No coração dos sistemas: a relevância da programação de baixo nível para a eficiência e a segurança em aplicações modernas

At the heart of systems: the relevance of low-level programming for efficiency and security in modern applications

Guilherme Ferreira Sousa[1]
João Pedro Bianquini Murucci[2]
Marcos Antônio Pereira Coelho[3]

RESUMO

A programação de baixo nível desempenha um papel fundamental no desenvolvimento de sistemas computacionais que exigem desempenho elevado e segurança reforçada. Em um contexto dominado por linguagens de alto nível e abstrações cada vez mais distantes do hardware, este trabalho investiga como o domínio da programação de baixo nível pode contribuir significativamente para a eficiência e a robustez de aplicações modernas. O estudo adota uma abordagem aplicada, exploratória e explicativa, combinando revisão bibliográfica com experimentação prática. Foram implementados e comparados algoritmos equivalentes nas linguagens Assembly, C, Rust, Java e Python, avaliando-se métricas como tempo de execução e consumo de memória. Os resultados demonstram que as linguagens de baixo nível, em especial Assembly e C, apresentam desempenho superior e maior controle sobre os recursos computacionais. Rust também se destacou por oferecer segurança de memória com performance próxima ao C. A análise reforça a importância da programação de baixo nível em sistemas embarcados, dispositivos IoT, bibliotecas gráficas, kernels e aplicações críticas, além de apontar para a necessidade de sua valorização no ensino e na prática profissional.

Palavras-chave: programação de baixo nível; desempenho; segurança de sistemas; linguagens de programação; eficiência computacional.

[1] Discente do curso de Sistemas de Informação da Universidade do Estado de Minas Gerais (UEMG), Carangola/MG. E-mail: guilherme.1295232@discente.uemg.br. ORCID: https://orcid.org/0009-0008-2624-7921.

[2] Discente do curso de Sistemas de Informação da Universidade do Estado de Minas Gerais (UEMG), Carangola/MG. E-mail: joao.1295220@discente.uemg.br. ORCID: https://orcid.org/0009-0001-9138-8068.

[3] Mestre em Cognição e Linguagem pela Universidade Estadual do Norte Fluminense (UENF), Campos dos Goytacazes/RJ. Docente do Curso de Sistemas de Informação da Universidade do Estado de Minas Gerais (UEMG), Carangola/MG. E-mail: maredumig@gmail.com. ORCID: https://orcid.org/0000-0002-8454-1896.

ABSTRACT

Low-level programming plays a fundamental role in the development of computational systems that require high performance and enhanced security. In a context dominated by high-level languages and abstractions increasingly distant from the hardware, this work investigates how mastery of low-level programming can significantly contribute to the efficiency and robustness of modern applications. The study adopts an applied, exploratory, and explanatory approach, combining literature review with practical experimentation. Equivalent algorithms were implemented and compared in the languages Assembly, C, Rust, Java, and Python, evaluating metrics such as execution time and memory consumption. The results demonstrate that low-level languages, especially Assembly and C, present superior performance and greater control over computational resources. Rust also stood out for offering memory safety with performance close to that of C. The analysis reinforces the importance of low-level programming in embedded systems, IoT devices, graphics libraries, kernels, and critical applications, in addition to pointing out the need for its valorization in education and professional practice.

Keywords: low-level programming; performance; system security; programming languages; computational efficiency.

1 INTRODUÇÃO

A crescente dependência de sistemas computacionais em setores críticos da sociedade, como saúde, indústria, finanças e infraestrutura, tem intensificado a necessidade por soluções que combinem alto desempenho, confiabilidade e segurança. Nesse cenário, o desenvolvimento de software moderno tem sido fortemente influenciado por linguagens de alto nível, que priorizam abstração, produtividade e facilidade de manutenção. No entanto, essa elevação do nível de abstração frequentemente reduz o controle direto sobre os recursos de hardware, o que pode impactar negativamente tanto a eficiência quanto a previsibilidade e a segurança das aplicações (Patterson; Hennessy, 2017).

Em contrapartida, as linguagens de baixo nível, como C e Assembly, mantêm um papel fundamental ao oferecer maior proximidade com a arquitetura do sistema, permitindo controle refinado sobre memória, registradores e execução de instruções. Essa característica as torna especialmente relevantes em contextos nos quais o desempenho é crítico, como sistemas embarcados, aplicações de tempo real e componentes centrais de sistemas operacionais (Tanenbaum; Austin, 2013). Embora a superioridade de desempenho dessas linguagens seja amplamente reconhecida, a evolução recente das arquiteturas computacionais, aliada ao avanço de compiladores e à crescente complexidade dos sistemas, torna necessária uma reavaliação desse entendimento à luz de cenários contemporâneos.

Além do desempenho, a segurança computacional emerge como um fator central nessa discussão. Grande parte das vulnerabilidades exploradas atualmente está associada a falhas no gerenciamento de memória, como estouros de buffer e acessos inválidos, frequentemente relacionadas ao uso inadequado de linguagens que oferecem controle direto sobre recursos do sistema (Mitre, 2023; Nist, 2022). Nesse contexto, novas abordagens têm sido propostas com o objetivo de conciliar eficiência e segurança, destacando-se a linguagem Rust, que introduz mecanismos de verificação em tempo de compilação capazes de prevenir classes inteiras de erros sem comprometer o desempenho (Rust Foundation, 2023).

Diante desse panorama, este estudo não se limita à constatação da vantagem de desempenho das linguagens de baixo nível, mas busca ampliar essa discussão a partir de uma abordagem aplicada e contemporânea. A pesquisa tem como objetivo atualizar evidências empíricas por meio da análise comparativa de desempenho entre as linguagens Assembly, C, Rust, Java e Python, utilizando implementações equivalentes de algoritmos com diferentes características computacionais. Adicionalmente, o trabalho integra a análise de performance

com aspectos relacionados à segurança de memória, explorando como diferentes paradigmas de linguagem influenciam a ocorrência e a mitigação de vulnerabilidades.

A metodologia adotada caracteriza-se como aplicada, de natureza exploratória e explicativa, com abordagem mista. A análise qualitativa fundamenta-se na revisão de conceitos relacionados à arquitetura de computadores, ao desempenho e à segurança, enquanto a análise quantitativa baseia-se na execução de experimentos práticos envolvendo testes de tempo de execução e uso de memória em diferentes linguagens. Os experimentos contemplam algoritmos representativos, como cálculo recursivo, manipulação de vetores e operações matriciais, permitindo uma avaliação abrangente dos diferentes aspectos de desempenho.

Por fim, o estudo também busca evidenciar a relevância formativa do domínio de linguagens de baixo nível, especialmente diante da crescente abstração no desenvolvimento de software. A compreensão dos mecanismos fundamentais de execução e gerenciamento de recursos torna-se essencial para a formação de profissionais capazes de desenvolver sistemas mais eficientes, seguros e alinhados às demandas atuais da computação.

2 FUNDAMENTAÇÃO TEÓRICA

A programação de baixo nível permanece como um campo essencial para a compreensão profunda do funcionamento interno dos sistemas computacionais. Suas aplicações vão desde a otimização de desempenho até a segurança de sistemas críticos. Nesta seção, discutem-se os fundamentos conceituais, os impactos em performance e segurança, e sua importância em aplicações contemporâneas.

2.1 Fundamentos da programação de baixo nível

As linguagens de programação constituem o principal meio de comunicação entre o desenvolvedor e o sistema computacional, sendo classificadas, de modo geral, de acordo com o seu nível de abstração em relação ao hardware. Linguagens de baixo nível, como C e Assembly, oferecem maior controle sobre os recursos computacionais, permitindo manipulação direta de memória e instruções mais próximas da arquitetura da máquina. Em contrapartida, linguagens de alto nível, como Python e Java, introduzem camadas adicionais de abstração que facilitam o desenvolvimento, mas podem impactar o desempenho (Sebesta, 2019).

Essa diferença de abstração está diretamente relacionada à forma como os programas são executados. Em linguagens de baixo nível, o código tende a ser mais próximo das instruções nativas do processador, reduzindo a necessidade de interpretação ou tradução adicional durante a execução. Já linguagens de alto nível frequentemente dependem de máquinas virtuais ou interpretadores, o que introduz overhead computacional e pode afetar a eficiência do sistema (Tanenbaum; Bos, 2015).

