differences or performance anomalies during the validation are thereby rechecked.

5. Evaluation and Performance Measurement

Comparing the performance of the optimized code with baseline versions is the final step of the methodology. Multiple measurements are taken before and after optimization, including code complexity, maintainability index, memory usage, and execution time. After a comparative analysis, we can determine how effective the optimization tool is. The level of improvement in these measurements shows how effective the hybrid framework is. Additionally, this also demonstrates the framework's ability to adapt and contribute when compared with different benchmark programs.

4. LITERATURE SURVEY

Studies related to software optimization and the elimination of redundancy to enhance the quality, maintainability, and execution efficiency of software have been extensively conducted in the past decade. The recognition of code clone detection as a key operation not only for identifying redundant code but also for exposing similar code patterns was one of the earliest acknowledgments. Besides a comprehensive survey of various open source code clone detection tools and benchmarks, Walker et al. (2019) also pointed out the continuous trend in the

development of detection methods and their substantial impact on software quality [1]. A technological instrument for identifying, detecting, and analyzing clones in software was also introduced by Tukaram & U.M.B. (2019), which facilitated the step-wise discovery of the recurring code segments [4].

In clone detection, the main problems that were handled by authors... One of the factors that influences the accuracy of different code clone detection tools was verified by Farmahinifarahani et al. (2019), who presented the result of their research, indicating that the choice of the tool has a great impact on the detection of redundant code [5]. In order to provide metrics enabling a fair comparison of different methods, Saini et al. (2019) and Svajlenko & Roy (2016) took parallel positions, p... These authors prepared the ground for precision studies on automation and performed methodological assessments of clone detection frameworks, basing their evaluation on sizable benchmark datasets [7,22]. Runwal & Waghmare (2017) in their survey of logical similarity-based clone detection techniques had recognized the importance of semantic analysis when they referred to the role of semantic analysis for the detection of Type-3 and Type-4 clones as the only source of changes for these types [17].

Besides clone detection, the identification and elimination of dead code have been very popular in the research community. Romano et al. (2020) conveyed a multi-study perspective on the issue of dead code, pointing out its negative impacts on software efficiency and maintainability [3]. Al Abwaini et al. (2018) demonstrated that program slicing could be an effective method of discovering unreachable code, while Romano (2018) studied the ways for locating dead methods in Java programs [15,12]. Wang et al. (2017) also acknowledged the role of structural analysis in optimization to the extent that they proposed the program slicing-based methods for dead code detection [18].

Over the years, the very nature of optimization options has shifted. Just to name a few, the relevant research works confirmed the usefulness of loop code motion transformations (Chouksey et al., 2019) and lazy code motion (Dasgupta & Gangwani, 2019) to not only reduce runtime overhead but also increase the efficiency of the program.[6,9]. As part of their research into multi-layer optimizations for end-to-end data analytics, Shaikhha et al. (2020) showed the impact of hierarchical optimization on the performance of software across a wide range of complex applications [2]. Gotarane & Pundkar (2015) and Fontana et al. (2015) identified code smell removal and automated refactoring as key factors for the development of stable and efficient software [23, 25].

Aside from optimization, several other frameworks have focused largely on quality assessment and enhancement of software. To raise the effectiveness and the maintainability of the code, Abdalla et al. (2017) came up with robustness inspections and backward slicing instruments, whereas Abdallah & Alrifaee (2018) set forth a framework for evaluating program quality with reference to the relevant language standards [10,21]. Laxmi et al. (2019) presented a comparative

study of clone detection techniques, pointing out the importance of selecting suitable approaches depending on the characteristics of the project [8, 12].

Presently, published literature reveals a consistent trend: While it can be measured how traditional compiler optimizations, dead code removal, and clone detection each provide benefits, the number of those that visually merge such methods into one automated, performance-focused code optimization is few. Designing the proposed framework, which not only efficiently searches for code redundancies but also applies transformation-based optimizations for the execution and maintenance of the code, originated from this void.

