The Influence of Programming Languages on Computational Efficiency and Performance

Amine Azenkouk[1], Hiba Aghbal[2], Fatima Zahra Mouchtachi[3]

[1]Major: Computer Science, Nanjing University of Information Science and Technology

[2,3]Major: Artificial Intelligence, Nanjing University of Information Science and Technology

DOI: https://dx.doi.org/10.47772/IJRISS.2025.91100023

Received: 07 November 2025; Accepted: 14 November 2025; Published: 27 November 2025

INTRODUCTION

Imagine you need to build a house. You could choose to build it quickly with pre-made materials, or you could take more time to craft everything by hand for perfect precision. The tools and materials you choose change the speed of construction and the final quality of the home.

Programming languages are like those tools for building software. Every programming language is designed with different goals. Some, like Python, are created to be simple and allow developers to write code quickly. Others, like C++, are built to give the programmer a lot of control to make software run as fast and efficiently as possible.

This paper explores a simple but important question: How does the choice of a programming language affect the speed and efficiency of the software it creates?

We will explore why a program written in one language might run instantly, while the same program written in another language might be slower. We will look at the key reasons for these differences, such as whether a language compiled (translated into machine code beforehand, like C++) or interpreted (translated on the fly while running, like Python). We will also discuss how languages manage memory and how that impacts performance.

Ultimately, this research shows that there is no single "best" language. The choice is a classic trade-off: the need for raw speed and efficiency versus the need for fast development and ease of use. Understanding this balance is crucial for software developers and engineers to make the right choice for their specific project.

LITERATURE REVIEW:

Existing research on programming language performance centers on key debates and benchmarking studies.

2.1 The Compiled vs. Interpreted Divide:

Early studies (e.g., Prechelt, 2000) established a clear performance gap. Natively compiled languages like **C++** consistently outperformed interpreted languages like Python in execution speed and memory use, due to interpretation overhead and dynamic typing.

2.2 The Java JIT Revolution:

The introduction of **Java** and its Just-In-Time (JIT) compiler complicated this narrative. Research showed that JIT compilation could profile code at runtime and optimize it to achieve performance much closer to that of C++ for many tasks, though often at the cost of higher memory usage and occasional garbage collection pauses.

2.3 The Scripting Language Paradox:

A significant finding explains the popularity of "slow" languages like Python in high-performance fields (e.g., data science). Studies note that their performance-critical libraries (e.g., NumPy) are implemented in fast, compiled languages like C and Fortran. The language itself acts as a productive "glue," delegating expensive operations to efficient native code.

2.4 Modern Trade-offs: Safety vs Speed:

Recent debate focuses on newer languages like **Rust** and **Go**, which seek a better trade-off between performance and safety. Benchmarks show Rust achieves C-level performance without its vulnerabilities, using a novel ownership model for memory safety without a garbage collector. Go prioritizes fast compilation and simple concurrency for web services, accepting slightly lower peak performance than Rust or C++.

In summary, research shows that performance is not a simple ranking but a trade-off between speed, memory, safety, and development time. The choice between a natively compiled, JIT-compiled, or interpreted language depends heavily on the specific application domain and its requirements.

1. Comparative analysis of Programming languages:

3.1 Efficiency and Execution Time:

A meaningful comparison of programming language efficiency requires that benchmarks be conducted on a consistent hardware and software platform to ensure equitable testing conditions. The most critical evaluation must occur within a practical operational environment, measuring tangible performance metrics such as real execution time to draw valid conclusions.

However, this empirical approach introduces the challenge of environmental variability, which can affect the consistency of results and reduce their accuracy. For instance, Java's implementation presents specific hurdles. Its execution within a Java Virtual Machine (JVM), while enabling platform independence and dynamic type safety, necessitates frequent runtime checks—such as array bounds verification and type casting validation. Although the JVM's Just-In-Time (JIT) compiler optimizes these operations, they nonetheless impose a measurable performance overhead compared to natively compiled languages.

Despite these inherent variabilities, such testing remains essential because software must perform reliably in diverse real-world conditions. To accurately compare the performance of languages like C and Java, a robust methodological approach is required. This involves implementing identical algorithms in both languages, executing them repeatedly under controlled conditions, and analyzing the results using statistical measures like the mean and standard deviation of execution times. Such a process typically reveals that natively compiled C code executes with significantly lower overhead and greater speed than Java.