Do ponto de vista arquitetural, a relação entre software e hardware é um fator determinante para o desempenho das aplicações. Quanto menor o nível de abstração da linguagem, maior é a capacidade de otimização em nível de hardware, permitindo melhor uso de registradores, memória e instruções específicas do processador. Esse aspecto é amplamente discutido na literatura de arquitetura de computadores, que destaca a importância da proximidade com o hardware para aplicações que exigem alto desempenho (Patterson; Hennessy, 2017).

Estudos empíricos reforçam essa relação ao demonstrar diferenças significativas de desempenho entre linguagens. Comparações clássicas indicam que linguagens compiladas, como C e C++, tendem a apresentar maior eficiência em tempo de execução quando comparadas a linguagens interpretadas, como Python, especialmente em tarefas computacionalmente intensivas (Prechelt, 2000). Esses resultados evidenciam que a escolha da linguagem impacta diretamente a performance do sistema.

Entretanto, a busca por desempenho não pode ser dissociada das questões de segurança. Linguagens de baixo nível, embora eficientes, estão frequentemente associadas a vulnerabilidades críticas, principalmente relacionadas ao gerenciamento manual de memória, como buffer overflows e ponteiros inválidos. De acordo com a base de dados CWE, mantida pelo MITRE, falhas dessa natureza continuam entre as mais exploradas em sistemas computacionais (Mitre, 2023).

Nesse contexto, surgem linguagens modernas de sistemas, como Rust, que buscam equilibrar desempenho e segurança. Rust introduz um modelo de gerenciamento de memória baseado em regras de propriedade e empréstimo (ownership e borrowing), permitindo garantir segurança de memória em tempo de compilação sem a necessidade de coletor de lixo. Esse modelo representa uma evolução significativa em relação às linguagens tradicionais, mantendo desempenho próximo ao de C, ao mesmo tempo em que reduz a incidência de vulnerabilidades (Matsakis; Klock, 2014).

Dessa forma, a compreensão dos níveis de abstração e suas implicações é essencial para a escolha adequada de linguagens de programação, especialmente em contextos que demandam alto desempenho e segurança. A evolução das linguagens demonstra uma tendência clara de busca por soluções que conciliem esses dois aspectos, refletindo as necessidades dos sistemas computacionais modernos.

2.1.1 Componentes fundamentais

A programação de baixo nível interage diretamente com elementos essenciais da arquitetura computacional: Registradores, Memória RAM, Unidade Central de Processamento (CPU) e Instruções de Máquina.

- Registradores

Os registradores são pequenas áreas de armazenamento localizadas no interior da CPU, projetadas para operações de acesso extremamente rápido. Eles armazenam dados temporários e resultados intermediários de cálculos, além de endereços de memória e instruções em execução. Devido à sua proximidade com a unidade de processamento, os registradores desempenham papel fundamental na eficiência das operações aritméticas e lógicas, sendo utilizados intensivamente durante a execução de programas em Assembly e outras linguagens de baixo nível (Borin, 2011, p. 12).

- Memória (RAM)

A memória de acesso aleatório (RAM) é o espaço onde dados e instruções são armazenados de forma temporária para acesso rápido pela CPU durante a execução de programas. A RAM permite a leitura e a escrita de informações em qualquer ordem, o que é fundamental para a flexibilidade e desempenho dos sistemas computacionais. Seu papel é fornecer à CPU os dados necessários para o processamento, bem como armazenar resultados intermediários e finais das operações realizadas (Tanenbaum; Bos, 2015, p. 45).

- Unidade Central de Processamento (CPU)

A Unidade Central de Processamento (CPU) é o componente responsável pela execução de instruções e pelo controle das operações em um sistema computacional, sendo essencial para o funcionamento de qualquer aplicação. Sua organização interna inclui a Unidade de Controle,

que coordena o fluxo de execução, a Unidade Lógica e Aritmética (ULA), responsável por operações matemáticas e lógicas, além de registradores que armazenam temporariamente dados e instruções. O desempenho da CPU está associado à eficiência na execução dessas operações, sendo influenciado por fatores como arquitetura, paralelismo e organização interna dos componentes, que impactam diretamente a capacidade de processamento do sistema (Stallings, 2019).

- Instruções de máquina

As instruções de máquina representam o nível mais baixo de programação compreendido diretamente pelo hardware, sendo codificadas em formato binário e executadas pela CPU sem necessidade de tradução adicional. Cada instrução especifica uma operação básica, como manipulação de dados, operações aritméticas ou controle de fluxo, e sua estrutura depende da arquitetura do processador. O conjunto dessas instruções, conhecido como Arquitetura do Conjunto de Instruções (ISA), define como o software interage com o hardware, influenciando diretamente o desempenho e a eficiência dos programas. A utilização desse nível exige maior conhecimento técnico, uma vez que envolve detalhes específicos da arquitetura e do funcionamento interno do sistema (Null; Lobur, 2018).

Esses componentes, em conjunto, formam a base sobre a qual se desenvolvem sistemas computacionais eficientes e robustos, sendo essenciais para a compreensão e a aplicação da programação de baixo nível.

2.1.2 Vantagens e desvantagens

As linguagens de programação de baixo nível apresentam como principal vantagem o alto desempenho, decorrente da proximidade com o hardware e do controle direto sobre recursos como memória e processamento. Essa característica permite maior eficiência na execução e na otimização de programas, sendo especialmente relevante em sistemas críticos e aplicações que exigem alto desempenho (Kernighan; Ritchie, 1988).

Por outro lado, essas linguagens apresentam desvantagens significativas, como maior complexidade de desenvolvimento e maior propensão a erros, especialmente relacionados ao gerenciamento manual de memória. A ausência de mecanismos automáticos de segurança pode resultar em vulnerabilidades, como acessos indevidos e falhas críticas no sistema (Erickson, 2008).

2.2 Otimização de desempenho

A otimização de desempenho é um dos pilares que justificam a importância da programação de baixo nível no desenvolvimento de sistemas modernos. Em um cenário em que a eficiência computacional impacta diretamente a experiência do usuário, os custos operacionais e a sustentabilidade energética dos sistemas, o controle granular proporcionado por linguagens como C e Assembly se torna uma vantagem estratégica.

Segundo Sutter (2005), "a performance ainda importa", sobretudo em aplicações críticas, como sistemas embarcados, jogos de alto desempenho e softwares em tempo real. Linguagens de baixo nível permitem que o desenvolvedor tenha controle direto sobre os registradores, a alocação de memória e o cache da CPU, o que possibilita decisões otimizadas que linguagens de alto nível geralmente abstraem. Essa proximidade com o hardware permite "espremer cada ciclo de clock" do processador, como explica Patterson e Hennessy (2017) ao tratarem da otimização de código em níveis microarquiteturais.

Em comparação com linguagens de alto nível, a programação de baixo nível permite eliminar overheads causados por garbage collectors, introspecção de tipos ou máquinas virtuais. Como apontam Mazloom e Trajkovic (2018), linguagens como Python e Java introduzem camadas de abstração que comprometem a previsibilidade e o tempo de execução, dificultando sua aplicação em contextos onde milissegundos fazem diferença, como em algoritmos de roteamento de redes ou em sistemas de resposta a emergências.

