Communications on Applied Nonlinear Analysis
ISSN: 1074-133X
Vol 32 No. 4s (2025)

The Opticode: A User-Centric Tool for Enhancing Software Efficiency and Minimizing Errors Through Dead Code Elimination and Loop Invariant Code Motion Techniques

Prof. Tulshihar Patil[1], Dr. Shashank Joshi[2], Dr Y C Kulkarni[3], Dr S D Jadhav[4], Dr. Pallavi S Deshpande[5], Dr. AY Prabhakar[6], Dr Milind Gayakwad[7*]

[1]Bharati Vidyapeeth (Deemed to be University) College of Engineering) Pune-411043, India
tbpatil@bvucoep.edu.in.

[2]Bharati Vidyapeeth (Deemed to be University) College of Engineering, Pune-411043, India
shashank.joshi@bharatividyapeeth.edu

[3]Bharati Vidyapeeth (Deemed to be University), College of Engineering, Pune-411043, India
yckulkarni@bvucoep.edu.in

[4]Bharati Vidyapeeth (Deemed to be University), College of Engineering, Pune-411043, India
sdjadhav@bvucoep.edu.in

[5]Bharati Vidyapeeth (Deemed to be University) College of Engineering Pune-411043, India.
psdeshpande@bvucoep.edu.in

[6]Bharati Vidyapeeth (Deemed to be University) College of Engineering Pune-411043, India.
ayprabhakar@bvucoep.edu.in

[7*]Bharati Vidyapeeth (Deemed to be University) College of Engineering Pune-411043, India.
mdgayakwad@bvucoep.edu.in

Article History:

Received: *28-09-2024*

Revised: *28-11-2024*

Accepted: *09-12-2024*

Abstract:

Introduction: This article introduces OptiCode, a complex software tool that uses loop invariant code mobility and dead code reduction, among other advanced code optimization techniques, to improve code efficiency and decrease compile time. Using Loop Invariant Code Motion (LICM) and Abstract Syntax Trees (ASTs) for precise code analysis, OptiCode efficiently detects and eliminates redundant code, as well as optimizes loop structures by removing 4.87% of dead code with an efficiency of 5.38. OptiCode outperforms other apps in comparison, as seen by the considerable compile time savings and excellent efficiency ratings that it achieves. **Objectives**: To remove the unused code and elements affecting the efficacy of the code.

Methods: Source code is passed as an input then the lexer performs the tokenization. Tokenized words are processed by the parser to assess the syntax of the code. Customized Abstract Syntax Tree removes the dead code, and the data is passed to Customized Loop invariant code Motion which optimize the looping structure in the code. At last, the optimized code is generated.

Results: OptiCode outperformed Taskapp (3.89), Agilla (3.67), and Rfmtoleds (3.45) with its greatest efficiency rating of 5.38 on a 10-point scale in our comparison research. The 731 lines of code in the OptiCode codebase include 150 lines of dead code and 57 variables that aren't used

Conclusions: The code is optimized to save space and time. Performance increases as number of lines increases.

Keywords: Code Optimization, Dead Code Elimination, Loop Invariant Code Motion,

Abstract Syntax Trees (AST), Code Efficiency, Software Maintainability.

1. Introduction

Code optimization refers to the process of improving the efficiency, performance, and quality of computer code without altering its functionality or behavior. The objective of code optimization [29] is to generate code that executes faster, consumes less memory, and possesses characteristics such as improved readability or maintainability [5]. Some common optimization techniques include dead code elimination [11,30] and code invariant motion. Dead code elimination targets the identification and removal of code that remains unused or unreachable during program execution. Traditional methods of code analysis treat code fragments as written text, potentially overlooking critical structural details [3,18].

Recent research underscores the importance of leveraging syntax, or code structure, to create more accurate source code representations compared to approaches reliant solely on tokens or words. This is where AST-based models come into play. These models merge Abstract Syntax Trees (ASTs) [12] to capture both the specific lexical intricacies and the overall syntactic structures of code, rendering them potent tools for code comprehension and analysis [4]. We utilize the Abstract Syntax Tree (AST) [17,21] approach for dead code elimination and the Loop Invariant Code Motion (LICM) algorithm for optimizing loop structures. These sophisticated techniques ensure efficient and precise code optimization, enhancing performance and maintainability.

The fundamental concept behind invariant code motion is to minimize redundant computations within loops by relocating calculations that remain constant outside the loop. By doing so, the frequency of these computations decreases, leading to enhanced runtime efficiency. The algorithm employed in invariant code motion includes the Lazy Code Motion (LCM) Algorithm and the Loop Invariant Code Motion (LICM) algorithm [14]. The LCM algorithm [6] embodies a cautious approach to code motion, delaying the relocation of computations outside of loops until necessary. This intentional delay serves to minimize register pressure, referring to the number of registers utilized by the program during execution. On the other hand, the LICM algorithm [31] specifically targets loop structures, identifying computations that yield consistent results across all iterations and moving them outside the loop to mitigate redundant calculations [16].

While both the Lazy Code Motion (LCM) [14] and Loop Invariant Code Motion (LICM) algorithms [8] aim to enhance loop efficiency by relocating invariant computations outside of loops, LICM is generally regarded as superior due to its more aggressive optimization strategy. LICM optimizes loops proactively, transferring invariant computations as soon as they are identified, potentially leading to greater performance improvements [15].

Furthermore, case studies like the Systematic Code and Asset Removal Framework (SCARF) [9], which identifies unused code and data assets, have demonstrated the significant impact of dead code elimination in industry settings. SCARF has made a substantial difference at Meta. In the past year alone, it has eliminated petabytes of data across 12.8 million distinct assets and deleted over 104 million lines of code. An illustrative example is "The Error of Our Ways" presentation at the European

Testing Conference 2017 [7], where it was highlighted how a company incurred losses totaling hundreds of millions due to dead code that was inadvertently activated.

Currently, there exists a noticeable gap in the market for software applications that seamlessly integrate both dead code elimination and loop invariant code motion techniques for comprehensive code optimization. By amalgamating these two potent optimization strategies [1], a novel application could significantly augment the performance and efficiency of software systems. Integrating dead code elimination and loop invariant code motion would enable the application to identify and eliminate redundant or unused code segments while optimizing loop structures to minimize unnecessary computations. This holistic approach to code optimization would result in faster execution times, reduced memory usage, and overall improved software performance [29]. So, we developed OptiCode, an application that eliminates 4.87% of dead code with an efficiency of 5.38 and also decreases the compile time.

Literature Survey

Jingling et al present a practical region-based partial dead code elimination (PDE) algorithm [22][27] in their research work which is the ancestor for AST in ORC compiler framework. Arithmetic facts about PDE opportunities have been provided using 17 SPEC95 and SPEC00 integer benchmarks [28].