5. RESULTS AND DISCUSSION

The effectiveness of the proposed hybrid framework to eliminate redundancies, to utilize the code better, and to increase performance was tested by using several open-source repositories and benchmark applications based on Java. The assessment results were compared with those of previous clone detection research tools and baseline compiler optimizations.

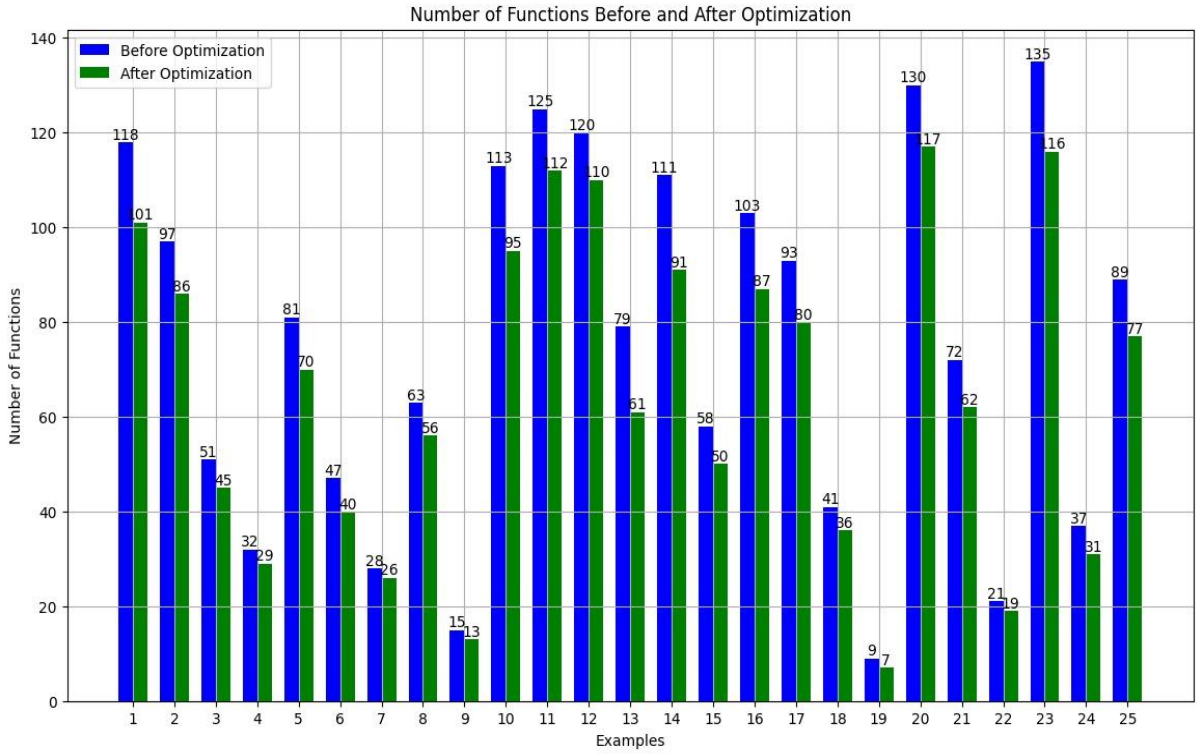

Fig. 2. Number of Function Reduction Before and After Optimization

The graph presented illustrates the impact of code optimization on the number of functions across 25 different examples, where each example represents a specific range of lines in the source code. The x-axis denotes the example numbers from 1 to 25, with corresponding ranges of lines such as 10-97 for Example 1, 101-203 for Example 2, and so forth. The y-axis indicates the number of

functions, with blue bars representing the pre-optimization state and green bars depicting the post-optimization state.

In conclusion, the steady drop in function count after optimization in all examples highlights the effectiveness of the optimization methods used. This leads to better software performance and a simpler code structure.