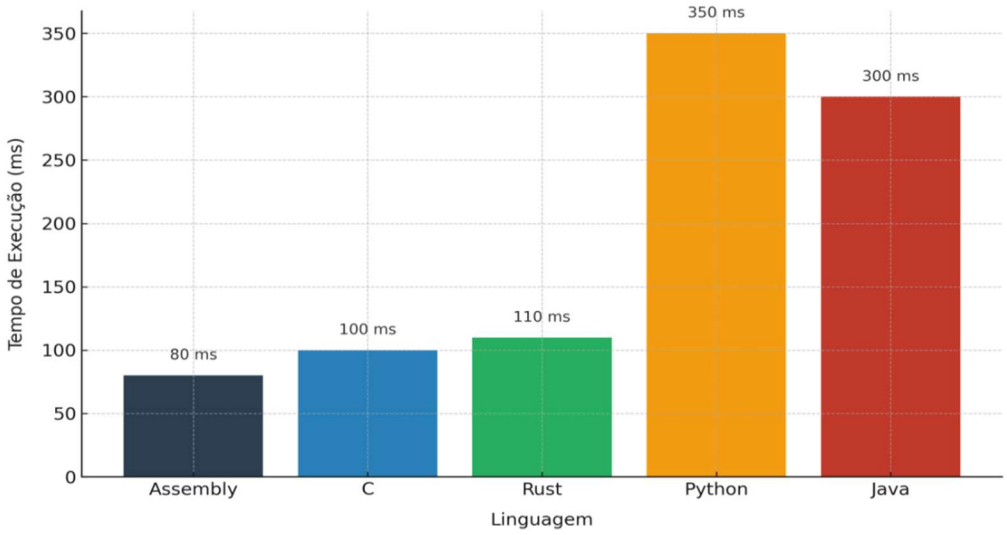
Além disso, o uso de instruções SIMD (Single Instruction, Multiple Data) e operações paralelas explicitadas manualmente são facilitadas por esse tipo de programação. Conforme descrito por Fog (2023), em sua referência sobre microarquitetura x86, compiladores modernos podem fazer otimizações, mas programadores experientes ainda conseguem resultados superiores ao escrever rotinas críticas manualmente.

Outro aspecto relevante é que o domínio da arquitetura de hardware permite ao programador explorar afinidades de cache, organização da memória e pipelines de instrução. Isso é especialmente importante em ambientes de tempo real e em sistemas operacionais, nos quais o desempenho e a previsibilidade precisam ser garantidos com precisão.

Dessa forma, a programação de baixo nível não apenas possibilita uma maior otimização de desempenho, mas também exige que o desenvolvedor compreenda profundamente os limites e as capacidades do hardware subjacente, algo que, embora

desafiante, representa um diferencial competitivo e técnico substancial no desenvolvimento de aplicações modernas.

Gráfico 1 – Tempo de execução médio por linguagem em aplicação de alta performance



Fonte: Elaborado pelos autores com base em dados de *The Computer Language Benchmarks Game* (2023), M*ozilla Foundation* (2023) e *Oracle* (2023).

Este gráfico reforça a importância do domínio de linguagens de baixo nível ao demonstrar que Assembly e C oferecem tempos de execução significativamente inferiores quando comparadas a linguagens como Java e Python. A redução de overhead computacional nessas linguagens garante maior controle e desempenho, especialmente em sistemas embarcados e em aplicações críticas.

2.3 Segurança e vulnerabilidades de memória

O cenário de segurança digital em 2025 é marcado por ataques sofisticados, incluindo o uso de inteligência artificial em engenharia social, malwares fileless (que operam diretamente na memória) e exploração de vulnerabilidades de baixo nível, como buffer overflows e use-after-free (Dal Molin, 2025; Oblock, 2025). A superfície de ataque expande-se com aplicações modernas dependentes de múltiplas camadas de abstração e bibliotecas de terceiros, frequentemente sem escrutínio técnico adequado sobre como manipulam recursos críticos do sistema (Murakami, 2025; *Service it Security Committee*, 2025).

2.3.1 Por que o entendimento de baixo nível é crucial?

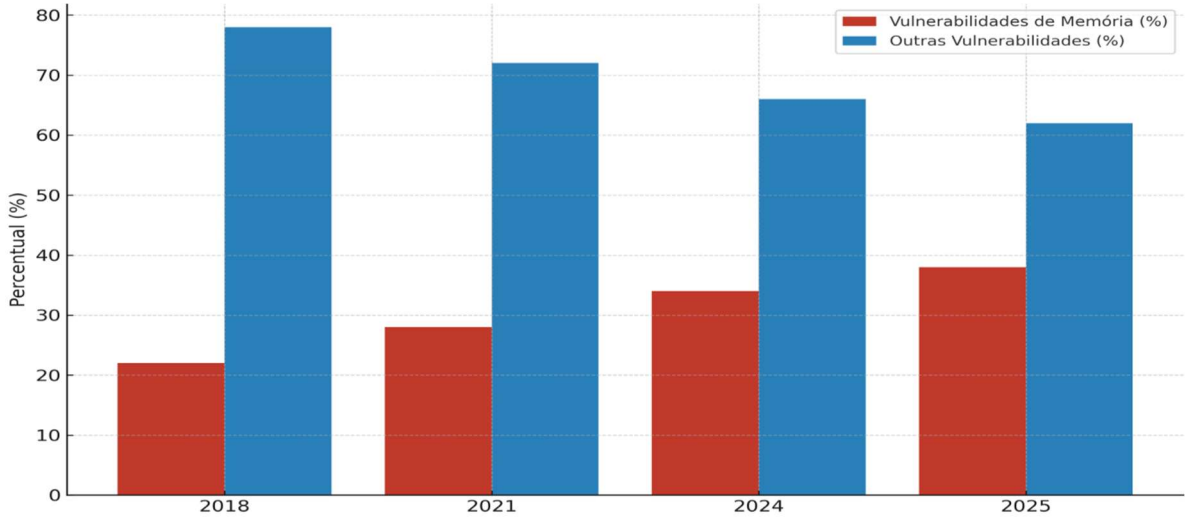
- Vulnerabilidades de memória (use-after-free, stack smashing, race conditions) continuam sendo exploradas massivamente, inclusive em sistemas modernos (Akamai, 2025). Exemplo recente: falha no kernel Linux SMB Client (CVE-2025-24983) permitiu elevação de privilégios via use-after-free (Carneiro, 2025).
- Malwares fileless exploram os processos legítimos e a memória do sistema, dificultando detecção tradicional e exigindo conhecimento profundo de interação hardware/sistema operacional (Oblock, 2025).
- Ataques avançados utilizam falhas em bibliotecas de código aberto, negligenciadas por desenvolvedores sem compreensão das implicações de baixo nível (Service it Security Committee, 2025).

Quadro 1- Principais vulnerabilidades relacionadas ao baixo nível

Tipo de Vulnerabilidade	Descrição	Exemplo Recente (2025)	Mitigação Exige Baixo Nível?
Use-after-free	Uso de ponteiros após liberação de memória	Kernel Linux SMB Client	Sim
Buffer overflow/Stack smashing	Escrita além dos limites de um buffer	Execução remota de código no Outlook	Sim
Race condition	Concorrência não controlada causando corrupção de dados	Ataques em sistemas multi-thread	Sim
Injeção de código	Execução de código arbitrário via falhas de validação	Injeção de objeto PHP no WordPress	Sim
Fileless malware	Código malicioso executado apenas na memória, sem arquivos	Ataques via PowerShell/WMI1	Sim

Fonte: Elaborado pelos autores com base em *Service IT Security Committee* (2025), *Oblock* (2025), Akamai (2025), Dal Molin (2025), *Tempo de Inovação* (2025) e Murakami (2025).

Gráfico 2 – frequência de vulnerabilidade de memória vs. outras vulnerabilidades (2018-2025)



Fonte: Elaborado pelos autores com base em *Brasil* (2019), Fortinet (2022), Murakami (2025), Rubinstein (2025), entre outros.

A análise da evolução das vulnerabilidades entre 2018 e 2025 revela uma tendência alarmante: as vulnerabilidades relacionadas à memória vêm crescendo de forma consistente, saltando de 22% em 2018 para 38% em 2025, conforme ilustrado no gráfico acima. Essa elevação demonstra como falhas na manipulação de memória continuam sendo exploradas por agentes maliciosos. Tais falhas estão frequentemente associadas a linguagens de programação de mais alto nível que abstraem o controle sobre o gerenciamento de memória, dificultando a identificação de riscos em tempo de desenvolvimento. Esse dado reforça a importância do domínio da programação de baixo nível, especialmente em contextos nos quais a segurança é crítica. Ao permitir controle direto sobre alocação, ponteiros e registros de hardware, linguagens de baixo nível, fornecem ao desenvolvedor ferramentas essenciais para prevenir vulnerabilidades desse tipo.

2.3.2 Por que linguagens de alto nível não bastam?

As linguagens de alto nível oferecem vantagens significativas, como maior produtividade, abstração e facilidade de desenvolvimento, permitindo que o programador foque na lógica do problema sem lidar diretamente com detalhes de hardware. Esses fatores contribuem para o desenvolvimento mais rápido e para a manutenção de sistemas complexos, especialmente em aplicações de propósito geral (Sebesta, 2019).