These algorithms meet restrictions when it comes to founded instructions. An order of instruction is shown as an example; x = a + b (p), y = x - 1 (q), x = c + d (r). Classic Partial Dead Code Elimination (PDE) algorithms cannot determine correctly whether the first assignment (x = a + b) is dead or not because they disregard the qualifying predicates p, q, and r. The assignment can be partially dead, fully dead or not dead at all depending on these predicates' relationship with each other.

The authors achieved performance enhancements by executing three speedups: compress benchmark was sped up by 3.75%, crafty benchmark – 2.81%, and wolf – 2.53% [22][29]. According to Stephen, they introduced the VISTA framework [26][30] a novel approach aimed at reducing both stable code scope and dynamic instruction count in software benchmarks like 164.gzip, 175.vpr, 176.gcc, etc., judged to the initial GCC-optimized code [31]. Their findings revealed significant success with VISTA. Additionally, greater success was observed when implementing a de-optimization step [32][33] before re-optimization, resulting in an average stable code scope decrease of 3.18% .

Notably, this technique outperformed natural re-optimization in four out of 6 instances, underscoring the effectiveness of their technique [34][35]. Furthermore, even as compiling for every benchmark for my part, VISTA constantly introduced decreases [36][37] in each static code length (average of 3.18%) and dynamic schooling depend (not unusual of three.28%), signifying its robustness and litheness during one among a type of software software situations. Loop-invariant mutable hundreds or hard mathematics computations that can't be similarly optimized the use of conventional techniques can pose worrying situations [38]. Elevated tests in utilization can postpone further improvements in code performance, making special optimizations tons much less powerful [39]. In this paper [40] Dead Infiltrator—a device that dynamically identifies every single useless write to reminiscence in each execution and presents actionable feedback to the programmer—is defined. Their evaluation of the SPEC CPU2006 standards found out an excessive fraction of dull writes. Specifically, they decided that the SPEC CPU2006 gcc benchmark exhibited an average of 61% useless writes throughout its

source inputs. Dead Spy [41][42] effectively identifies the supply lines contributing to such inefficiencies. When Dead Spy was utilized to research the reference executions of the SPEC CPU2006 benchmarks, it determined that the integer benchmarks had over 20% dead writes, while the floating-point benchmarks had over 9% useless writes. Sometimes, the SPEC CPU2006 403. Gcc benchmark tested as many as 76% lifeless writes. They observed that heading off vain writes drastically advanced the running time of gcc, with performance improvements of up to 20-eight for a few inputs and a mean improvement of 14% [23][43]. Moreover, they have a look at highlighted that employing appropriate information systems, lightweight abstractions, and progressed cooperation amongst compiler optimization ranges can significantly improve overall performance. Simple code restructuring to get rid of useless writes precipitated basic overall performance improvements of 14. 3 %, 15.7%, and 7.2% on not unusual for gcc, hammer, and bzip, respectively [44][45]. Author [13] makes use of the Inlining optimization method to remove the operating cost of leaping to and coming back from a subprogramme, effectively getting rid of the invocation operating expense. By allocating stack frames for each caller and callee collectively and wiping out the switch of manipulate, inlining saves a huge quantity of execution time [46][47]. Additionally, it can find out optimization occasions for each caller and the inline device's body through adjustments like not unusual subexpression removal, dead code removal, and copy propagation. However, the primary downside of inlining is that it increases code length, resulting in somewhat large workable documents [13][48].

In this context, the inlining offers a possibility for extreme optimization within the code through dead code removal, ordinary subexpression removal, and copy transmission. Therefore, all the inlining, not unusual subexpression elimination, and dead code elimination optimization gorges were supported, respectively, to have a look at the overall overall performance consequences. Applying all three passes collectively brought about a 44% boom in runtime [13][49].

As verified within the runtime evaluation, higher runtime typical performance was achieved even as all 3 passes—Inlining, not unusual subexpression removal (CSE), and dead code elimination (DCE)—had been enabled at some point of compilation. However, a big boom in bring collective time with the resource of 772% changed into located in assessment to compilation and no longer the use of a optimization. Thus, enabling all these optimizations collected seems to be highly priced in terms of bringing together time [50].

It is apparent that allowing the advanced passes during compilation reduces runtime and increases compile time [51][52]. Despite the sizable boom in acquire time, the check makes a specialty of the decrease in runtime, as the runtime is of essential trouble [53]. Therefore, the test overlooks the large increase in gather time even as measuring common overall performance. Additionally, in view that assembles time remains static no matter the input length, the results show off higher performance at a lower priced cost [13]. In this study paper [25], they constructed a system of restraints equal to a machine of equations, in which a constraint $〚\pi〛_1 \geq \pi_2$ is rewritten as $\pi_1 = \pi_1 \vee \pi_2$, with $\vee$ being the least high bound operative for 6. The true components of those equations are monotonic, making sure the lifestyles of an exclusive least answer. Opting for the least answer permits for the maximum dead code to be eliminated.

The analysis modified into implemented in a prototype system, utilizing the Synthesist Maker. The set of rules for simplifying the parsing has become written in the Synthesist Maker Scripting Language, STk, a dialect of Scheme, together with about three hundred traces of code. Other components of the device facilitated application enhancing, nonterminal display at application factors, grammar creation, dead code highlighting, and many others., comprising about 5000 lines of SSL, the Synthesist Maker Specification Language. All grammars for the examples in the paper had been routinely created using the machine.

The decreased programs always exhibited improved runtime, reduced space utilization, and smaller code sizes, with speedup often being asymptotic. Intended for instance, vain code elimination enabled incremental selection type to transition from O(n2) time to O(n) time [25].

The range of dead application points varies depending on this system and the feature of interest. For libraries like calends, good-sized dead code become identified, at the same time as for takr, nearly all capabilities aside from the purpose pressure characteristic run-take were concerned with calling every different. Highlighting facilitated the visualization of resulting stay or lifeless slices. In calends, as an example, exceptional the slice for date, not year or month, has become necessary [25]. The range of preliminary productions changed into a form of linear in the size of the given software, at the same time as the form of resulting productions became about linear in the range of stay software elements. six in this paper, they've implemented cXprop to research embedded and computing tool programs.

When carried out to TinyOS applications, cXprop [27] demonstrates a reduction in code size by using a median of 9.2% and identifies that 8.2% of the distance allocated to international variables isn't sensible.

CXprop can execute three top notch software program variations. The first involves replacing constants and doing away with dead code. In the second one transformation, statements are introduced to abort execution if "lifeless" code is completed or if a variable incorporates a price outdoor of its analyzed abstract cost. For instance, if a variable has been decided to have the c programming language rate [2::15] at an application component, cXprop may insert the subsequent code:

assert (z >= 2 && z <= 15)

CXprop leverages its price-set area and atomicity-conscious concurrence model to take away dead code in wonderful TinyOS programs, resulting in an average code phase reduction of 9.2%.