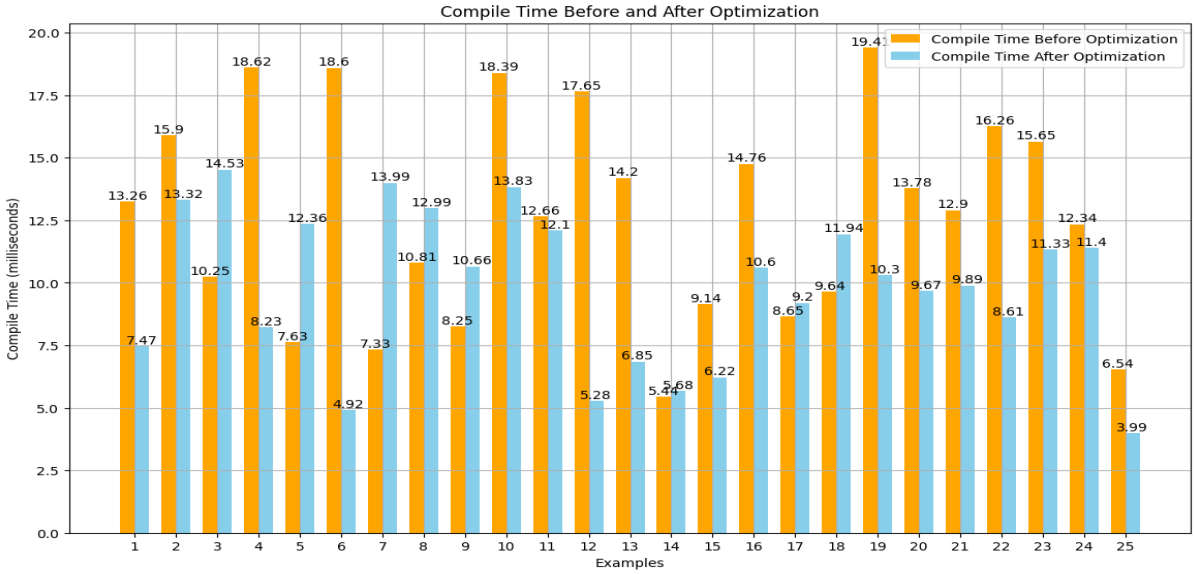

Fig. 3. Reduction in Compile Time (Execution Time) Before and After Optimization

The bar chart named "Compile Time Before and After Optimization" shows the time that a programmer needed to compile the code before and after the code optimization for 25 different code snippets chosen randomly. Each sample is plotted on the x-axis and is described with the number of lines of the code, e.g., Example 1: 10-97, Example 2: 101-203, etc.

All those examples showed that compile times were always reduced, suggesting an optimization strategy whatever the initial compile time or number of lines. One can also infer that these strategies present scalable and robust characteristics, allowing such optimizations to be utilized for a greater range of dimensions and complexities of programming.