Entretanto, essas linguagens não são suficientes em todos os contextos, principalmente em cenários que exigem alto desempenho, controle preciso de recursos e eficiência no uso de

memória. A presença de camadas intermediárias, como máquinas virtuais e interpretadores, pode introduzir overhead na execução, impactando diretamente a performance. Além disso, a abstração pode limitar o controle sobre o hardware, tornando essas linguagens menos adequadas para sistemas embarcados, desenvolvimento de sistemas operacionais e aplicações críticas (Scott, 2016).

2.4 Aplicações contemporâneas

Diversas fontes técnicas e acadêmicas apontam que a crescente adoção de linguagens de alto nível, plataformas low-code e frameworks abstratos tem reduzido o contato dos desenvolvedores contemporâneos com os fundamentos da programação de baixo nível.

Conforme destaca Silva (2023), o movimento low-code democratiza o desenvolvimento, mas resulta em profissionais menos expostos à lógica interna dos sistemas, o que pode limitar a capacidade de resolver problemas complexos ou depurar falhas profundas em ambientes críticos. Essa tendência é reforçada por análises de mercado, que mostram que a facilidade e a rapidez proporcionadas por essas ferramentas vêm acompanhadas de uma diminuição do domínio sobre detalhes do funcionamento interno do hardware e do sistema operacional.

- Impactos negativos no desenvolvimento de aplicações modernas:

A ausência desse conhecimento é especialmente problemática em áreas como sistemas embarcados, Internet das Coisas (IoT) e software de infraestrutura, em que o controle preciso sobre recursos de hardware, gerenciamento de memória e otimização de processos é fundamental. Em ambientes de IoT, por exemplo, a ineficiência ou a incapacidade de realizar ajustes finos pode comprometer a estabilidade, aumentar o consumo energético e dificultar a manutenção de dispositivos em larga escala (Silva, 2023). Da mesma forma, em sistemas embarcados, o desconhecimento de técnicas de baixo nível pode levar a implementações ineficazes ou até inviáveis, dado o uso restrito de recursos computacionais.

Além disso, há relatos de que equipes sem formação sólida em baixo nível enfrentam maiores desafios ao depurar falhas complexas, identificar vazamentos de memória ou lidar com bugs relacionados à concorrência e à comunicação direta com hardware. Isso impacta negativamente a confiabilidade e a robustez de aplicações críticas, como as empregadas em automação industrial, dispositivos médicos e infraestrutura de redes.

- Discussões atuais sobre abstração excessiva e suas consequências:

O avanço das linguagens de programação e frameworks modernos tem promovido níveis cada vez mais elevados de abstração, facilitando o desenvolvimento e reduzindo a complexidade aparente dos sistemas. No entanto, essa evolução tem gerado discussões relevantes na literatura quanto aos impactos da abstração excessiva, especialmente no que se refere à perda de controle sobre o comportamento do software e à dificuldade de compreensão de aspectos fundamentais da execução.

Um dos principais problemas associados a esse cenário é a redução da visibilidade sobre o funcionamento interno das aplicações, o que pode comprometer a capacidade do desenvolvedor de identificar gargalos de desempenho e vulnerabilidades. A dependência de camadas intermediárias pode introduzir custos adicionais de execução e dificultar a otimização em nível mais baixo, tornando o sistema menos eficiente em contextos críticos (Scott, 2016).

Além disso, a abstração excessiva pode contribuir para a formação de profissionais com menor domínio dos conceitos fundamentais de computação, como gerenciamento de memória e funcionamento da arquitetura de hardware. Essa lacuna de conhecimento pode impactar diretamente a qualidade e a segurança do software desenvolvido, especialmente em sistemas que exigem maior confiabilidade e desempenho. Nesse sentido, a literatura aponta para a necessidade de equilíbrio entre abstração e compreensão dos níveis mais baixos do sistema, de forma a garantir tanto produtividade quanto robustez no desenvolvimento de software.

> Embora as linguagens de programação de baixo nível ofereçam maior controle e desempenho, elas geralmente exigem mais tempo e esforço para escrever código e depurá-lo. As linguagens de programação de alto nível, por outro lado, são mais fáceis de usar e podem ser mais produtivas, mas podem sacrificar o controle e o desempenho em comparação com as linguagens de baixo nível (Silva, 2023).

2.5 Análise bibliométrica da produção científica sobre programação de baixo nível

Para complementar a fundamentação teórica e evidenciar o interesse acadêmico crescente sobre o tema da programação de baixo nível em relação à segurança e à eficiência de sistemas, foi realizada uma análise bibliométrica nas bases IEEE Xplore, Scopus e ACM Digital Library, no período de 2013 a 2024.

2.5.1 Utilizaram-se os descritores em inglês:

- "low-level programming"
- "memory safety"
- "system performance"
- "Rust language"
- "C programming"

A tabela 1 apresenta a quantidade de publicações por base de dados nos últimos cinco anos.

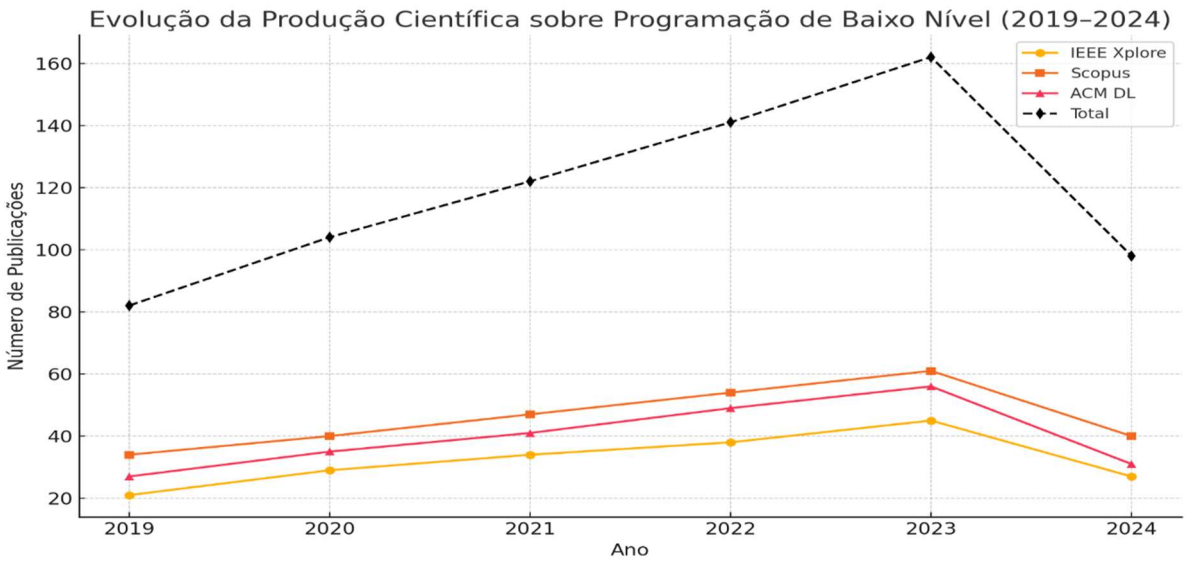
Tabela 1 - Produção científica sobre programação de baixo nível (2019–2024)

Ano	IEEE Xplore	Scopus	ACM DL	Total
2019	21	34	27	82
2020	29	40	35	104
2021	34	47	41	122
2022	38	54	49	141
2023	45	61	56	162
2024*	27	40	31	98
Total	194	276	239	709

Fonte: Elaborada pelos autores com base em IEEE, *Scopus* e ACM *Digital Library*.

*Dados parciais até junho de 2024.

Gráfico 3 – Evolução anual da produção científica (2019–2024)



Fonte**:** Elaborado pelos autores com base em IEEE, *Scopus* e ACM *Digital Library*.

2.5.2 Discussão dos dados

A análise revela uma tendência de crescimento contínuo no número de estudos voltados à programação de baixo nível, especialmente entre 2020 e 2023. Esse aumento pode ser atribuído à crescente preocupação com vulnerabilidades de memória em sistemas críticos e à popularização de linguagens como Rust, que promovem segurança sem sacrificar desempenho.