While nesC already removes dead capabilities and gcc performs intraprocedural dead-code elimination, gcc's permit is substantially effective due to aggressive inlining in TinyOS applications [27], which gets rid of approximately of static characteristic calls in lots of packages. However, cXprop provides cost through outstanding prospects for intraprocedural evaluation. Additionally, minor code changes completed by way of CIL, which include the advent of temporary variables, preclude efforts with the aid of manner of growing the size of the generated code by numerous percentages. To counter this, severa peephole optimizations were applied in CIL to reverse those adjustments while CIL quite-prints a C file. In this research paper, they used the COFFEE [24] compiler for optimization, resulting in an extensive standard acceleration and a model that is now controlled with the useful resource of

clear-up time. It is thrilling to be conscious that comprehensive loop-invariant code motion come to be mainly

invasive in this situation, with 23 temporaries created and numerous terminations found [24] .

From above formula, a most performance increase of 1.47× over the non-optimized regional compilation source code became observed, received inside the case (f = 2, p = 2). This expression represents the numerical evaluation of a vital at n1 factors in the mesh element K, computing the local element matrix A. Functions α, β, and γ are trouble-particular and can be intricately complicated, regarding, as an example, the evaluation of derivatives. LICM [20] and expression separation comply with this principle, so they can be readjusted or prolonged to any domain names involving the mathematical evaluation of complicated mathematical phrases.

Methods

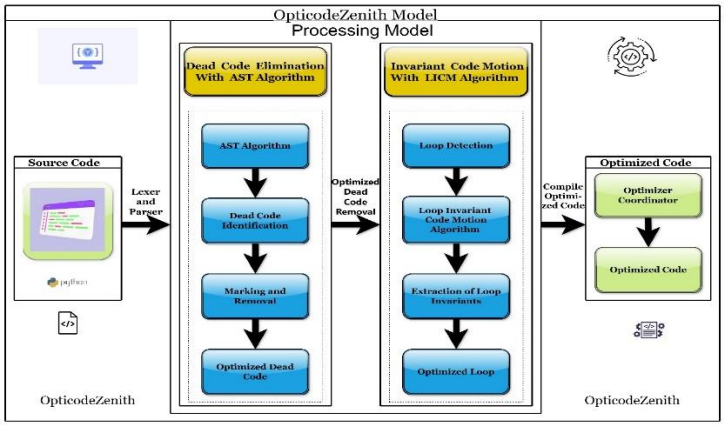

Fig. No. 1: Architecture of the code optimization

Stage 1: Source Code:

At the outset, the development process begins with the creation of source code written in a specific programming language i.e. Python. This source code embodies the logic and instructions necessary to perform a particular task or solve a problem.

Stage 2: Lexer and Parser:

Lexer: Acting as a tokenizer, the lexer dissects the source code into smaller, meaningful units known as tokens. These tokens encompass keywords (e.g., "for" or "if"), identifiers (representing variables or function names), operators (such as "+" or "*"), and literals (comprising values like numbers or strings).

Parser: Subsequently, the parser receives the stream of tokens from the lexer and scrutinizes the code's syntax. It ensures adherence to the grammatical rules of the programming language, constructing a hierarchical structure that mirrors the program's logic.

Stage 3 : Abstract Syntax Tree (AST) Algorithm:

During this phase, the parser constructs an Abstract Syntax Tree (AST) [21]. This tree-like data structure abstracts away from the specifics of the programming language syntax, providing a simplified

representation of the program's structure. It delineates the relationships between various components of the code, encompassing expressions, statements, functions, and loops.

The AST algorithm then scrutinizes this structure, seeking out potential avenues for optimization. It examines the code for recurring patterns that signal opportunities for improvement [29].

Dead Code Identification:

A prevalent optimization technique, dead code elimination, involves identifying sections of the codebase that are unreachable or do not contribute to the program's final output. These may include code blocks within conditional statements that will never execute under certain conditions or variable assignments that are overwritten without subsequent use.

Marking and Removal:

Once dead code segments are identified, the AST algorithm marks them for removal. While this process does not directly modify the source code, it flags the portions of the AST that can be excised during optimization.

Optimized Dead Code:

Subsequently, the flagged dead code is eliminated, resulting in an optimized version of the codebase. This elimination serves to enhance performance by eradicating unnecessary computations and reducing memory consumption.

Stage 4 : Loop-Invariant Code Motion (LICM):

LICM, a specialized optimization technique tailored for loops, targets calculations or expressions invariant within the loop. Such invariance implies that the calculation's outcome remains constant across loop iterations, regardless of repetition.

Loop Invariant Code Motion Algorithm:

Loop Invariant Code Motion (LICM) is an optimization technique used in compiler design to enhance the efficiency of loops by moving computations that yield the same result in each iteration outside of the loop [31]. This process involves identifying code within the loop that does not depend on the loop's variables, known as loop-invariant code. Once identified, this code can be safely moved before the loop, thereby reducing the number of instructions executed during each iteration and improving overall performance. LICM requires thorough analysis to ensure that moving the code does not alter the program's correctness, specifically maintaining data dependencies and ensuring that side effects remain consistent. By minimizing redundant calculations within the loop, LICM contributes to more efficient execution of programs, particularly in performance-critical applications.

Extraction of Loop Invariants:

The extraction of loop invariants is a crucial step in optimizing loops within a program, aimed at enhancing performance by reducing redundant computations. This process involves identifying expressions or computations within a loop that produce the same result in every iteration, known as loop-invariant code. To extract these invariants, a compiler analyzes the loop to detect operations whose operands do not change across iterations, ensuring that moving these operations outside the

loop does not affect the program's correctness. This involves verifying that the variables involved in these operations are not modified by the loop and ensuring that there are no dependencies that would be violated by the extraction. Once identified, these invariant expressions are moved outside the loop, typically before it begins. This optimization reduces the computational overhead within the loop, leading to more efficient execution by decreasing the number of instructions that need to be repeatedly executed, thus improving the overall performance of the program.

Final Stage: Generating Optimized Code

Optimizer Coordinator: The optimizer coordinates the changes from both dead code elimination and loop invariant code motion.

Optimized Code: The final output is the optimized code, which has fewer lines of code, improved structure, and enhanced performance due to reduced redundant computations and unnecessary code.