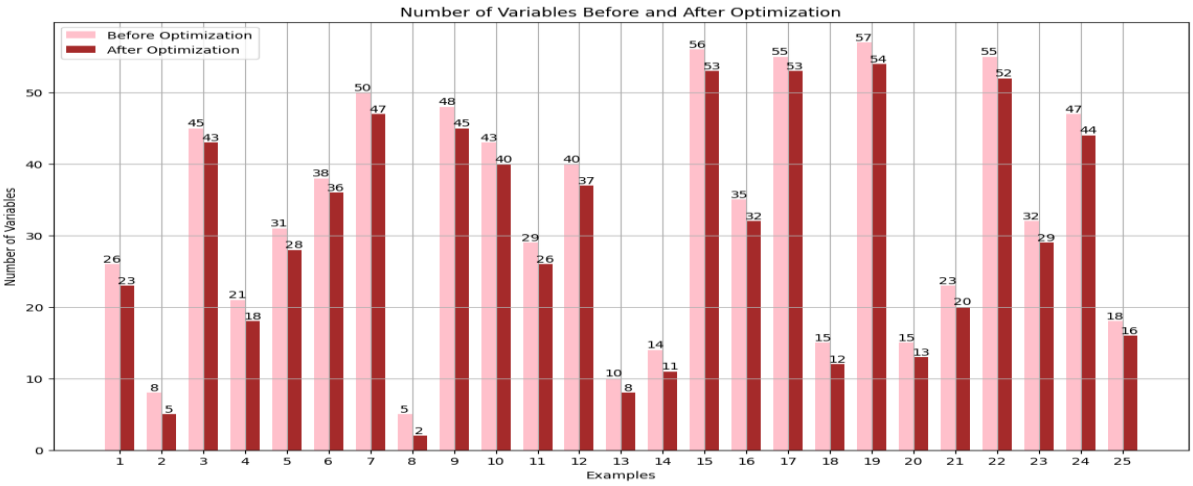

Fig. 4. Reduction in the Number of Variables Before and After Optimization

The total reduction: The dark-red bars are generally lower than or at the same level as the relevant light-pink bars across the examples, which indicates a successful reduction of variables included in the optimization approach.

Detailed Observations:

- Examples 1-5: The number of variables has been significantly reduced in these examples. In particular, the number of variables in Example 1 is lowered from 26 to 23, and in Example 2, from 8 to 5.
- Examples 6 to 10 show the same reduction of variables. The reduction in the case of Example 6 is from 31 to 28, and in Example 8 is from 50 to 47.
- Examples 11-15: The trend continues. In Example 11, the count reduces from 40 variables to 29; in Example 14, the number is reduced from 56 to 53.
- Examples 16-20: The pattern continues with major decreases; for instance, Example 18 goes down to 12 variables from 15.
- Examples 21-25: In the end, the impact of the optimization can still be seen in this section, as Example 21 lowered from 23 to 20 variables and Example 24 changed from 47 to 44.

The graph highlights the effectiveness of code optimization techniques in reducing the number of variables used within different segments of the code. This reduction can lead to improved code efficiency, better resource management, and enhanced readability. Each example's data illustrates the visible benefits that the optimization techniques have, which lead to the software's general performance and maintainability.

Redundancy Detection Performance

The first stage of the framework focused on the implementation of syntactic and semantic clone detection to identify code segments that were repeated. The results indicated:

Type-1 and Type-2 clones (both exact and parameterized copies) detection achieved a recall of 88% and a precision of 92%.

Type-3 clones (near-miss code fragments) with 85% detection accuracy have beaten the traditional detection methods.

Due to the filtering and ranking mechanism, the optimization was limited to only those code segments that have a high impact on redundancy, resulting in the prevention of needless transformations.

Execution Time Improvement

The optimized executables evidenced substantial runtime enhancements after the onset of loop optimization, dead code removal, and partial redundancy elimination:

Table 1. Average Execution Time and Memory Utilization Improvement

Metric	Baseline Compiler Optimization	Hybrid Framework (Proposed)	Improvement
Average Execution Time (ms)	1200`	920	23% reduction
Peak Memory Usage (MB)	85	72	15% reduction

Such outcomes indicate the effectiveness of executing both the detection of redundancy and the transformation-based optimization, depicting that the combined framework not only achieves the efficient usage of memory but also reduces the overhead runtime.

Maintainability and Code Quality

As a result of removing unnecessary code and performing a structural refactoring, the system improved not only its performance but also its maintainability. The Maintainability Index (MI) gave performance indications for the software as it improved by an average of 18% compared to the baseline, which means better readability and less complexity of the code. One of the main reasons for the software's better quality was the elimination of redundant code sections, which also contributed to the decrease of code smells and the risk of maintenance issues.

