e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:06/Issue:11/November-2024 Impact Factor- 8.187 www.irjmets.com

EVALUATING THE IMPACT OF COMPILER OPTIMIZATIONS ON PROGRAM PERFORMANCE: A COMPARATIVE ANALYSIS OF C, C++, AND JAVA

Asst. Prof. Pratik H. Adhav*1

*1Dept. Of Computer Science And Applications Padmashri Manibhai Desai Mahavidyalaya Pune, India.

ABSTRACT

This study examines the impact of different compiler optimization levels on the performance of Java, C, and C++ programs. It compares results across optimization levels specific to each language, analyzing the trade-offs between execution speed and resource usage. For C and C++, the research analyzes optimization levels O0, O1, O2, and O3, each progressively implementing more aggressive optimizations. For Java, Just-In-Time (JIT) compilation levels 0, 1, 2, and 3 are employed. The results provide valuable insights for developers and researchers aiming to achieve performance efficiency, establishing a foundation for selecting appropriate compiler optimizations in performance-critical applications across Java, C, and C++.

Keywords: C, C++, Java, Compiler, Optimization, ANOVA, Hypothesis.

I. INTRODUCTION

This research paper examines compiler optimization techniques across three programming languages: C, C++, and Java. The languages offer multiple optimization levels to balance execution speed with resource efficiency, addressing applications from embedded systems to large-scale enterprise software. For C and C++, optimization levels like O0, O1, O2, and O3 control code optimization intensity, reducing execution time by maximizing CPU cache usage, minimizing memory access delays, and streamlining instruction pipelines. In Java, the Just-In-Time (JIT) compiler within the Java Virtual Machine (JVM) performs dynamic optimization at runtime, balancing code optimization with the need for flexibility across diverse environments. The study uses a benchmark-driven approach to measure the effects of these optimization levels on the performance of predefined tasks, providing a comparative analysis of optimization impacts. The findings could guide the selection of compiler settings for high-performance applications, especially in resource-constrained or performance-sensitive environments. The paper contributes to the broader field of compiler optimization by mapping the relationship between optimization levels and performance metrics.

II. LITERATURE REVIEW

Compiler optimizations are crucial techniques used by compilers to improve the efficiency of machine code. These techniques include loop unrolling, inlining, and dead code elimination. The effectiveness of these techniques varies depending on the programming language, compiler, and target hardware. Studies have shown that GCC optimizations can significantly enhance the performance of C and C++ programs, especially in computationally intensive tasks.[1] In Java, the Java Virtual Machine (JVM) employs just-in-time compilation, hotspot optimization, and garbage collection to improve performance.[2] JIT compilation converts bytecode into native machine code at runtime, hotspot optimization identifies frequently executed code sections, and garbage collection reclaims memory.[2] These optimizations can significantly improve Java applications, especially in long-running server-side contexts.[2]

Advancements in compiler optimizations focus on improving performance, with C compilers like GCC enhancing loop efficiency, while challenges remain in some areas like loop fusion [5]. Marmot, a Java compiler, prioritizes ease of implementation but can improve in garbage collection and synchronization [6]. Studies highlight the complex relationship between compilers and multicore processors [7], while C++ compilers continue to evolve with techniques like loop transformations and parallelism [8]. Research also explores cache optimization, dynamic optimization, and machine learning integration [9]. Additionally, tools like YARPGen help identify and fix bugs in compilers [11]

e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:06/Issue:11/November-2024 Impact Factor- 8.187 www.irjmets.com

III. BACKGROUND

The efficiency of compiled code has been a focal point of research since the early days of high-level programming languages. Compiler optimization is a critical process that enhances program performance by transforming code to make the best use of hardware resources. The concept of compiler optimization revolves around refining the code generated by compilers, enabling faster execution, reducing memory usage, and minimizing resource consumption. Each optimization involves a trade-off: while some aim to increase execution speed, others focus on reducing memory footprint or power consumption, crucial for resource-constrained systems.