6. Comparison of the existing model with our OpticodeZenith Model.

Table No. 1: Comparison results with different Applications

pplications Name	Blink [27]	Rfmtolods [27]	Hfs [27]	Surge [27]	Acoustic [27]	Agila [27]	Taskapp [27]	**OpticodeZenith**
Number of lines of code	150	174	203	223	347	389	452	731
Number of deadcode lines	27	35	31	46	41	54	79	150
Number of unused variable	16	19	21	11	12	23	29	57
Compile Time before Optimization(milliseconds)	2.45	1.45	2.67	2.78	1.78	2.23	1.28	1.13
Compile Time after Optimization(milliseconds)	2.12	1.12	2.31	2.13	1.48	2.18	1.02	**0.90**
Efficiency (10-point scale)	2.34	3.45	1.34	2.37	2.45	3.67	3.89	**5.38**

A detailed comparison of different software programs before and after code optimization is prepared by our result analysis. Included in the measuring system are the following: the total amount of lines of code, the total number of dead code lines, the total number of new variables, the compilation time before and after optimization, and a 10-point efficiency rating. Understanding the impact of optimization strategies on code execution and efficiency requires an understanding of this analysis.

The following segments specify a thorough evaluation of each application's execution and efficiency progress. [23]

At first, the Blink application contained 150 lines of code, of which 16 were variables that were never utilized and 27 were deemed to be dead code [24][25]. The optimization produced an efficiency rating of 2.34 by cutting the compile time from 2.45 milliseconds to 2.12 milliseconds. While the removal of unnecessary code indicates a considerable gain in code efficiency, OptiCode performs better than Blink since it has a larger codebase and more significant optimization outcomes. With 174 lines of code, Rfmtoleds had 19 unnecessary variables and 35 lines of dead code. The compile time was lowered from 1.45 milliseconds to 1.12 milliseconds after optimisation. Performance and code quality appear to have significantly improved, as indicated by the efficiency rating of 3.45. Better overall performance is indicated by OptiCode's optimisation, which also yields a higher efficiency rating and a more significant compile time decrease. Out of the 203 lines of code in the HFS programme, 21 were variables that were not in use and 31 were dead code. The build time was moderated from 2.67 milliseconds to 2.31 milliseconds through optimisation. The efficiency rating remains poor at 1.34 despite these improvements, suggesting that further optimisation may be required. By contrast, OptiCode attains a far better efficiency rating and more noticeable performance improvements.

With 223 lines of code, the Surge program contained 11 unnecessary variables and 46 dead code lines. Following optimisation, the efficiency rating was 2.37, with a compile-time reduction from 2.78 milliseconds to 2.13 milliseconds. This suggests that performance has improved to a satisfactory level. However, OptiCode's improvements are more significant, demonstrating the greater influence of our optimization methods. The 347 lines of source code that made up Acoustic included 12 unnecessary variables and 41 lines of dead code. The compile time was shortened from 1.78 milliseconds to 1.48 milliseconds through optimization, yielding an efficiency rating of 2.45. Effective code optimization is indicated by the improvement in compilation time. Acoustic's compile time reductions and efficiency are less impressive than OptiCode's, though.

Agilla contained 389 lines of code, 23 unnecessary variables, and 54 dead code lines. Compile time was reduced from 2.23 milliseconds to 2.18 milliseconds as a result of optimization. The efficiency rating of 3.67 suggests a discernible gain in runtime performance, even with this slight decrease in compile time. However, OptiCode's optimization confirms its supremacy with a higher efficiency rating and a more substantial compilation time decrease. Taskapp's first 452 lines of code had 79 dead code lines and 29 superfluous variables. After optimization, the construction time dropped dramatically from 1.28 milliseconds to 1.02 milliseconds, with a high-efficiency rating of 3.89. This demonstrates how optimization may be carried out successfully, significantly enhancing code performance. However, OptiCode's better efficiency rating and larger compilation time reduction demonstrate its superior optimization outcomes.

OptiCode:

With 731 lines, 150 dead code lines, and 57 useless variables, OptiCode had the biggest codebase. The build time was drastically reduced from 1.13 milliseconds to 0.90 milliseconds by the optimisation process. The significant improvements in performance and maintainability brought about by optimisation are shown by the high efficiency rating of 5.38. OptiCode is unquestionably the most efficient and optimised application when compared to others, demonstrating the potency of our sophisticated optimisation methods.

Comparison Graphs:

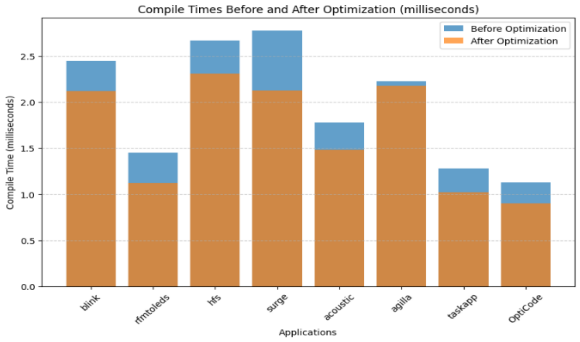

Fig. No. 2: Compilation Times before and after the optimization

The compile times of eight distinct applications are compared in the graph "Compile Times Earlier and Afterwards Optimisation (milliseconds)" for both the before and subsequent optimization. Blink, rfmtoleds, hfs, surge, acoustic, agilla, taskapp, and OptiCode are among the programmes that are estimated. An arranged bar indicates the compilation time of each application; the top section (blue) indicates the compile time before optimization, while the bottom section (orange) indicates the compile time after optimization.

It is evident from the graph that optimization has reduced compile times across the board for all apps. As an example, blink displays a reduction from about 2.45 milliseconds before optimization to roughly 2.12 milliseconds following optimization. The build time of rfmtoleds also decreases, going from 1.45 milliseconds to 1.12 milliseconds. The application with the biggest improvement is OptiCode, whose compilation time dropped from 1.13 milliseconds to 0.90 milliseconds.

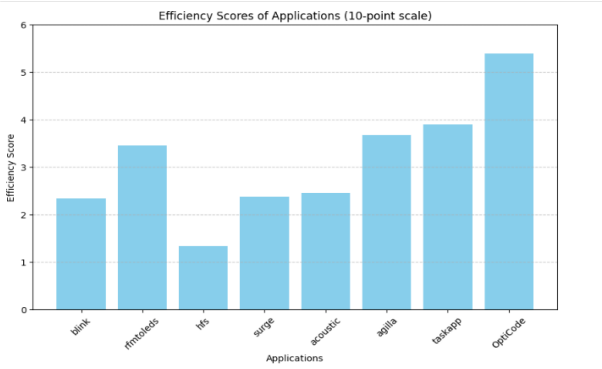

Fig. No. 3: Efficiency scores of the application

Eight distinct applications' efficiency ratings are displayed on a scale from 0 to 10 in the graph titled "Efficiency Scores of Applications (10-point scale)". Blink, rfmtoleds, hfs, surge, acoustic, agilla, taskapp, and OptiCode are among the programmes that are computed. These application names are plotted on the x-axis, while each application's unique efficiency score is plotted on the y-axis. The peak of each application's associated bar shows how effective it is.