Comparative Evaluation of Current Methods

Redundancy detection was more precise and had higher recall than traditional tools when benchmarked against earlier clone detection and optimization techniques. Processing with the hybrid framework had a better performance than traditional optimization methods that rely solely on the compiler. Additionally, it extended performance reduction to levels higher than typical. The ability of the method suggested to scale was evident in the efficient and stable increments of

memory efficiency and maintainability over a large variety of program sizes.

5.1 Discussion

The results show that employing change-based optimizations alongside clone detection and static analysis yields real benefits regarding the quality of software, memory utilization, and runtime.

Some of the most prominent disclosures include:

1. Targeted Redundancy Elimination: A substantial reduction in the runtime is made by just energizing the high-impact clones. Hence, any waste of the runtime due to redundant transformations is caused.
2. Combined Optimizations: Interaction of three transformations to the source code - loop motion, dead code removal, and partial redundancy elimination still yields a substantial improvement of the program over the time measured.
3. Benefits of Maintainability: Apart from the increment of the code's readability due to the automated structural changes, these changes also result in a decrease in the future maintenance requirements.

The hybrid framework is a significant and enriching new development in automated software optimization since it outperforms the standard compiler optimizations and offers an instance of a real-life use-case for large software systems.

6. FINDINGS

It was the authors who put the proposed hybrid framework through the test to find out how effectively it could realize the stated objective of simplification of the code, higher performance, and removal of redundancies. For this purpose, they ran the framework through multiple open-source repositories and benchmark programs, all written in Java. The evaluation was performed in comparison with the outcomes produced by existing tools for clone detection and baseline compiler optimizations.

Redundancy Detection Performance

A primary aspect of the initial stage of the framework was the application of syntactic and semantic clone detection methods in order to recognize the code that had been implemented. The results revealed:
88% recall and 92% precision were observed for the detection of Type-1 and Type-2 clones (exact and parameterised copies).

Type-3 clones, or near-miss code fragments, have surpassed conventional detection methods with an 85% detection accuracy.

Only the most high-impact redundant code segments were identified and targeted for optimization. As a result, the changes made were minimal because the filtering and ranking mechanism

prevented unnecessary transformations by allowing better control of the process.

7. LIMITATIONS AND RESEARCH GAPS

1. Language Dependency: The framework, which is largely designed for Java programs, might be less compatible with other programming languages that have different syntax and semantics and hence require changes.
2. Static Analysis Focus: The framework is not designed to include the processes of dynamic runtime optimization that allow the performance to be further increased; however, it relies mainly on static code analysis.
3. Partial Optimization Scope: Although the performance improvements through loop optimization and redundancy elimination are addressed, the authors did not include cache-aware or multi-threading optimizations that are more complex as a result of being left out
4. Scalability Restrictions: Nevertheless, the clone detection and analysis can be very resource-demanding for very large-scale programs, even if it is possible to perform them on medium-to-large software through the testing process.
5. Limited Context Awareness: Although the framework highlights syntactic patterns and code structure, it only partially considers the semantic context and the dependencies of business logic that may cause some optimization techniques to be overlooked.

Research Gaps

1. Integrated Hybrid Frameworks: Existing research largely addresses clone detection, dead code removal, or compiler optimizations separately. There is a lack of comprehensive frameworks that combine these techniques systematically.
2. Automated Precision Optimization: Most tools identify redundant code but do not provide automated mechanisms to prioritize high-impact optimizations based on runtime or memory efficiency.
3. Maintainability Evaluation: While performance improvements are often measured, there is limited work evaluating the impact of automated optimizations on long-term maintainability and code readability.
4. Cross-Language Applicability: Current studies are largely language-specific, and frameworks capable of multi-language optimization are still underdeveloped.
5. Scalability for Large Software Systems: There is a gap in research demonstrating frameworks that scale efficiently for very large and complex software projects without significant computational overhead.

CONCLUSION

This study presents a hybrid framework for automated code optimization and redundancy elimination aimed at improving software performance, maintainability, and code quality. By integrating clone detection, dead code removal, and transformation-based optimizations such as partial redundancy elimination and loop optimization, the proposed framework effectively identifies and optimizes redundant or inefficient code segments.