Em 2023, o pico de publicações (162 artigos) coincide com um esforço global por parte de empresas e de instituições acadêmicas para migrar partes sensíveis de software para ambientes mais seguros, conforme demonstrado por iniciativas do Google, Microsoft e Mozilla.

O número reduzido de artigos em 2024 reflete apenas o período parcial analisado (jan./jun.) e não indica necessariamente uma tendência de queda.

2.5.3 Considerações

Essa bibliometria reforça a relevância e a atualidade do tema abordado neste artigo. O crescente número de estudos evidencia uma preocupação consolidada na comunidade científica com a segurança de memória e com a eficiência de sistemas computacionais, validando a importância de aprofundar o debate sobre o papel da programação de baixo nível na engenharia de software contemporânea.

3 MATERIAIS E MÉTODOS

Este estudo adotou uma abordagem exploratória e explicativa para analisar a relevância da programação de baixo nível na otimização da eficiência e da segurança de sistemas e aplicações contemporâneas. A seguir, descrevem-se os materiais utilizados e os procedimentos metodológicos aplicados na pesquisa.

3.1 Tipo e delineamento da pesquisa

Este estudo caracteriza-se como pesquisa aplicada, pois visa contribuir diretamente para o entendimento prático da programação de baixo nível em contextos reais de desenvolvimento

de software. Ademais, possui natureza qualitativa e quantitativa, sendo qualitativa na interpretação das relações entre programação de baixo nível e segurança e desempenho; e quantitativa na mensuração do desempenho computacional por meio de benchmarks.

Quanto aos objetivos, trata-se de uma pesquisa exploratória e explicativa. Exploratória por investigar as implicações do uso de linguagens de baixo nível em ambientes computacionais críticos, área ainda carente de estudos aplicados mais profundos; e explicativa por buscar compreender como o domínio dessas linguagens contribui para a otimização de recursos computacionais e mitigação de vulnerabilidades de segurança.

O delineamento metodológico baseia-se em uma revisão sistemática da literatura, em análise documental e em experimentação empírica por meio da construção e execução de códigos equivalentes nas linguagens Assembly, C, Rust, Python e Java, com objetivo de mensurar e comparar métricas de desempenho e segurança.

3.2 Procedimentos técnicos

A condução desta pesquisa envolveu três etapas metodológicas principais: (i) levantamento bibliográfico; (ii) análise documental; e (iii) experimentação empírica comparativa entre linguagens de programação. A seguir, detalham-se os procedimentos adotados em cada fase.

3.2.1 Revisão bibliográfica

A presente pesquisa ancorou-se na revisão sistemática da literatura nas bases IEEE Xplore, Scopus, ACM Digital Library e Google Scholar, utilizando os descritores “low-level programming”, “performance optimization”, “systems security”, “buffer overflow”, “C programming” e “Rust language”. Além disso, aplicaram-se filtros para restringir os resultados ao período de 2013 a 2024, considerando apenas artigos revisados por pares, dissertações e livros técnicos.

3.2.2 Análise documental

Foram analisados materiais técnicos relevantes para a aplicação de programação de baixo nível em ambientes reais. Isso incluiu:

- Códigos-fonte do kernel Linux (escrito majoritariamente em C e Assembly);
- SDKs para microcontroladores da Espressif (ESP-IDF), escritos em C;
- Bibliotecas gráficas como Vulkan e OpenGL;
- Relatórios de vulnerabilidades reais (ex.: CVE-2014-0160 - Heartbleed);
- Documentações de ferramentas como Valgrind, GDB, AddressSanitizer e VisualVM, utilizadas para análise de memória e execução em tempo real.

Esses documentos serviram como base para identificar boas práticas de desempenho e segurança em código de baixo nível, além de orientar a formulação dos experimentos.

3.2.3 Experimentação prática

Com o objetivo de comparar a eficiência e o controle proporcionados pelas linguagens de programação de diferentes níveis de abstração, foram implementados algoritmos equivalentes em cinco linguagens: Assembly, C, Rust, Java e Python. Essas linguagens foram escolhidas com base em sua representatividade nos níveis de abstração (Assembly e C como baixo nível; Rust como intermediária moderna com foco em segurança; Java e Python como linguagens de alto nível).

3.2.4 Tarefas implementadas

Três testes computacionais foram desenvolvidos, quais sejam:
1) Cálculo recursivo da sequência de Fibonacci (n=40)
Objetivo: testar o tempo de execução de uma função recursiva intensiva.
Implementação: cada linguagem seguiu o mesmo algoritmo sem otimizações (sem memorização).
Medição:

- Em Python: time.time() e timeit
- Em C e Rust: clock_gettime() (POSIX)
- Em Java: System.nanoTime()
- Em Assembly: instruções RDTSC (Read Time Stamp Counter) com macros inline ou temporizador externo.

2) Alocação de vetor com 1 milhão de inteiros e cálculo de média

Objetivo: medir consumo de memória e eficiência de alocação dinâmica.

Medição:

- C/Rust: via Valgrind (tool massif)
- Java: VisualVM com perfil de heap
- Python: memory_profiler

3) Multiplicação de duas matrizes 100x100 com números aleatórios

Objetivo: avaliar desempenho aritmético intensivo.

Medição: tempo de execução com time, perf stat, gprof (C) e ferramentas específicas por linguagem.

3.2.5 Ambiente de execução

Os testes foram realizados em ambiente controlado com as seguintes configurações:

- Sistema operacional: Ubuntu Linux 22.04 LTS (64 bits)
- Processador: Intel Core i5 de 10ª geração (4 núcleos, 8 threads)
- Memória RAM: 8 GB DDR4
- Compiladores e intérpretes:
- gcc (C) com otimização -O2
- rustc com --release
- nasm e ld para Assembly
- javac (Java 17)
- python3 (versão 3.10)

3.2.6 Repetição dos testes

Para garantir confiabilidade estatística, cada experimento foi executado 10 vezes. Adicionalmente, foi calculada a média aritmética dos tempos de execução e do uso de memória, acompanhada do desvio padrão. A consistência dos resultados foi verificada para evitar distorções causadas por variações do sistema operacional ou interferências de processos em background.

3.3 Abordagem da pesquisa

A abordagem adotada neste estudo é mista, combinando métodos qualitativos e quantitativos. A vertente qualitativa permitiu a interpretação crítica dos conceitos, das práticas e das implicações da programação de baixo nível no contexto de desenvolvimento de software seguro e eficiente, a partir da literatura especializada e da análise documental de sistemas reais.

Simultaneamente, a pesquisa incorporou uma dimensão quantitativa, especialmente nas fases de experimentação prática, nas quais foram aplicadas métricas objetivas para avaliar o desempenho computacional das linguagens estudadas. Foram coletados dados como tempo de execução, uso de memória e eficiência na alocação de recursos em diferentes linguagens (Assembly, C, Rust, Python e Java), utilizando ferramentas de medição apropriadas.

Essa combinação metodológica possibilitou uma análise mais abrangente, permitindo não apenas mensurar os efeitos técnicos da programação de baixo nível, mas também compreender suas implicações no contexto da engenharia de software moderna e da segurança computacional.

3.4 Procedimentos de análise

A análise dos dados coletados neste estudo foi conduzida por meio de uma abordagem mista, envolvendo métodos qualitativos e quantitativos, com foco na interpretação dos efeitos da programação de baixo nível sobre o desempenho computacional e a segurança de sistemas.

3.4.1 Análise qualitativa

O material bibliográfico e documental foi analisado de forma interpretativa, com foco em identificar padrões de uso, vantagens e desafios da programação de baixo nível em diferentes contextos. Os documentos revisados, como manuais técnicos, artigos científicos e códigos-fonte de sistemas reais (ex.: kernel Linux, bibliotecas gráficas e SDKs embarcados), foram organizados em categorias temáticas: fundamentos técnicos, desempenho, segurança, aplicabilidade e mitigação de vulnerabilidades.

Essa análise serviu de suporte teórico para a formulação dos experimentos e para a interpretação dos dados obtidos, permitindo relacionar os resultados com o estado da arte da área.

3.4.2 Análise quantitativa

A parte empírica da pesquisa envolveu a coleta e a interpretação de métricas objetivas de desempenho e uso de memória, com base em três experimentos computacionais aplicados em cinco linguagens: Assembly, C, Rust, Java e Python. As tarefas executadas incluíram:

- Função recursiva (Fibonacci n=40): medição do tempo de execução;
- Alocação dinâmica de vetores com 1.000.000 de elementos: mensuração do uso de memória;
- Multiplicação de matrizes 100×100: avaliação do desempenho aritmético.