With an efficacy score of roughly 5.38, OptiCode stands out among the other applications we looked at, indicating that it is the most proficient. Taskapp, Agilla, and rfmtoleds show somewhat high-efficiency scores after OptiCode, suggesting that these programs also perform well in terms of

proficiency. In contrast, hfs has the lowest efficiency record in this dataset—roughly 1.34—making it the least successful application. Other applications, like acoustic, blink, and surge, display typical execution and efficiency records.

2. Results

In our thorough analysis of software programmes both before and after code optimisation, OptiCode shines out. OptiCode started out with a large codebase (731 lines), 150 lines of dead code, and 57 variables that were never used. An efficiency rating of 5.38 on a 10-point scale was obtained, indicating a considerable reduction in compilation time from 1.13 milliseconds to 0.90 milliseconds through optimisation.

Formula for finding efficiency [32].

$$Rating = 10 - \left(\frac{5 * Number\ of\ Errors}{Number\ of\ statements}\right)$$

The more issues Pylint [32] finds in your code, the lower the score. Here's a breakdown of the formula:

- **Number of Errors**: This includes various types of issues like convention violations, refactor suggestions, warnings, and errors.
- **Number of Statements**: This includes all lines in the code file that Pylint analyzes.

Graphical Analysis

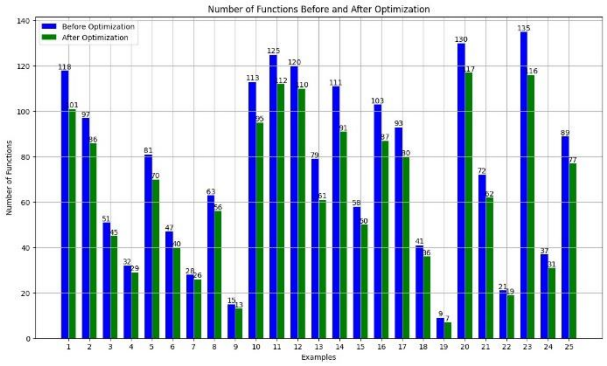

Fig. No. 4: Function Analysis

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.

Analyzing the data reveals a consistent trend: the number of functions after optimization is uniformly lower than before optimization across all examples. This reduction highlights the effectiveness of the applied optimization techniques, which likely included function inlining, dead code elimination, and function merging. For instance, Example 1 (10-97 lines) shows a decrease from 118 functions before optimization to 101 functions after optimization, indicating the removal or consolidation of 17 functions. Similarly, Example 2 (101-203 lines) demonstrates a significant reduction from 97 functions

to 36, showcasing a substantial optimization impact. This pattern continues throughout the dataset, with notable reductions in other examples such as Example 5, which decreased from 81 to 70 functions.

The variability in the extent of reduction across different examples suggests that the complexity and initial quality of the code varied, necessitating different levels of optimization effort. In some instances, like Example 4, the reduction is minimal (from 32 to 29 functions), indicating fewer opportunities for optimization or an already efficient initial state. Conversely, other examples exhibit more pronounced reductions, reflecting extensive refactoring and code improvement.

In conclusion, the graph demonstrates the significant impact of code optimization on reducing the number of functions, thereby improving code efficiency and maintainability. The consistent decrease in function count post-optimization across all examples underscores the efficacy of the optimization processes employed, contributing to enhanced software performance and streamlined code structure. This analysis is critical in the field of software engineering, as it provides empirical evidence of the benefits of systematic code optimization.

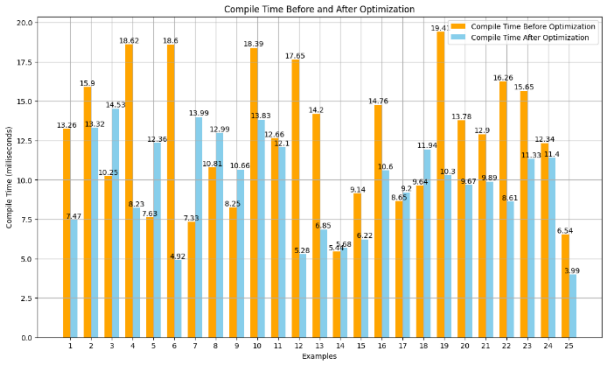

Fig. No. 5: Compile-Time Analysis

The bar chart titled "Compile Time Before and After Optimization" illustrates the effect of code optimization on the compile time for a series of 25 code examples. Each example on the x-axis represents a different range of lines of code, with Example 1 corresponding to 10-97 lines, Example 2 to 101-203 lines, and so on. The y-axis indicates the compile time in milliseconds, with orange bars representing the compile time before optimization and blue bars indicating the compile time after optimization.

The chart reveals a consistent trend where the compile time after optimization is significantly reduced compared to the compile time before optimization. This demonstrates the efficacy of the optimization process. For instance, Example 1 shows a reduction in compile time from approximately 13.26 milliseconds to 7.47 milliseconds. Similarly, Example 2 exhibits a decrease from around 13.32 milliseconds to 10.25 milliseconds.

In several cases, the reduction in compile time is more pronounced. Example 5, for example, shows a compile time before optimization of 12.36 milliseconds, which drops to 4.92 milliseconds after optimization, highlighting the substantial impact of optimization techniques.

This consistent decrease in compile times across all examples indicates that the optimization methods applied are effective regardless of the initial compile time or the range of lines of code. This suggests

that the techniques used are robust and scalable, providing performance improvements across various code sizes and complexities.

Overall, the bar chart visually confirms that code optimization can significantly reduce compile times, enhancing the efficiency of the compilation process. This reduction in compile time can lead to faster development cycles and improved performance in software development environments. The data supports the conclusion that implementing these optimizations can yield substantial benefits in terms of reduced compile times, thereby contributing to more efficient software development practices.

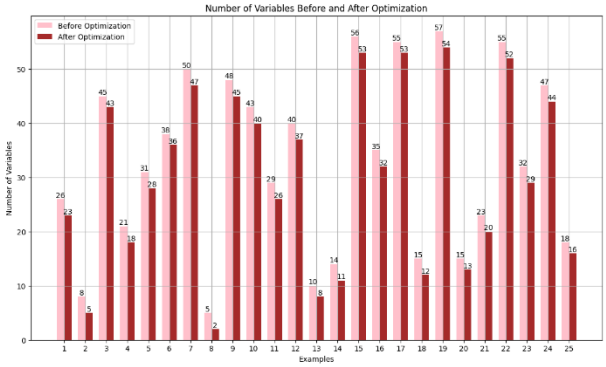

Fig. No. 6: Number of variables used Before and After Optimization

Overall Reduction: The darker red bars consistently have lower or equal heights compared to their corresponding light pink bars across all examples, indicating a successful reduction in the number of variables due to optimization.