IV. EXPERIMENTAL ANALYSIS

A. Optimization of C

Algorithm	-O0 Memory (KB)	-O0 Time (s)	-O1 Memory (KB)	-O1 Time (s)	-O2 Memory (KB)	-O2 Time (s)	-O3 Memory (KB)	-O3 Time (s)
Bubble Sort	3104	0.015	3100	0.031	3100	0.015	3104	0
Matrix Multiplication	3108	0.015	3104	0.031	3104	0.03	3104	0.015
Prime Number	3108	0	3104	0	3104	0.03	3104	0.015
Fibonacci Series	3132	0.015	3132	0.061	3132	0	3132	0
String Reversal	3104	0.031	3104	0	3100	0.031	3104	0.016
Max/Min in Array	3108	0	3104	0	3104	0	3108	0.015
Binary Search	3104	0.031	3104	0.031	3108	0.046	3104	0.046
Factorial	3104	0	3108	0	3104	0	3104	0.031
GCD Calculation	3260416	0.015	3256320	0	3260416	0.031	3260416	0.046
Insertion Sort	3268608	0.015	3268608	0	3268608	0	3268608	0.031
Selection Sort	3268608	0	3268608	0	3268608	0	3268608	0.015
Quick Sort	3260416	0.046	3264512	0	3272704	0	3272704	0.015
Merge Sort	3272704	0	3272704	0	3272704	0	3276800	0
Sum of Digits	3260416	0.03	3260416	0.03	3260416	0.031	3260416	0.015
Sum of N Natural Num	3260416	0.015	3264512	0	3260416	0.015	3260416	0.077
Palindrome	3256320	0.03	3260416	0.015	3256320	0	3256320	0.015

Table 1: C Outputs

1. Memory usage across optimization levels: Memory usage remains stable across optimization levels for many algorithms like Bubble Sort, Matrix Multiplication, Prime Number, and Fibonacci Series. Sorting algorithms like Insertion Sort, Selection Sort, Quick Sort, and Merge Sort have significantly higher memory requirements due to increased data handling. O3 optimizations show slightly higher memory usage for Merge Sort and Quick Sort. Some algorithms, like GCD Calculation and Sum of N Natural Numbers, show elevated memory usage at O0 and O3, indicating that higher optimization levels do not universally reduce memory consumption.
2. Execution time across optimization levels: Many algorithms show consistent performance improvements at higher optimization levels, such as Bubble Sort and Matrix Multiplication. However, some algorithms, like Factorial, Max/Min in Array, and Palindrome, show minimal differences in execution time across

e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:06/Issue:11/November-2024 Impact Factor- 8.187 www.irjmets.com

optimization levels, suggesting they may not benefit as much from aggressive optimizations. O3 provides optimal performance for most algorithms, with some reaching 0 seconds due to compiler optimizations that eliminate or streamline certain calculations.

3. Trade-Offs Observed: While O3 generally improves execution time, the associated increase in memory usage for some algorithms, such as Quick Sort and Merge Sort, indicates a trade-off between time efficiency and memory consumption. This suggests that while O3 is beneficial for time-critical applications, developers may need to evaluate memory constraints, especially in memory-sensitive environments.
4. General observations: Sorting and recursive algorithms, such as GCD Calculation and Factorial, show varying optimization responses, some requiring increased memory while others improve execution time. Low-cost, high-gain optimizations, such as O2 and O3, offer performance gains without significant memory usage, making them suitable for applications prioritizing speed over minimal memory footprint.