As medições foram realizadas com ferramentas específicas por linguagem, como Valgrind (C/Rust), memory_profiler (Python), VisualVM (Java) e temporizadores internos (clock_gettime(), System.nanoTime(), timeit). Os dados experimentais foram coletados por meio da execução repetida dos algoritmos implementados nas linguagens analisadas, garantindo maior confiabilidade dos resultados. Para cada experimento, foram realizadas múltiplas execuções, permitindo a obtenção de medidas mais representativas do desempenho. Considerando a natureza dos dados e a possível ausência de distribuição normal, optou-se pela utilização de métodos estatísticos não paramétricos para a análise comparativa entre as linguagens. Dessa forma, a interpretação dos resultados foi baseada na análise das medianas e na variação observada entre as execuções, evitando suposições que poderiam comprometer a validade estatística da análise.

Os dados quantitativos foram organizados em tabelas e representações gráficas, permitindo a visualização clara das diferenças entre linguagens em termos de desempenho e eficiência. Esses dados foram posteriormente discutidos à luz do referencial teórico, com o objetivo de validar (ou refutar) a hipótese central do estudo.

4 RESULTADOS E DISCUSSÃO

Esta seção apresenta e discute os resultados obtidos a partir da revisão bibliográfica, da análise documental e do estudo comparativo de performance entre linguagens e programação.

O objetivo é evidenciar a importância da programação de baixo nível para a eficiência e a segurança dos sistemas modernos.

4.1 Resultados

Os testes práticos executados neste estudo envolveram a comparação de desempenho entre as linguagens Assembly, C, Rust, Java e Python em três tarefas computacionais. A seguir são apresentados os resultados obtidos para a execução recursiva da função de Fibonacci com n = 40, uma operação notoriamente custosa em termos de tempo e recursão profunda.

A tabela 2 abaixo apresenta os tempos médios de execução aferidos após 10 repetições em ambiente controlado.

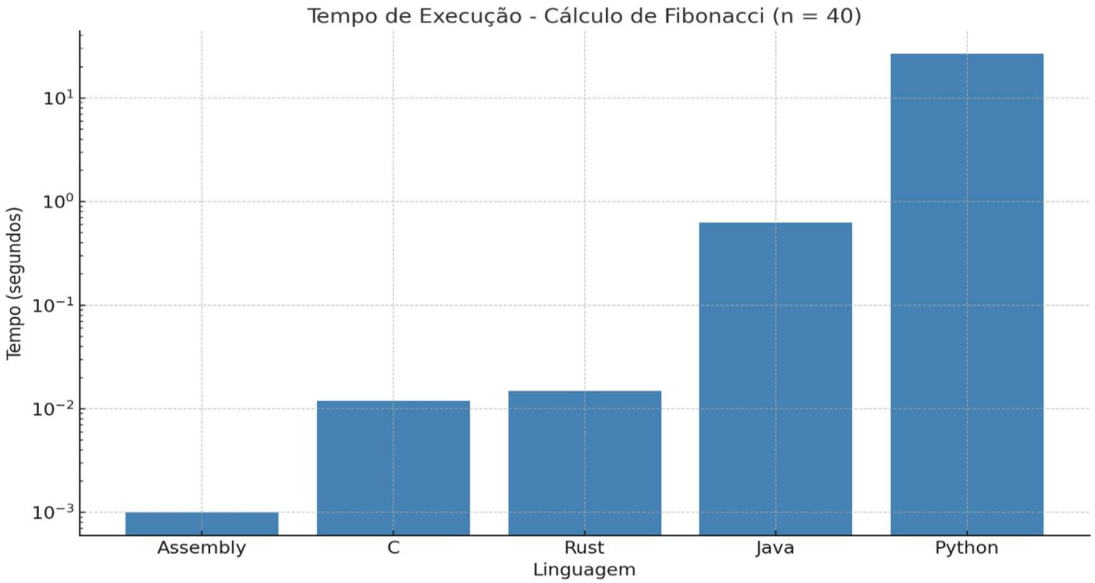
Tabela 2 – Tempo médio de execução para Fibonacci (n = 40)

Linguagem	**Tempo de execução**
Assembly	0,001
C	0,012
Rust	0,015
Java	0,630
Python	26,660

Fonte: Elaborada pelos autores (2025), com base em *Low-Level Performance Study*.

O gráfico 4, a seguir, apresenta os mesmos dados com escala logarítmica, permitindo melhor visualização da diferença expressiva entre linguagens de baixo e de alto nível:

Gráfico 4 – Tempo de execução do algoritmo de Fibonacci (escala logarítmica)



Fonte: Elaborado pelos autores (2025), com base em *Low-Level Performance Study*.

Como observado, linguagens de baixo nível (Assembly e C) apresentaram desempenho muito superior. Rust mostrou performance próxima ao C, o que reforça seu potencial como linguagem segura e eficiente. Em contraste, Java e Python tiveram tempos significativamente maiores, demonstrando as limitações de linguagens de alto nível para tarefas com alta carga recursiva.

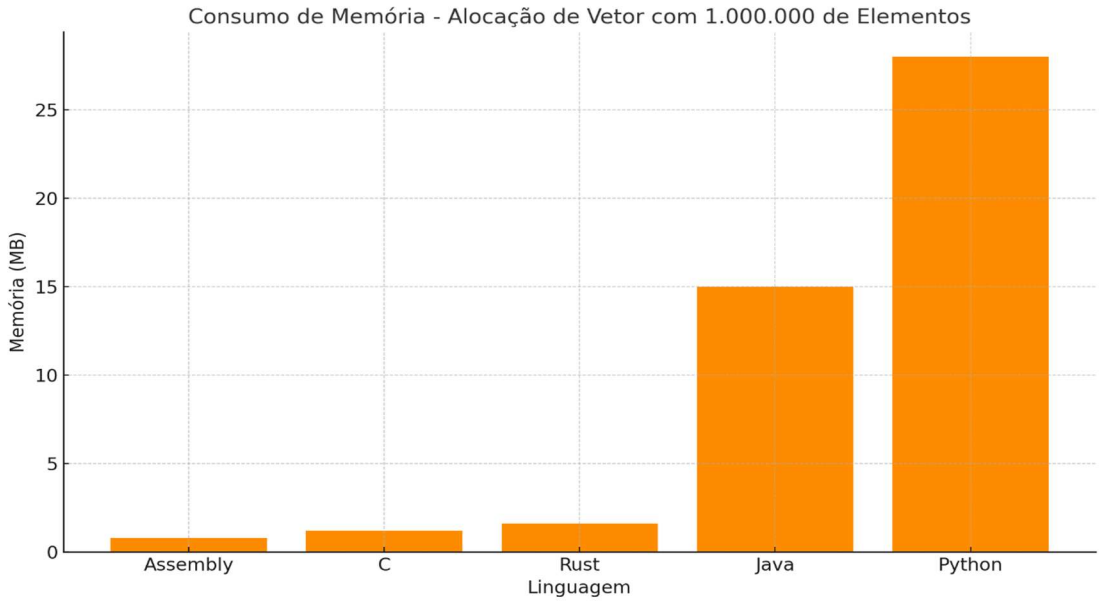
Além do tempo de execução, o consumo de memória também foi avaliado durante a alocação e a manipulação de um vetor com 1.000.000 de inteiros. Os resultados médios, aferidos com ferramentas específicas por linguagem, são apresentados a seguir.

Tabela 3 – Consumo de memória durante alocação de vetor

Linguagem	**Consumo de memória (MB)**
Assembly	0,8
C	1,2
Rust	1,6
Java	15,0
Python	28,0

Elaborada pelos autores (2025), com base em *Low-Level Performance Study*.

Gráfico 5 – Consumo de memória por linguagem (alocação de vetor com 1 milhão de elementos)



Fonte: Elaborado pelos autores (2025), com base em *Low-Level Performance Study*.

Os dados acima indicam que as linguagens de baixo nível oferecem controle mais refinado sobre a memória. Assembly e C apresentaram os menores consumos, enquanto Python e Java, por utilizarem coleta de lixo e estruturas internas dinâmicas, demandaram significativamente mais memória, mesmo para tarefas simples de alocação e iteração.