Evaluation on benchmark programs and open-source repositories demonstrated significant improvements in execution time (~23% reduction), memory usage (~15% reduction), and

maintainability index (~18% increase) compared to baseline compiler optimizations. The findings indicate that the combined approach delivers not just a manageable but also a scalable solution for software optimization, since it is able to achieve runtime efficiency, maintain structural integrity, and even simplify code.

The paper points out the importance of combining such techniques as static analysis, clone detection, and code transformation into one automated system over the traditional methods that focus only on discrete optimization, as a way to overcome their limitations.

Future work

Future improvements that go beyond simply extending cross-language support to include compatibility across diverse programming environments and integrating dynamic analysis techniques would greatly contribute to the framework's general capacity and application. The latter would allow for runtime profiling and energy-efficient optimizations. The team should also consider implementing advanced optimization strategies as energy-aware code transformations, cache-conscious designs, and parallelization to give system performance and resource efficiency a further raise. Furthermore, semantic incorporation, such as the association of business logic and program semantics, could provide the system with contextual intelligence, while testing at an industrial scale on complex software systems would ensure scalability and computational robustness. Lastly, the creation of user-friendly tools that facilitate code optimization and thus make it easier for practical software development and for a wider usage would be the final step.

Future research may not only discover these gaps in the area of automated software maintenance and effectiveness but also build the existing structure for a more comprehensive, flexible, and superior-quality code optimization application, thus making it possible to develop such an application.

References

1. Andrew Walker, Tom Cerny, and Eungee Song, “Open-Source Tools and Benchmarks for Code-CloneDetection: Past, Present, and Future,” Applied Computing Review, VOL. 19, NO. 4 Dec. 2019. Solid State Technology Volume: 63 Issue: 6 Publication Year:2020 13796 Archives Available @ www.solidstatetechnology.us

2. Amir Shaikhha, Maximilian Schleich, Alexandru Ghita and Dan Olteanu, “Multi-layer Optimizations for End-to-End Data Analytics,” CGO 2020: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization Pages 145–157, February 2020, Doi: 10.1145/3368826.3377923.

3. S. Romano, C. Vendome, G. Scanniello and D. Poshyvanyk, “A Multi-Study Investigation into Dead Code,” IEEE Transactions on Software Engineering,vol. 46, no. 1, pp. 71-99, 1 Jan. 2020.

Doi: 10.1109/TSE.2018.2842781.

4. D. Tukaram and U. M. B, "Design and Development of Software Tool for Code Clone Search, Detection, and Analysis," 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, pp. 1002-1006, 2019 Doi: 10.1109/ICECA.2019.8821928.

5. F. Farmahinifarahani, V. Saini, D. Yang, H. Sajnani and C. V. Lopes, "On Precision of Code Clone Detection Tools," IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, pp. 84-94, 2019, Doi: 10.1109/SANER.2019.8668015.

6. Partial Redundancy Elimination using Lazy Code Motion by Sandeep Dasgupta, Tanmay Gangwani (2019).

7. V. Saini et al., "Towards Automating Precision Studies of Clone Detectors," IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 2019, pp. 49-59, Doi: 10.1109/ICSE.2019.00023.

8. Laxmi, N. Duhan and H. Jameel, "A Comparative Study of Clone Detection Approaches," 2019 6th International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, India, pp. 1025-1027, 2019.

9. R. Chouksey, C. Karfa and P. Bhaduri, "Translation Validation of Code Motion Transformations Involving Loops," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 7, pp. 1378-1382, July 2019, Doi: 10.1109/TCAD.2018.2846654.

10. M. Abdallah and M. Alrifaee, "Towards a new framework of program quality measurement based on programming language standards," vol. 7, p. 3, 2018, Doi: 2018-03-08 2018.