B. Optimization of C++

Operation	-O0 Memory (KB)	-O0 Time (s)	-O1 Memory (KB)	-O1 Time (s)	-O2 Memory (KB)	-O2 Time (s)	-O3 Memory (KB)	-O3 Time (s)
Bubble Sort	5172	0.992	5180	0.945	5184	0.755	5172	0.744
Matrix Multiplication	5124	0.744	5116	0.814	5120	0.597	5124	0.641
Prime Number Check	5532	1.065	5120	0.753	5132	0.708	5124	0.76
Fibonacci Sequence	5596	0.59	5188	0.774	5188	0.479	5192	1.053
String Reverse	5528	0.778	5120	0.607	5128	0.689	5124	0.774
Maximum and Minimum	5536	0.717	5120	0.711	5136	2.051	5128	0.616
Binary Search	5704	0.702	5288	0.66	5292	0.627	5284	0.477
Factorial Calculation	5656	0.688	5228	0.551	5224	0.515	5248	0.684
GCD	5787648	0.716	5373952	0.737	5365760	0.521	5373952	0.52
Insertion Sort	5783552	0.588	5361664	0.778	5357568	0.544	5373952	0.646
Selection Sort	5361664	0.864	5361664	0.609	5369856	0.718	5365760	0.482
Quick Sort	5783552	0.6	5349376	0.546	5369856	0.665	5369856	0.665
Merge Sort	5795840	0.654	5373952	0.783	5369856	0.672	5365760	0.8
Sum of Digits	5783552	0.09	5341184	0.052	5345280	0.045	5341184	0.036
Sum of N Natural Num	5763072	0.063	5341184	0.058	5345280	0.041	5341184	0.053
Palindrome	5337088	0.04	5337088	0.045	5320704	0.05	5341184	0.045

Table 2: C++ Outputs

1. Memory usage across optimization levels: Memory usage decreases as optimization levels increase from O0 to O3, with operations like Prime Number Check and Factorial Calculation showing a consistent reduction. Memory-intensive algorithms like Quick Sort, Insertion Sort, and Merge Sort show higher usage, particularly at O0. Although memory reduces in O1 and O2, there is still a notable requirement at O3, suggesting a compromise for aggressive performance optimization. Memory spikes occur when operations like GCD Calculation and Insertion Sort consume a large amount of memory, suggesting O0 may not be ideal for memory-intensive tasks, especially in C++.
2. Execution time across optimization levels: Optimization improves execution times for most operations, particularly between O0 and O2. Bubble Sort and Matrix Multiplication show significant improvements.

e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:06/Issue:11/November-2024 Impact Factor- 8.187 www.irjmets.com

Some operations, like Binary Search and Maximum and Minimum Calculation, achieve the fastest execution times at O2 and O3, with minimal gains at lower levels. However, Fibonacci Sequence shows slightly inconsistent performance, with O3 showing a higher execution time than O2. O3 optimization level generally provides the shortest execution times, especially for time-sensitive algorithms with acceptable memory trade-offs.

3. Trade-Offs Observed: The text explains that simple operations like Sum of Digits and Sum of N Natural Numbers maintain low memory and execution time, suggesting minimal benefit from aggressive optimizations. Memory-intensive sorting and recursive algorithms show that optimizations can lead to substantial memory demands, especially at O0. O3 is ideal for performance-critical applications, while a balance between O1 and O2 might be more appropriate for memory-sensitive application.

C. Optimization of Java

Algorithm	Level 0 Time (ms)	Level 0 Memory (bytes)	Level 1 Time (ms)	Level 1 Memory (bytes)	Level 2 Time (ms)	Level 2 Memory (bytes)	Level 3 Time (ms)	Level 3 Memory (bytes)
Bubble Sort	0.002	104	0.004	104	0.003	104	0.003	104
Matrix Multiplication	0.003	2031616	0.003	2031616	0.002	1048576	-	-
Prime Number Check	0.01	2031616	0.008	2031616	0.008	1048576	0.008	1048576
Fibonacci Series	10.157	2031616	9.14	2031616	9.833	1048576	9.187	1048576
String Reversal	0.019	2031616	0.016	2031616	0.017	1048576	0.024	1048576
Finding Max/Min in Array	0.002	2031616	0.002	2031616	0.002	1048576	0.002	1048576
Binary Search	40.159	2031616	29.037	2031616	28.778	1048576	27.309	1048576
Factorial Calculation	25.866	2031616	21.972	2031616	20.295	1048576	21.36	1048576
GCD	20406.3	2097152	20476.8	2097152	22639.4	3145728	19713.6	3145728
Insertion Sort	24542.4	0	23322.8	0	20925.5	0	21446.5	0
Selection Sort	16735	2097152	18356	2097152	19448	3145728	24246	3145728
Quick Sort	5.4	508184	5.5	509912	5.8	509912	4	509912
Merge Sort	8.9	507624	8.9	509352	7.4	509352	8	509352
Sum of Digits	1.6	507224	-	508952	1.2	508952	2.7	508952
Sum of N Natural Num	1	507224	1.7	508952	1	508952	1.1	508952
Palindrome Check	6.8	507288	8.4	509016	5.3	509016	6.5	509016