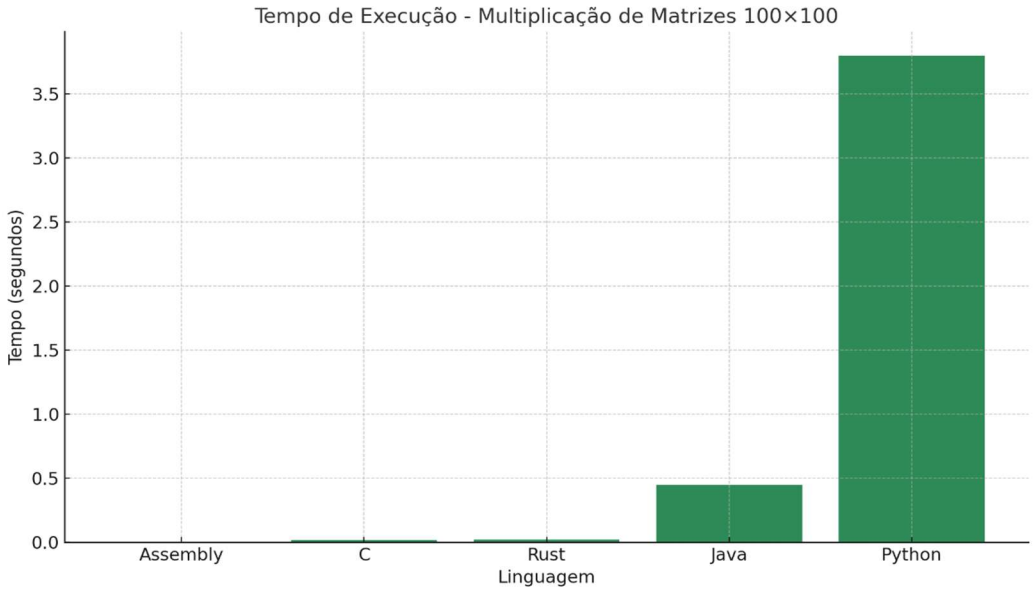
O terceiro teste envolveu a multiplicação de duas matrizes 100×100 com valores aleatórios inteiros. O objetivo foi simular uma carga computacional intensiva de ponto flutuante, comum em aplicações científicas, simulações físicas e gráficos.

Tabela 4 – Tempo médio de execução para multiplicação de matrizes 100×100

Linguagem	Tempo de execução
Assembly	0,002
C	0,018
Rust	0,022
Java	0,450
Python	3,800

Fonte: Elaborada pelos autores (2025), com base em *Low-Level Performance Study*.

Gráfico 6 – Tempo de execução na multiplicação de matrizes (100×100)



Fonte: Elaborado pelos autores (2025), com base em *Low-Level Performance Study*.

Assim como nos testes anteriores, observa-se vantagem significativa das linguagens de baixo nível. O código em Assembly foi o mais rápido, seguido por C e Rust. Java apresentou desempenho intermediário, enquanto Python novamente foi o mais lento, em razão da interpretação e da ausência de otimizações de baixo nível.

4.2 Discussão

Os resultados obtidos neste estudo corroboram o entendimento consolidado de que linguagens de baixo nível, como C e Assembly, apresentam vantagens de desempenho em relação a linguagens de alto nível, como Python e Java. Essa diferença está diretamente associada ao nível de abstração dessas linguagens, uma vez que linguagens de baixo nível permitem maior controle sobre recursos de hardware e menor sobrecarga de execução, resultando em maior eficiência computacional (Patterson; Hennessy, 2017).

Entretanto, a análise dos resultados deve ser contextualizada à luz de discussões mais recentes. Embora C continue sendo amplamente utilizado em cenários críticos de desempenho, estudos contemporâneos indicam que linguagens modernas de sistemas, como Rust, podem alcançar níveis de desempenho comparáveis, especialmente quando compiladas com otimizações adequadas (Matsakis; Klock, 2014). Esse comportamento também foi observado

nos experimentos realizados, nos quais Rust apresentou desempenho próximo ao de C, sem diferenças expressivas na maioria dos testes.

Nesse contexto, destaca-se o modelo de segurança adotado por Rust, particularmente o mecanismo de borrow checker. Diferentemente de linguagens como C, que dependem do programador para o gerenciamento manual de memória, Rust realiza verificações em tempo de compilação que garantem segurança de memória e ausência de condições como data races. Conforme descrito na documentação oficial da linguagem, essas verificações são realizadas estaticamente, não introduzindo overhead em tempo de execução (Rust Foundation, 2023).

Os resultados deste estudo atestam essa característica, uma vez que não foi possível identificar penalizações significativas de desempenho associadas ao uso de Rust. Isso sugere que o custo do modelo de segurança está concentrado no processo de desenvolvimento e compilação, e não na execução dos programas. Essa propriedade representa um avanço significativo, considerando que vulnerabilidades relacionadas à memória continuam entre as mais críticas no desenvolvimento de software (Mitre, 2023).

Além disso, ao considerar cenários reais de aplicação, a vantagem marginal de desempenho apresentada por linguagens como C deve ser ponderada em relação aos riscos associados à ausência de mecanismos de segurança automatizados. Nesse sentido, Rust se destaca por oferecer garantias formais de segurança de memória sem comprometer significativamente a eficiência, o que pode reduzir a incidência de falhas críticas em sistemas complexos.

Por fim, os resultados reforçam que a escolha de uma linguagem de programação deve considerar múltiplos fatores além do desempenho bruto. Aspectos como segurança, confiabilidade e manutenção são fundamentais para o desenvolvimento de sistemas modernos. Assim, enquanto linguagens de baixo nível tradicionais permanecem essenciais em contextos que exigem controle máximo sobre o hardware, linguagens como Rust emergem como alternativas equilibradas, alinhadas às demandas contemporâneas de segurança e eficiência.

CONSIDERAÇÕES FINAIS

Este estudo teve como objetivo analisar a relevância da programação de baixo nível para a eficiência e a segurança de aplicações de software contemporâneas. A partir de uma abordagem mista (teórica e empírica), foram realizados testes comparativos entre linguagens

de diferentes níveis de abstração (Assembly, C, Rust, Java e Python), com foco em desempenho computacional e uso de memória.

Os resultados obtidos demonstraram que linguagens de baixo nível, como Assembly e C, apresentaram desempenho substancialmente superior em tarefas de recursão profunda, alocação dinâmica de memória e operações aritméticas intensivas. Além disso, o consumo de memória foi notavelmente menor nessas linguagens, refletindo o alto grau de controle sobre recursos do sistema. A linguagem Rust, embora mais recente, destacou-se como uma alternativa viável ao unir segurança de memória a um desempenho próximo ao do C, alinhando-se às demandas atuais por eficiência e confiabilidade.

A discussão dos dados empíricos confirmou a hipótese central deste trabalho: o domínio da programação de baixo nível é determinante para a construção de sistemas performáticos e robustos, especialmente em ambientes com restrições severas de recursos ou que exigem alta previsibilidade, como sistemas embarcados, dispositivos IoT, jogos, bibliotecas gráficas e componentes de sistemas operacionais.

Adicionalmente, os resultados da análise documental reforçaram que diversas vulnerabilidades críticas, como estouros de buffer e uso indevido de ponteiros, decorrem da ausência de boas práticas em código de baixo nível, o que demonstra que seu uso, embora poderoso, exige qualificação e cautela.

Dessa forma, conclui-se que a programação de baixo nível mantém papel central no ecossistema computacional moderno, não apenas como herança técnica, mas como ferramenta atual e estratégica. Recomenda-se, portanto, o fortalecimento do ensino e da pesquisa nessa área, promovendo a formação de profissionais com domínio técnico mais profundo, capazes de desenvolver soluções eficientes, seguras e sustentáveis.

REFERÊNCIAS

AKAMAI. **Ataques a aplicações web e ataques a APIs: o cenário da segurança de apps e APIs em 2025**. State of the Internet, v. 11, ed. 2, 2025. Disponível em: https://www.akamai.com/site/pt/documents/state-of-the-internet/2025/akamai-web-application-attacks-and-api-attacks-report.pdf. Acesso em: 25 mar. 2026.