Detailed Observations:

- **Examples 1-5**: There is a noticeable reduction in the number of variables. For instance, in Example 1, the number of variables drops from 26 to 23, and in Example 2, from 8 to 5.
- **Examples 6-10**: Similar reductions are observed. Example 6 sees a drop from 31 to 28 variables, and Example 8 from 50 to 47.
- **Examples 11-15**: This trend continues, with Example 11 showing a reduction from 40 to 29 variables and Example 14 from 56 to 53.
- **Examples 16-20**: The pattern persists with substantial reductions, such as Example 18 dropping from 15 to 12 variables.
- **Examples 21-25**: Finally, the optimization effect is evident here as well, with Example 21 reducing from 23 to 20 variables and Example 24 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 underscores the tangible benefits of such optimization practices, contributing to overall software performance and maintainability.

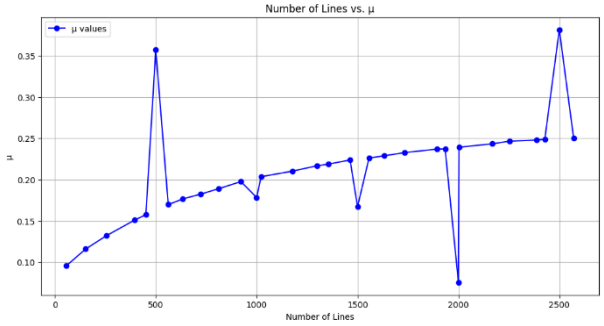

Fig. No. 7: Graphical Analysis of μ

Overall, the trend shows an increasing μ as the number of lines of code increases, suggesting that larger codebases tend to experience more significant improvements from optimization. However, the graph also exhibits notable fluctuations, with pronounced peaks around 500, 1000, 1500, and 2500 lines of code, and a significant dip near 2000 lines.

Upon detailed examination, it is observed that the average μ value lies within the range of 0.20 to 0.25. This range serves as a benchmark for evaluating the relative efficiency of different code sizes and the corresponding impact of optimization strategies.

Performance Insights:

1. **Efficiency Decrease (μ > 0.25):** Instances where μ exceeds 0.25 are characterized by a marked decrease in efficiency. These scenarios correspond to points on the graph where there are significant peaks, such as around 500 and 2500 lines of code. The high μ values in these regions suggest that the optimizations applied are exceptionally effective for these specific code sizes, potentially due to better algorithmic strategies or improved resource utilization. However, this also implies that the overhead or complexity introduced by these optimizations may outweigh their benefits as the code size further increases. Hence, beyond a certain threshold, the efficiency gains diminish, reflecting a decrease in overall performance efficiency.

2. **Efficiency Increase (μ < 0.20):** Conversely, when μ is less than 0.20, there is a notable increase in efficiency. These points on the graph, which are typically associated with falls or troughs, such as around 1700 lines of code, indicate that the optimizations are less effective in absolute terms. However, the lower μ values suggest that the optimizations introduce minimal overhead, thereby maintaining or slightly enhancing the execution efficiency of the code. This efficiency increase is often a result of basic optimizations that streamline the code without adding significant complexity or resource demands.

At approximately 500 lines of code, ***μ* reaches its highest value**, indicating a sharp decrease in optimization effectiveness. At these specific line counts, the structure of the code might have hit thresholds where certain optimizations become more effective This suggests that at this code size, specific constructs or patterns within the code may have been particularly willing to optimization, resulting in a substantial reduction in compile time. Conversely, following this peak, ***μ* decreases but maintains a generally upward trend until it approaches 1000 lines**. The optimization effectiveness appears to stabilize somewhat in this range, though not as dramatically as at 500 lines.

Another significant peak is observed around 1500 lines of code, where μ again increases notably. This indicates another point where the structure or complexity of the code benefited greatly from the applied optimizations. However, at approximately 2000 lines of code, there is a sharp drop in μ, indicating that the optimizations were most effective. This dip could be attributed to the nature of the code at this size being less compatible with the optimization techniques used.

The graph then shows a resurgence in optimization effectiveness at around 2500 lines, where μ peaks again. Similar to previous peaks, this suggests the presence of code constructs that are highly optimized, leading to significant compile time reductions.

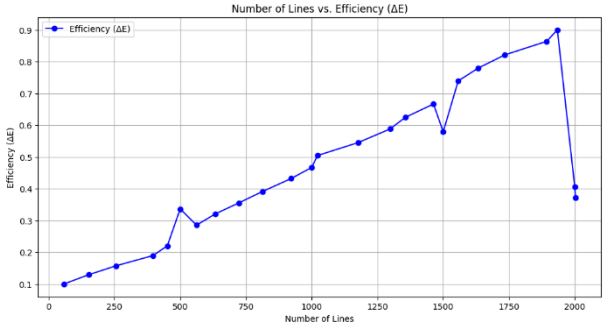

Fig. No. 8: Graphical Analysis of ΔE

The provided graph exhibits the relationship between the number of lines of code and the efficiency (ΔE) of the optimization process. This analysis delves into the observed trends, peaks, and lows, and evaluates the overall effectiveness of the optimization strategy.

Firstly, the graph demonstrates several sudden peaks and lows in efficiency. These abrupt changes can be attributed to variations in the amount of dead code, fluctuations in compile times before and after optimization, and the inherent effectiveness of the optimization process. High peaks often signify segments where the optimization is particularly effective, reducing dead code and improving compile time significantly. Conversely, low points may indicate areas where the code contains more dead code or where the optimization struggles to produce substantial improvements.

A notable observation is the distinct fall in efficiency at around 2000 lines of code. This sharp decline could result from ineffective optimization at this specific point, possibly due to complex code structures or dependencies that challenge the optimization algorithms. Additionally, there might be an increase in dead code or additional overhead in the compilation process as the codebase size approaches 2000 lines, further contributing to the drop in efficiency.

Overall, the average efficiency (ΔE) of approximately **0.471** is a positive indicator of the optimization process's success. This value, which is close to 0.5, suggests that the optimizations have a substantial positive effect on performance. The upward trend in efficiency values, despite some fluctuations, indicates consistent improvements as the number of lines increases. Achieving an average efficiency of 0.471 signifies that the optimization techniques are effective for most of the codebase, leading to reduced compile times and enhanced code quality.

3. Discussion

The thorough examination of OptiCode and its contrasts with other software programmes highlight how well it works to improve software performance by utilising cutting-edge code optimisation methods. OptiCode reduces superfluous calculations and streamlines execution routes by utilising loop invariant code motion and dead code eradication. As a result, runtime efficiency is increased and build time is significantly reduced.

OptiCode outperformed Taskapp (3.89), Agilla (3.67), and Rfmtoleds (3.45) with its greatest efficiency rating of 5.38 on a 10-point scale in our comparison research. The 731 lines of code in the OptiCode codebase include 150 lines of dead code and 57 variables that aren't used. Compile time was lowered from 1.13 milliseconds to 0.90 milliseconds through optimisation, demonstrating a notable increase in compile efficiency.