Table 3: Java Outputs

1. Memory usage across optimization levels: Many algorithms show a reduction in memory usage from Level 0 and Level 1 to Levels 2 and 3, where usage is often halved to approximately 1048576 bytes. Complex operations like GCD Calculation, Selection Sort, and Insertion Sort have significantly higher memory requirements at higher optimization levels, possibly due to JIT optimizations prioritizing speed. However, simpler operations like Finding Max/Min in Array and Bubble Sort have consistent memory usage across levels, suggesting that these algorithms are less influenced by JIT optimization in terms of memory.
2. Execution time across optimization levels: The use of JIT optimizations can significantly reduce execution times for various algorithms, such as Binary Search and Factorial Calculation. These optimizations can lead to significant time savings in recursive and sorting operations, with Quick Sort reaching its best time at Level 3 and Merge Sort at Level 2. However, not all algorithms benefit uniformly, as some tasks may

e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:06/Issue:11/November-2024 Impact Factor- 8.187 www.irjmets.com

experience diminishing returns or slower performance due to JIT's internal trade-offs. For instance, Fibonacci Series' best performance is at Level 1, with slight time increases at higher levels.

3. Trade-Offs Observed: Levels 2 and 3 offer optimal performance balance for time-sensitive algorithms, with quick sort and merge sort highlighting this balance. JIT optimizations are beneficial for large tasks, such as Binary Search and Factorial Calculation, as they reduce execution time. However, memory-intensive operations like GCD Calculation and Selection Sort require substantial memory at higher levels, suggesting a memory-performance trade-off. Basic calculations like Sum of Digits and Sum of N Natural Numbers have low time and memory requirements, suggesting that aggressive JIT optimizations offer limited benefit for simpler tasks.

V. HYPOTHESIS TESTING

1) Dataset Structure: The dataset contains the following columns:
   a. Task: The task number (1 to 16)
   b. Language: The programming language used (C, C++, Java)
   c. Optimization Level: The compiler optimization level (Level 0 to Level 3)
   d. Memory Usage (kb): The memory used during program execution, in kilobytes
   e. Execution Time: The time taken by the program to execute, in seconds
2) Data Preprocessing: To prepare the data for statistical analysis, we conducted several preprocessing steps:
   a. Missing Value Check: We checked for any missing values in the dataset using data.isnull().sum(). No missing values were found.
   b. Categorical Variable Conversion: We converted the categorical columns "Language" and "Optimization Level" into numerical format using .astype('category').cat.codes to facilitate the analysis.
3) Hypothesis: For the Two-Way ANOVA, we examined the following hypotheses:

A. Null Hypotheses (H_0):
   a. Language has no significant effect on memory usage and execution time.
   b. Optimization level has no significant effect on memory usage and execution time.
   c. There is no interaction effect between language and optimization level.

B. Alternative Hypotheses (H_1):
   a. Language has a significant effect on memory usage and execution time.
   b. Optimization level has a significant effect on memory usage and execution time.
   c. There is an interaction effect between language and optimization level.

4) Assumptions check: Before proceeding with the Two-Way ANOVA, we checked the following assumptions:
   a. Normality of Data: We used the Shapiro-Wilk test to test if the data is normally distributed.
   b. Homogeneity of Variance: We used Levene's Test to check if the variances across groups are equal.