This performance disparity can be attributed to fundamental design differences. C allows for fine-grained control and the use of highly optimized library functions, though excessive function calls can introduce their own overhead. Java, while inheriting and abstracting many features from C into a more verbose but safer high-level language, pays for its security and portability features with increased runtime cost. Consequently, obtaining accurate performance data necessitates rigorous, repeated testing to account for these inherent design trade-offs.

3.2 Usage of various programming languages:

Selecting an appropriate programming language is a critical determinant of success in software development, as an unsuitable choice can significantly increase both project workload and overall cost. Different languages excel in specific domains due to their inherent design philosophies and performance characteristics.

The C programming language is predominantly employed in the development of operating systems, embedded systems, and server infrastructure. Its power lies in providing high-level abstraction while retaining low-level

control, making it a foundational tool for system-level programming. A quintessential example of its capability is the Microsoft Windows kernel, which is written in C and powers over 90% of the world's desktop operating systems.

In contrast, Java is a cornerstone of enterprise application development, website platform construction, and Android mobile application development. It remains one of the most widely available and competitive languages in the market. Its influence on the software industry and computing paradigm is profound, particularly through its object-oriented design and platform-independent virtual machine. However, addressing all software development challenges requires more than a single paradigm; effective solutions often depend on stylistic choices and the specific problem domain, necessitating the effective integration of multiple approaches and languages.

C++ extends the capabilities of C and is heavily utilized in performance-intensive domains such as game development, office software, relational databases, and browser engines. Its enduring popularity is largely attributed to its ability to facilitate efficient low-level memory management while generating highly optimized machine code, offering a unique blend of high performance and object-oriented features.

Python has emerged as a dominant force in the fields of data science, machine learning, automated testing, and web crawling. Its extensive ecosystem of scientific libraries (e.g., NumPy, Pandas, Scikit-learn) and frameworks (e.g., Django, Flask) also makes it a popular choice for backend development, particularly among startups and small to *medium*-sized enterprises. A key factor in its widespread adoption is its emphasis on readability and simplicity, making it one of the most accessible languages for beginners to learn and use effectively.

CONCLUSION:

This paper demonstrates that programming languages inherently shape software performance through their design paradigms. Compiled languages (C, C++, Rust) excel in execution speed and memory efficiency but require greater development effort. JIT-compiled languages (Java) balance performance and productivity, while interpreted languages (Python) prioritize rapid development at the cost of runtime overhead. The rise of languages like Rust and Go highlights an ongoing evolution toward safer, more efficient alternatives. Ultimately, the "best" language depends on project-specific needs: raw performance, development speed, or safety. Developers must weigh these trade-offs to make informed choices.

REFERENCES

1. **Prechelt, L. (2000). "An Empirical Comparison of Seven Programming Languages." Computer, vol. 33, no. 10, pp. 23-29.**
   Classic benchmarking study comparing several popular languages for speed and memory usage.
2. **Sestoft, P. (2017). "Programming Language Performance Comparison." Rosetta Code**.
   Provides thorough comparative benchmarks for language efficiency across standard algorithms.
3. **Ousterhout, J. (1998). "Scripting: Higher-Level Programming for the 21st Century." IEEE Computer, vol. 31, no. 3, pp. 23-30.**
   Explains why "scripting" languages like Python thrive despite slower performance through their use of compiled extensions.
4. **Flanagan, D., & Matsumoto, Y. (2008). "The Ruby Programming Language." O'Reilly Media.**
   Discusses interpreter trade-offs and their impact on performance and developer productivity.
5. **Morandat, F., Hill, M., Osvald, L., & Vitek, J. (2012). "Evaluating the Design of the R Language: Objects and Interfaces." Proceedings of the 26th European Conference on Object-Oriented Programming.**
   Benchmarks scripting languages, exploring why libraries are often implemented in compiled languages.
6. **Microsoft. (n.d.). "Windows Kernel Programming." Microsoft Docs.**
   Official documentation highlighting C's critical role in high-performance system-level programming.
7. **Matsakis, N. D., & Klock, F. S. (2014). "The Rust Language." ACM SIGAda Ada Letters, vol. 34, no. 3, pp. 103-104.**

Explains Rust's ownership model and its effect on safety and performance.

8. **Golang Team. (n.d.). "Go at Google: Language Design in the Service of Software Engineering." ACM Queue, vol. 10, no. 2.**
   Discusses Go's design philosophy for balancing concurrency, compilation speed, and computational efficiency.