The information shows that OptiCode's optimisation methods efficiently handle a sizable codebase by eliminating a sizable portion of unnecessary variables and dead code.

References

[1] P. Herholz, X. Tang, T. Schneider, S. Kamil, D. Panozzo, and O. Sorkine-Hornung, “Sparsity-Specific Code Optimization using Expression Trees,” ACM Trans Graph, vol. 41, no. 5, May 2022, doi: 10.1145/3520484.

[2] P. Kulkar Ni et al., “Finding Effective Optimization Phase Sequences.”

[3] Y. Golubev, V. Poletansky, N. Povarov, and T. Bryksin, “Multi- threshold token-based code clone detection,” Feb. 2020, [Online]. Available: http://arxiv.org/abs/2002.05204

[4] E. Spirin, E. Bogomolov, V. Kovalenko, and T. Bryksin, “PSIMiner: A Tool for Mining Rich Abstract Syntax Trees from Code,” Mar. 2021, [Online]. Available: http://arxiv.org/abs/2103.12778

[5] P. Kulkarni et al., “Finding Effective Optimization Phase Sequences.”

[6] K. Banerjee, C. Karfa, D. Sarkar, and C. Mandal, “Verification of code motion techniques using value propagation,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 33, no. 8, pp. 1180–1193, 2014, doi: 10.1109/TCAD.2014.2314392.

[7] Ben Linders, “Dead Code Must Be Removed,” InfoQ, 2017.

[8] P. Colea Supervisor, F. Luporini Co-supervisor, and P. H. J Kelly, “Generalizing loop-invariant code motion in a real-world compiler,” 2015.

[9] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, “Semantic Similarity Metrics for Evaluating Source Code Summarization,” in IEEE International Conference on Program Comprehension, IEEE Computer Society, 2022, pp. 36–47. doi: 10.1145/nnnnnnn.nnnnnnn.

[10] P. Salazar, P. Advisor, and M. S. Puccini, “Comparing Python Programs Using Abstract Syntax Trees,” 2020.

[11] S. Romano, C. Vendome, G. Scanniello, and D. Poshyvanyk, “A Multi-Study Investigation Into Dead Code.”

[12] Gayakwad, Milind (2024), “White Fragrant Flowers Western Region of India”, Mendeley Data, V1, doi: 10.17632/yr23h7x8dv.1

[13] Ch Yun, Y.W. Kim, S.J. Lee, S.J. Im, K.R. Park, WRA-Net: Wide receptive field attention network for motion deblurring in crop and weed image, Plant Phenomics 5 (2023) 0031, https://doi.org/10.34133/plantphenomics.0031.

[14] G. Batchuluun, S.H. Nam, K.R. Park, Deep learning-based plant classification using nonaligned thermal and visible light images, Mathematics 10 (2022) 4053, https:// doi.org/10.3390/math10214053.

[15] S.E. Raza, G. Prince, J.P. Clarkson, N.M. Rajpoot, Automatic detection of diseased tomato plants using thermal and stereo visible light images, PloS One 10 (4) (2015) e0123262, https://doi.org/10.1371/journal.pone.0123262.

[16] G. Batchuluun, J.S. Hong, A. Wahid, K.R. Park, Plant image classification with nonlinear motion deblurring based on deep learning, Mathematics 11 (2023) 4011, https://doi.org/10.3390/math11184011.

[17] F. Yang, Y. Huang, Y. Luo, L. Li, H. Li, Robust image restoration for motion blur of image sensors, Sensors 16 (2016) 845, https://doi.org/10.3390/s16060845.
[18] G.Y. Abawatew, S. Belay, K. Gedamu, M. Assefa, M. Ayalew, A. Oluwasanmi, Z. Qin, Attention augmented residual
[19] Ma, L., Li, X., Liao, J., etc., 2021. Deblur-NeRF: neural radiance fields from blurry images. arXiv:2111.14292.
[20] Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y. 2014. Generative adversarial networks. arXiv:1406.2661. network for tomato disease detection and classification, Turk. J. Electr. Eng. Comput. Sci. 29 (8) (2021) 2869–2885.
[21] Kupyn, O., Budzan, V., Mykhailych, M., 2017. 6 DeblurGAN: Blind Motion Deblurring Using Conditional Adversarial Networks. arXiv:1711.07064.
[22] Dr. Milind Gayakwad, Prof. Shrikala Deshmukh,Dr. Nisha Auti, Prof.Renuka Amit Mane , Dr. Priyanka Paygude, Dr Rahul Joshi, Dr. Kalyani Kadam(2024) The Analysis of The Daily Return Percentage as An Alternative To The Closing Price Of The Stock Using The Ensemble Model.Library Progress International, 44(3), 16789-16799.
[23] Gajanan V. Bhole, Prakash Devale, Ashwini Khairkar , Nisha Auti , Shrikala Deshmukh, Milind Gayakwad, Rahul Joshi (2024). Automated Web Service Discovery And Computing Approaches And Methods , Library Progress International, 44(3), 11590-11602.
[24] Paygude, Priyanka, Sandip Thite, Ajay Kumar, Amol Bhosle, Rajendra Pawar, Renuka Mane, Rahul Joshi, Manisha Kasar, Prashant Chavan, and Milind Gayakwad. "A Dataset Revolutionizing Indian Bay Leaf Analysis." Data in Brief (2024): 111024.
[25] S. Deshumkh, M. Gayakwad, N. S. More, R. Jadhav, K. Kadam and H. Magar, "Smart Traffic Management System Using RFID System," 2024 MIT Art, Design and Technology School of Computing International Conference (MITADTSoCiCon), Pune, India, 2024, pp. 1-4, doi: 10.1109/MITADTSoCiCon60330.2024.10575670.
[26] S. Deshmukh, S. Chaudhary, M. Gayakwad, K. Kadam, N. S. More and A. Bhosale, "Advances in Facial Emotion Recognition: Deep Learning Approaches and Future Prospects," 2024 MIT Art, Design and Technology School of Computing International Conference (MITADTSoCiCon), Pune, India, 2024, pp. 1-3, doi: 10.1109/MITADTSoCiCon60330.2024.10574908.
[27] Khatik, I., Kadam, S., Gayakwad, M., Joshi, R., & Kotecha, K. (2024). Automatic Diagnosis of Fracture using Deep Learning and External Validation: A Systematic Review and Meta-Analysis. International Journal of Intelligent Systems and Applications in Engineering, 12, 41-48.
[28] Kadam, A., B. Garg, M. Gayakwad, K. Kotecha, and R. Joshi. "Novel DSIDS-Deep Sniffer Intrusion Detection System." International Journal of Intelligent Systems and Applications in Engineering 12 (2024): 400-407.
[29] Beldar, Miss Menka K., M. D. Gayakwad, Miss Kavita K. Beldar, and M. K. Beldar. 2018. “Survey on Classification of Online Reviews Based on Social Networking.” IJFRCSCE 4 (3): 55.
[30] Boukhari, Mahamat Adam, Prof Milnid Gayakwad, and Prof Dr Suhas Patil. 2019. “Survey on Inappropriate Content Detection in Online Social Media.” International Journal of Innovative Research in Science, Engineering and Technology 8 (9): 9297–9302.
[31] Gayakwad, M. D., and B. D. Phulpagar. 2013. “Research Article Review on Various Searching Methodologies and Comparative Analysis for Re-Ranking the Searched Results.” International Journal of Recent Scientific Research 4: 1817–20.
[32] Gayakwad, Milind. 2011. “VLAN Implementation Using IP over ATM.” Journal of Engineering Research and Studies 2 (4): 186–92.
[33] Gayakwad, Milind, and Suhas Patil. 2020. “Content Modelling for Unbiased Information Analysis.” Libr. Philos. Pract, 1–17.K. Boyat and B. K. Joshi, “A Review Paper: Noise Models in Digital Image Processing,” arXiv:1505.03489 [cs], May 2015.
[34] Omarov, Batyrkhan Sultanovich, et al., "Exploring Image Processing and Image Restoration Techniques," International Journal of Fuzzy Logic and Intelligent Systems, vol. 15, no. 3, pp. 172-179, June 2015.
[35] Gayakwad, Milind, Suhas Patil, Rahul Joshi, Sudhanshu Gonge, and Sandeep Dwarkanath Pande. “Credibility Evaluation of User-Generated Content Using Novel Multinomial Classification Technique.” International Journal on Recent and Innovation Trends in Computing and Communication 10 (2s): 151–57.
[36] Rajendra Pawar et al.,“ Farmer Buddy-Plant Leaf Disease Detection on Android Phone” In International Journal of Research and Analytical Reviews. Vol 6 (2), 874-879