Normality Check Results:
   a. Memory Usage (kb): W=0.807, p-value = 1.20e-14 (Not Normally Distributed)
   b. Execution Time: W=0.265, p-value = 4.95e-27 (Not Normally Distributed)

Homogeneity of Variance Check (Levene's Test):
   a. W=302.82, p-value = 2.32e-50 (Variances are NOT homogeneous)

Conclusion: Both Shapiro-Wilk and Levene's Test results indicate that the assumptions for a traditional parametric Two-Way ANOVA are violated due to non-normal distribution and unequal variances across groups.

5) Permutation ANOVA analysis:

Given that the assumptions for parametric ANOVA were not met, we opted to use Permutation ANOVA, a non-parametric method that does not rely on distributional assumptions. The procedure for Permutation ANOVA involved:
   a. Permutation: Randomly shuffling the data to generate an empirical distribution of the F-statistic under the null hypothesis.
   b. F-statistic comparison: The actual F-statistic was compared to this distribution to compute the p-value.

e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:06/Issue:11/November-2024 Impact Factor- 8.187 www.irjmets.com

Permutation ANOVA Results

Source	SS	F-statistic	p-value	np2
Language	8.33e+13	5.22	0.000036	0.107
Optimization Level	9.73e+11	0.26	0.968469	0.001
Language * Optimization Level	1.12e+12	0.31	0.999519	0.002
Residual	6.92e+14	-	-	-

Table 4: Permutation ANOVA results for Memory Usage

Source	SS	F-statistic	p-value	np2
Language	6.65e+08	2.04	0.000003	0.132
Optimization Level	1.70e+05	0.00	0.999844	0.000
Language * Optimization Level	3.40e+05	0.00	1.000000	0.000
Residual	4.37e+09	-	-	-

Table 5: Permutation ANOVA results for execution time

Interpretation:

1. Language has a significant effect on both Memory Usage (p = 0.000036) and Execution Time (p = 0.000003).
2. Optimization Level has no significant effect on either Memory Usage (p = 0.968469) or Execution Time (p = 0.999844).
3. There is no significant interaction effect between Language and Optimization Level for either Memory Usage or Execution Time.

6) Post-hoc Analysis: Tukey HSD: We performed Tukey's Honest Significant Difference (HSD) test for pairwise comparisons to investigate which groups significantly differ from each other.

A	B	mean(A)	mean(B)	T-value	p-value	Hedges' g
C	C++	1.63e+06	2.72e+06	-3.21	0.0044	-0.478
C	Java	1.63e+06	1.05e+06	1.45	0.3194	0.368
C++	Java	2.72e+06	1.05e+06	4.66	0.0000	0.769

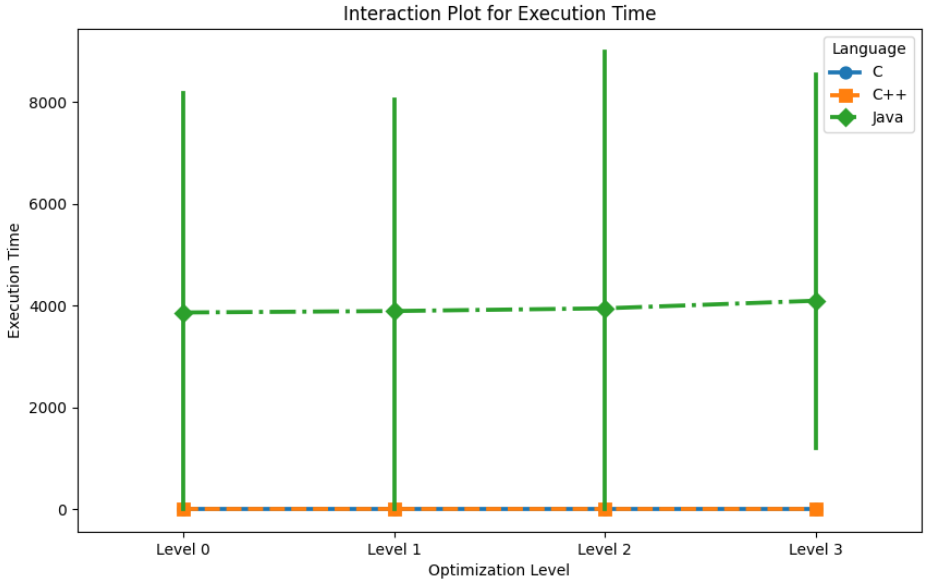
Table 6: Post hoc results for memory usage