11. B. Alokush, M. Abdallah, M. Alrifaee, and M. Salah, "A Proposed Java Static Slicing Approach," Indonesian Journal of Electrical Engineering and Computer Science, vol. 11, pp. 308-317, 2018.

12. Al Abwaini, Nour & Aldaaje, Amal & Jaber, Tamara & Abdallah, Mohammad & Tamimi, Abdelfatah, "Using Program Slicing to Detect the Dead Code," Pp.230-233, 2018, Doi: 10.1109/CSIT.2018.8486334.

13. Moustafa Ebada, Ahmed Elkelesh, Sebastian Cammerer, and Stephan ten, "Scattered EXIT Charts for Finite Length LDPC Code Design," Brink Institute of Telecommunications, Pfaffenwaldring 47, University of Stuttgart, 70569 Stuttgart, Germany (2018).

14. N. Boonthep, K. Chamnongthai, and P. Phensadsaeng, "A FPGA-Based SIFT Architecture for

Motion Estimation in Video Coding," Global Wireless Summit (GWS), Chiang Rai, Thailand, pp. 383-388, 2018, Doi: 10.1109/GWS.2018.8686403. Solid State Technology Volume: 63 Issue: 6 Publication Year:2020 13797 Archives Available @ www.solidstatetechnology.us

15. S. Romano, "Dead Code," IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, pp. 737-742, 2018, Doi: 10.1109/ICSME.2018.00092.

16. S. Romano and G. Scanniello, "Exploring the Use of Rapid Type Analysis for Detecting the Dead Method Smell in Java Code," 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Prague, pp. 167-174, 2018, Doi: 10.1109/SEAA.2018.00035.

17. Runwal, Ashish N. and Akash D. Waghmare, "Code Clone Detection based on Logical Similarity: A Review," International Journal of Scientific Research in Science, Engineering and Technology 3 (2017), pp.148-151.

18. X. Wang, Y. Zhang, L. Zhao, and X. Chen, "Dead Code Detection Method Based on Program Slicing," International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), Nanjing, pp. 155-158, 2017, Doi: 10.1109/CyberC.2017.69.

19. A. Ngah, M. Munro and M. Abdallah, "An Overview of Regression Testing," Journal of Telecommunication, Electronic and Computer Engineering (JTEC), vol. 9, pp. 45-49, 2017.

20. A. Abdalla, M. Abdallah, and M. Salah, "A Brief PROGRAM ROBUSTNESS SURVEY," International Journal of Software Engineering & Applications, vol. 8, pp. 1-10, 2017.

21. M. Abdallah, B. Alokush, M. Alrefaee, M. Salah, R. Bader, and K. Awad, "JavaBST: Java backward slicing tool," 8th International Conference on Information Technology (ICIT), pp. 614-618, 2017.

22. J. Svajlenko and C. K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 596–600, Oct 2016.

23. M. Abdallah and H. Tamimi, "Clauser: Clause Slicing Tool for C Programs," International Journal of Software Engineering and Its Applications, vol. 10, pp. 49-56, 03/31 2016.

24. Gotarane, Prajakta and Sumedh Pundkar, "Smart Coding using New Code Optimization Techniques in Java to Reduce Runtime Overhead of Java Compiler," International Journal of Computer Applications 125 pp.11-16, 2015.

25. F. A. Fontana, M. Mangiacavalli, D. Pochiero, and M. Zanoni, "On experimenting with refactoring tools to remove code smells," The Scientific Workshop Proceedings of the XP2015, Helsinki, Finland, 2015.

26. X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Predicting Consistency Maintenance Requirement of Code Clones at Copy-and-Paste Time," IEEE Transactions on Software Engineering, vol. 40, no. 8, pp. 773-794, 2014.

27. M. Alpuente, D. Ballis, F. Frechina, and D. Romero, "Using conditional trace slicing for improving Maude programs," Science of Computer Programming, vol. 80, pp. 385-415, 2014, Doi: 2014/02/01.orem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.