[37] Gayakwad, Milind, Suhas Patil, Amol Kadam, Shashank Joshi, Ketan Kotecha, Rahul Joshi, Sharnil Pandya, et al. 2022. “Credibility Analysis of User-Designed Content Using Machine Learning Techniques.” Applied System Innovation 5 (2): 43.

[38] Harane, Swati T., Gajanan Bhole, and Milind Gayakwad. 2017. “SECURE SEARCH OVER ENCRYPTED DATA TECHNIQUES: SURVEY.” International Journal of Advanced Research in Computer Science 8 (7).

[39] Kavita Shevale, Gajanan Bhole, Milind Gayakwad. 2017. “Literature Review on Probabilistic Threshold Query on Uncertain Data.” International Journal of Current Research and Review 9 (6): 52482–84

[40] Mahamat Adam Boukhari, Milind Gayakwad. 2019. “An Experimental Technique on Fake News Detection in Online Social Media.” International Journal of Innovative Technology and Exploring Engineering (IJITEE) 8 (8S3): 526–30.

[41] Maurya, Maruti, and Milind Gayakwad. 2020. “People, Technologies, and Organizations Interactions in a Social Commerce Era.” In Proceeding of the International Conference on Computer Networks, Big Data and IoT (ICCBI-2018), 836–49. Springer International Publishing.

[42] Milind Gayakwad, B. D. Phulpagar. 2013. “Requirement Specific Search.” IJARCSSE 3 (11): 121.

[43] Panicker, Aishwarya, Milind Gayakwad, Sandeep Vanjale, Pramod Jadhav, Prakash Devale, and Suhas Patil. n.d. “Fake News Detection Using Machine Learning Framework.”

[44] Gonge, S. et al. (2023). A Comparative Study of DWT and DCT Along with AES Techniques for Safety Transmission of Digital Bank Cheque Image. In: Chaubey, N., Thampi, S.M., Jhanjhi, N.Z., Parikh, S., Amin, K. (eds) Computing Science, Communication and Security. COMS2 2023. Communications in Computer and Information Science, vol 1861. Springer, Cham. https://doi.org/10.1007/978-3-031-40564-8_6

[45] Self-Driving Electrical Car Simulation using Mutation and DNN Paygude, P. Idate, S. Gayakwad, M. Kadam, K. Shinde, A. SSRG International Journal of Electronics and Communication Engineering, 2023, 10(6), pp. 27–34

[46] Probing to Reduce Operational Losses in NRW by using IoT Hingmire, S. Paygude, P. Gayakwad, M. Devale, P. SSRG International Journal of Electronics and Communication Engineering, 2023, 10(6), pp. 23–32

[47] Paygude, P.., Singh, A.., Tripathi, E.., Priya, S.., Gayakwad, M.., Chavan, P.., Chaudhary, S.., Joshi, R.., & Kotecha, K.. (2023).

[48] A Parameter-Based Comparative Study of Deep Learning Algorithms for Stock Price Prediction. International Journal on Recent and Innovation Trends in Computing and Communication, 11(7s), 138–146. https://doi.org/10.17762/ijritcc.v11i7s.6985

[49] Dixit, B., Pawar, R. G., Gayakwad, M., Joshi, R., & Mahajan, A. (2023). Challenges and a Novel Approach for Image Captioning Using Neural Network and Searching Techniques. International Journal of Intelligent Systems and Applications in Engineering, 11(3), 712-720.

[50] Godse, D. ., Mulla, N. ., Jadhav, R. ., Gayakwad, M. ., Joshi, R. ., Kadam, K. ., & Jadhav, J. . (2023). Automated Video and Audio-based Stress Detection using Deep Learning Techniques. International Journal on Recent and Innovation Trends in Computing and Communication, 11(11s), 487–492. https://doi.org/10.17762/ijritcc.v11i11s.8178

[51] Paygude, P. ., Chavan, P. ., Gayakwad, M. ., Gupta, K. ., Joshi, S. ., Gopika, G., Joshi, R., Gonge, S., & Kotecha, K. . (2023). Optimizing Hyperparameters for Enhanced LSTM-Based Prediction System Performance. International Journal on Recent and Innovation Trends in Computing and Communication, 11(10s), 203–213. https://doi.org/10.17762/ijritcc.v11i10s.7620

[52] Bhole, G. V., et al. "Implementation of Virtual Mouse Control System Using Hand Gestures for Web Service Discovery." International Journal of Intelligent Systems and Applications in Engineering 12.13s (2024): 663-672.

[53] M. Jane. Irwin, K. de. Bosschere, ACM Special Interest Group on Programming Languages., and Association for Computing Machinery. Special Interest Group on Embedded Systems., LCTES 2006 : proceedings of the 2006 ACM SIGPLAN/SIGBED Conference for Languages, Compilers, and Tools for Embedded Systems : June 14-16, 2006, Ottawa, Ontario, Canada. Association for Computing Machinery, 2006.