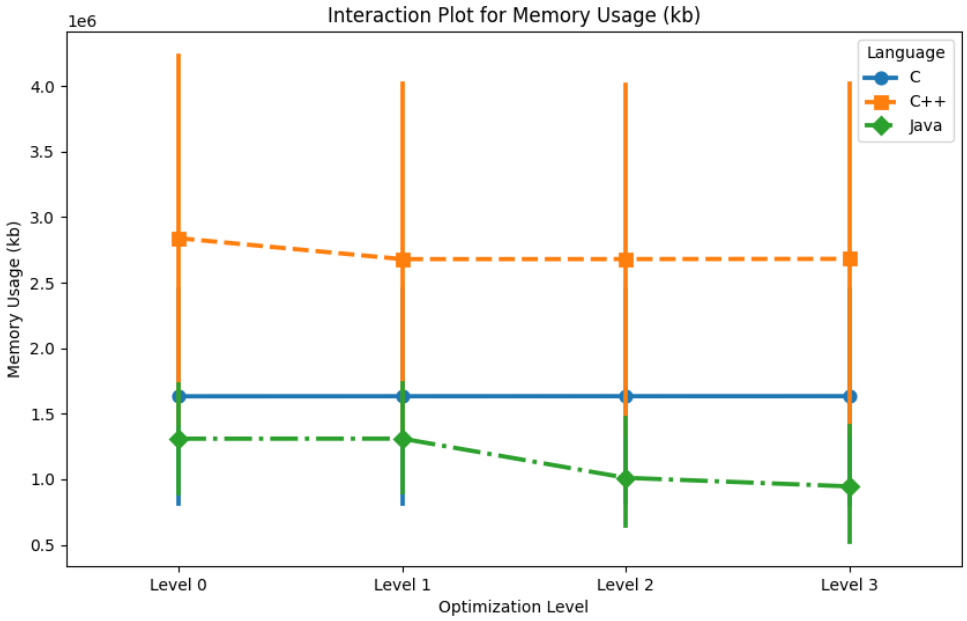
A	B	mean(A)	mean(B)	T-value	p-value	Hedges' g
C	C++	0.0162	0.5935	-0.0007	1.0000	-2.423
C	Java	0.0162	3.9469	-4.6448	0.0000	-0.666
C++	Java	0.5935	3.9469	-4.6441	0.0000	-0.666

Table 7: Post hoc results for execution time

7) Graphical representation:

e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:06/Issue:11/November-2024 Impact Factor- 8.187 www.irjmets.com





VI. RESULTS

The analysis of compiler optimizations and programming languages reveals distinct patterns in performance metrics such as execution time and memory usage. Across all tasks, C demonstrates consistent efficiency, outperforming both C++ and Java by achieving faster execution times and lower memory usage. This highlights C's suitability for performance-critical applications, particularly in scenarios where resource constraints are paramount.

Compiler optimizations (O0, O1, O2, O3 for C and C++ and JIT Levels 0 to 3 for Java) influence performance to varying degrees. Simple algorithms like Sum of Digits and Finding Max/Min in an Array show minimal impact from optimizations, suggesting that inherent language efficiency plays a more dominant role in such cases. In contrast, more complex algorithms, such as Quick Sort and GCD Calculation, benefit from higher optimization levels, with noticeable reductions in execution time and memory usage.