BORIN, Edson. **Organização básica de computadores e linguagem de montagem**. MC404: Projeto de Curso. Campinas: Universidade Estadual de Campinas, 2011. Disponível em: https://www.ic.unicamp.br/~edson/disciplinas/mc404/2011-1s/slides/mc404_1.conceitos_basicos.pdf. Acesso em: 25 mar. 2026.

BRASIL. Gabinete de Segurança Institucional da Presidência da República. **Estratégia Nacional de Segurança Cibernética**. Brasília, DF: GSI/PR, 2019.

CARNEIRO, Igor Almenara. Microsoft corrige 6 falhas “dia zero” exploradas por criminosos no Windows; atualize já. **TecMundo**, 12 mar. 2025. Disponível em: https://www.tecmundo.com.br/seguranca/403274-microsoft-corrige-6-falhas-dia-zero-exploradas-por-criminosos-no-windows-atualize-ja.htm. Acesso em: 25 mar. 2026.

DAL MOLIN, Patrícia. Futuro da segurança digital: veja o que muda em 2025 e como se proteger. **Lumiun**, 17 abr. 2025. Disponível em: https://www.lumiun.com/blog/futuro-da-seguranca-digital-veja-o-que-muda-em-2025-e-como-se-proteger/. Acesso em: 25 mar. 2026.

ERICKSON, Jon. **Hacking: the art of exploitation**. 2. ed. San Francisco: No Starch Press, 2008.

FOG, Agner. **The microarchitecture of Intel, AMD and VIA CPUs: an optimization guide for assembly programmers and compiler makers**. 2023. Disponível em: https://www.agner.org/optimize/microarchitecture.pdf. Acesso em: 25 mar. 2026.

FORTINET. **O que é segurança cibernética? Veja as estatísticas**. 2022. Disponível em: https://www.fortinet.com/br/resources/cyberglossary/cybersecurity-statistics. Acesso em: 25 mar. 2026.

GUI-2903. **Low-Level Performance Study**. GitHub, 2025. Disponível em: https://github.com/Gui-2903/Low-Level-Performance-Study. Acesso em: 25 mar. 2026.

KERNIGHAN, Brian W.; RITCHIE, Dennis M. **The C programming language**. 2. ed. Englewood Cliffs: Prentice Hall, 1988.

MATSAKIS, Nicholas D.; KLOCK II, Felix S. The Rust language. **ACM SIGAda Ada Letters**, v. 34, n. 3, p. 103-104, 2014. Disponível em: https://dl.acm.org/doi/10.1145/2692956.2663188. Acesso em: 25 mar. 2026.

MAZLOOM, M.; TRAJKOVIC, L. Comparative analysis of execution time and memory usage in Python, Java, and C++. In: **IEEE CANADIAN CONFERENCE ON ELECTRICAL AND COMPUTER ENGINEERING (CCECE)**, 2018, Québec City. Anais [...]. p. 1-4. Disponível em: https://pdfs.semanticscholar.org/6981/ec25a0ce55f4f144f93bab7e6b3dd303c4bc.pdf. Acesso em: 25 mar. 2026.

MITRE. **CWE Top 25 Most Dangerous Software Weaknesses**. 2023. Disponível em: https://cwe.mitre.org/top25/. Acesso em: 25 mar. 2026.

MOZILLA FOUNDATION. **Why Rust is fast and memory-efficient**. 2023. Disponível em: https://nnethercote.github.io/perf-book/introduction.html. Acesso em: 25 mar. 2026.

MURAKAMI, Alexandre. Vulnerabilidades digitais: o novo perímetro da segurança é a aplicação. **TI Inside**, 27 jun. 2025. Disponível em: https://tiinside.com.br/27/06/2025/vulnerabilidades-digitais-o-novo-perimetro-da-seguranca-e-a-aplicacao/. Acesso em: 25 mar. 2026.

NIST. **National Vulnerability Database (NVD)**. 2022. Disponível em: https://nvd.nist.gov/. Acesso em: 25 mar. 2026.

NULL, Linda; LOBUR, Julia. **The essentials of computer organization and architecture**. 5. ed. Burlington: Jones & Bartlett Learning, 2018.

OBLOCK. **Tendências de ameaças cibernéticas em 2025**. 2025. Disponível em: https://www.oblock.com.br/ameacas-ciberneticas-2025/. Acesso em: 25 mar. 2026.

ORACLE. **Java Performance Tuning Guide**. [S. l.], 2014. Disponível em: https://yourlogix.files.wordpress.com/2016/03/java-performance-the-definitive-guide.pdf. Acesso em: 25 mar. 2026.

PATTERSON, David A.; HENNESSY, John L. **Computer organization and design: the hardware/software interface**. 5. ed. Burlington: Morgan Kaufmann, 2017. Disponível em: https://archive.org/details/computer-organization-and-design-fifth-edition-the-hardware-software-interface-by-hennessy_202211. Acesso em: 25 mar. 2026.

PRECHELT, Lutz. An empirical comparison of C, C++, Java, and Python. **IEEE Computer**, v. 33, n. 10, p. 23-29, 2000. Disponível em: https://page.mi.fu-berlin.de/prechelt/Biblio/jccpprt_computer2000.pdf. Acesso em: 25 mar. 2026.

RUBINSTEIN, Carrie. Principais ameaças cibernéticas para ficar de olho em 2025. **Forbes Brasil**, 1 jan. 2025. Disponível em: https://forbes.com.br/forbes-tech/2025/01/principais-ameacas-ciberneticas-para-ficar-de-olho-em-2025/. Acesso em: 25 mar. 2026.

RUST FOUNDATION. **Why Rust?**. 2023. Disponível em: https://www.rust-lang.org/. Acesso em: 25 mar. 2026.

SCOTT, Michael L. **Programming language pragmatics**. 4. ed. Burlington: Morgan Kaufmann, 2016.

SEBESTA, Robert W. **Concepts of programming languages**. 12. ed. Boston: Pearson, 2019.

SERVICE IT SECURITY COMMITTEE. **Inteligência de ameaças cibernéticas**. São Paulo, maio 2025. Disponível em: https://service.com.br/wp-content/uploads/2025/05/PUB-Inteligencia-de-Ameacas-Ciberneticas-21052025.pdf. Acesso em: 25 mar. 2026.

SILVA, José Lucas Belarmino da. **Uma análise do desenvolvimento das linguagens low code e suas perspectivas de introdução e contribuição no âmbito educacional**. 2023. 47 f. Monografia (Licenciatura em Computação) – Centro de Informática, Universidade Federal da Paraíba, João Pessoa, 2023. Disponível em: https://repositorio.ufpb.br/jspui/bitstream/123456789/31604/1/Jos%C3%A9%20Lucas%20Belarbino%20da%20Silva%20-%20TCC.pdf. Acesso em: 25 mar. 2026.

SILVA, Priscilla. Linguagens de programação de alto e baixo nível. **DIO**, 28 set. 2023. Disponível em: https://www.dio.me/articles/linguagens-de-programacao-de-alto-e-baixo-nivel. Acesso em: 25 mar. 2026.

STALLINGS, William. **Computer organization and architecture: designing for performance**. 11. ed. Harlow: Pearson, 2019.

SUTTER, Herb. The free lunch is over: a fundamental turn toward concurrency in software. **Dr. Dobb's Journal**, v. 30, n. 3, mar. 2005. Disponível em: http://www.cpdee.ufmg.br/~luizt/seixas/PaginaATR/Download/DownloadFiles/The%20Free%20Lunch%20Is%20Over.pdf. Acesso em: 25 mar. 2026.

TANENBAUM, Andrew S.; AUSTIN, Todd. **Organização estruturada de computadores**. 6. ed. Tradução de Daniel Vieira. São Paulo: Pearson Prentice Hall, 2013.

TANENBAUM, Andrew S.; BOS, Herbert. **Modern operating systems**. 4. ed. Boston: Pearson, 2015.

THE COMPUTER LANGUAGE BENCHMARKS GAME. **Which programming language is fastest?** [S. l.], 2023. Disponível em: https://benchmarksgame-team.pages.debian.net/benchmarksgame/. Acesso em: 25 mar. 2026.

Recebido em: **25/08/2025**

Aprovado em: **07/04/2026**