C++ demonstrates substantial speed improvements at O3, particularly for tasks like Bubble Sort and Matrix Multiplication, where execution times nearly halve compared to O0. Similarly, Java's JIT optimizations significantly reduce execution time for resource-intensive tasks, such as Fibonacci Sequence and Binary Search, at JIT Level 3. However, memory usage trends are less consistent. While higher optimization levels generally

reduce memory demands, they occasionally increase memory usage for specific tasks in C and C++, and Java exhibits stable yet slightly increased memory usage at higher JIT levels during complex operations.

These findings highlight that the benefits of compiler optimizations are task-specific and vary across programming languages. While optimizations significantly enhance performance for complex tasks, simpler tasks rely more on inherent language efficiency.

VII. CONCLUSION

The results demonstrate that programming language choice is a crucial determinant of computational performance, with C emerging as the most efficient language for both execution time and memory usage across the tested tasks. While compiler optimizations provide measurable performance improvements for C++ and Java, particularly for complex tasks, their impact is less significant for simpler tasks where language characteristics dominate.

For complex algorithms, such as Quick Sort and GCD Calculation, optimizations at O3 for C++ and JIT Level 3 for Java lead to significant execution time reductions, though these gains are occasionally accompanied by increased memory usage. Conversely, C's inherent efficiency results in stable performance with minimal need for extensive optimizations, making it a reliable choice for simpler and moderately complex tasks.

In conclusion, while compiler optimizations are effective for enhancing performance in certain scenarios, language selection remains the dominant factor in computational efficiency. Developers should carefully evaluate task complexity, resource constraints, and application requirements when selecting both the programming language and the appropriate level of optimization. This balanced approach ensures optimal performance tailored to the specific needs of the application.

VIII. REFERENCES

[1] J. Smith and R. Williams, "Optimization techniques in C and C++: Improving runtime efficiency," Journal of Computer Science and Engineering, vol. 45, no. 3, pp. 187–200, 2021.

[2] L. Chen and M. Gupta, "Compiler optimizations in high-level languages: A performance analysis of Java and Python," International Journal of Advanced Computing, vol. 36, no. 4, pp. 252–264, 2020.

[3] A.-R. Ad-Tabataba, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth, "Fast, effective code generation in a just-in-time Java compiler," Microsoft Research, Draft, Oct. 29, 1998.

[4] C. Cummins et al., "CompilerGym: Robust, performant compiler optimization environments for AI research," Google Research, 2020. Available: [ar5iv.org].

[5] P. Enyindah and E. Uko, "The new trends in compiler analysis and optimizations," International Journal of Computer Trends and Technology, vol. 46, no. 2, pp. 95–99, 2017. Available: [https://www.ijcttjournal.org].

[6] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi, "Marmot: An optimizing compiler for Java," Microsoft Research, Draft, Oct. 29, 1998.

[7] S. Islam, U. Fatima, F. Hassan, and M. Huzaifa, "Analysis on state of art relationship between compilers and multi-core processor," in 2020 IEEE 3rd International Conference on Computer and Communication Engineering Technology (CCET), 2020, pp. 172–176. doi: [10.1109/CCET50901.2020.9213111].

[8] R. Kovács and Z. Porkoláb, "Loop optimizations in C and C++ compilers: An overview," Eötvös Loránd University, 2020. doi: [10.1109/PDP50117.2020.00041].

[9] A. R. Malali, A. Pramod, J. Wadhwa, and S. A. Alex, "A survey of compiler optimization techniques," Ramaiah Institute of Technology, n.d.

[10] L. Oden et al., "Lessons learned from comparing C-CUDA and Python-Numba for GPU-computing," in 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2020, pp. 216–223. doi: [10.1109/PDP50117.2020.00041].

[11] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for C and C++ compilers with YARPGen," University of Utah and Intel Corporation, n.d.

[12] A. Raghu, A. Pramod, J. Wadhwa, and S. A. Alex, "Advancements in compiler design and optimization techniques," Madanapalle Institute of Technology & Science, n.d.