STM JOURNALS

Recent Trends in
Programming Languages

ISSN: 2455-1821
Volume 11, Issue 3, 2024
September–December
DOI (Journal): 10.37591/RTPL

https://journals.stmjournals.com/rtpl

Review RTPL

Comparative Analysis of Modern Programming Paradigms: Evaluating Language Efficiency and Compiler Design Technique

Shilpi Saxena[1], Ashish Singh[2], Vaibhav Rajendra Chaudhari[3], Jolly Pandey[4], Ashwin Singh[5], Mritunjay Kr. Ranjan[3,*]

Abstract

This paper attempts to provide some insights into the efficiency of modern programming paradigms via a comparative study and explore the important role played by compiler design in the optimization of these languages. Programming languages have been evolving quickly over time and different paradigms: imperative, functional, or object-oriented programming come with their idiosyncrasies and optimization techniques. The study starts by defining the foundational principles of each paradigm. It then goes on to perform an in-depth analysis of how these organizational ideas impact computer design and efficiency. A series of benchmark tests were conducted over several different but representative languages from each paradigm, comparing performance metrics such as execution speed, memory usage, and ease of optimization. These results clearly show big differences in efficiency, dependent on the programming paradigm and individual design decisions made inside of a compiler. Additionally, an investigation of what trade-offs are implied by language features and how they impact programmer productivity is conducted. The goal of this work is to generate insights supporting language designers, educators, and the developers themselves in their decisions on which programming paradigm best supports their requirements. In short, the results demonstrate that using appropriate high-level programming paradigms leveraged to specific project needs can effectively maximize both development and runtime efficiency.

Keywords: programming paradigms, language efficiency, compiler design, performance analysis, execution speed, memory optimization.

***Author for Correspondence**
Mritunjay Kr. Ranjan
E-mail: mritunjaykranjan@gmail.com

[1]Assistant Professor, Department of Computer Application and IT, Lords University, Rajasthan, India
[2]Student, School of Computer Sciences and Engineering, Sandip University, Nashik, Maharashtra, India
[3]Assistant Professor, School of Computer Sciences and Engineering, Sandip University, Nashik, Maharashtra, India
[4]Assistant Professor, Department of Information Technology, Gaya College, Gaya, Bihar, India
[5]Lecturer, Department of Information Technology, Sandip Polytechnic, Sandip Foundation, Nashik, Maharashtra, India

Received Date: October 22, 2024
Accepted Date: October 23, 2024
Published Date: November 05, 2024

Citation: Shilpi Saxena, Ashish Singh, Vaibhav Rajendra Chaudhari, Jolly Pandey, Ashwin Singh, Mritunjay Kr. Ranjan. Comparative Analysis of Modern Programming Paradigms: Evaluating Language Efficiency and Compiler Design Technique. Recent Trends in Programming Languages. 2024; 11(3): 1–9p.

INTRODUCTION

The development of programming languages has had a profound impact on the software world as many new and old paradigms have emerged to meet different needs in this field. A programming paradigm provides a style in which computer programmers work and think while constructing software [1]. At the highest level, we have three main paradigms: imperative, functional, and object-oriented programming, with different flavors of how to structure your code/your data/how you solve problems. One of the effects of using a particular programming paradigm is that it negatively changes the design and software organization, as well as the application performance and efficiency. Ever since the development of programming languages, it has been important to know how well or fast your paradigms are working because, in modern

development, performance constraints have a greater impact than ever [2]. The speed, memory usage, and responsiveness of software are important in web services, embedded systems, and any other type of application [3]. Therefore, the compilation and optimization processes are crucial for achieving the best performance of this class of high-level code that translates into machine-readable instruction using programs called compilers. Efficiency in execution is dependent on the underlying principles of a programming paradigm, which dictates compiler design techniques. One can relate programming paradigms of the modern day and compiler design techniques and contribute towards the efficiency analysis for a variant group of languages. By looking at the prevalent languages embodying each paradigm, the essence of optimizations improves software implementation performance [4]. For example, object-oriented languages generally choose encapsulation and inheritance to foster code reuse but incur the cost of execution speed. Functional programming languages, conversely, really emphasize immutability and higher-order functions that enable beautifully concise expressions for difficult problems but potentially hamper traditional compiler optimizations. To benchmark the analysis in a more complete way, we chose languages from different paradigms and performed some executions to reach better conclusions about execution speed versus memory consumption [5]. We hope that our systematic performance evaluation will show the trade-offs involved in each paradigm and how compiler features affect language efficiency. Second, this study explores the implications of this finding in computer science practice and education. Because the choice of programming paradigm greatly affects the development time, code maintainability, and application performance, a better understanding of these relationships will enable developers to make informed decisions when selecting their next or current programming style. In the end, this research can add to the ongoing discussion of programming language design and optimization so that programmers everywhere are more aware of how our choice in paradigms may change software development as a whole.

Objective

- Evaluate the quality and interplay of programming paradigms and their impact on design, organization, performance, and compiler design techniques in optimization for code execution, saving memory.
- Perform benchmarking to look for execution time and memory usage of different languages with various paradigms, which will help in performance considerations.
- Explore how programming paradigms are influenced in practice by their effects on development time, code maintainability, and application efficiency to help influence their choice of programming paradigms in future software projects.

RELATED WORK

The design of programming languages has been and probably will be a hot topic in the field known as "programming language theory." Although compiler technology has been heavily researched, language design is far less researched. This study targets Multiprotocol Label Switching (MPLs) and uses a set of modern programming languages for an application-specific view to be analyzed systematically. More specifically, the author evaluated their programming paradigms as well as type systems and language performances using benchmarking tests. Based on these theoretical and practical analyses, the relationship between the programming language and its application scenarios is interpreted; thus, it can be deduced to predict the developing trend of future programming languages [6].

In this essay, the author provides an overview of quantum programming languages and their corresponding runtimes and algorithmic modality development. Targeting the scientific high-performance computing (HPC) community, this paper describes the state-of-the-art programming paradigms useful for implementing a broad range of scientific workflows. A more important take-home lesson is that the author still has a long way to go in honing the concept of what it takes for quantum processing units to coexist with classical HPC environments. Although programming today's quantum computers is getting closer to a full-blown modern HPC-compatible workflow, new challenges still need to be solved in the field [7].

© STM Journals 2024. All Rights Reserved

Currently, cluster computing based on central processing unit, and GPU stands for graphics processing unit (CPU-GPU) has a domain of difficult and high-depth computation. The traditional programming paradigm is not compatible with the resource utilization of a cluster for effective utilization. Thus, this study focuses on performance parallel programming paradigms such as OpenMP for CPU clusters and Compute Unified Device Architecture (CUDA) for GPU clusters in Breadth First Search (BFS) and Depth-first search (DFS) graph algorithms. In the following section, the author analyzes how long it takes to pass through these graphs with a certain number of nodes in two different processors. A CPU based on OpenMP and GPU using CUDA Platform multi-thread processing is supported to obtain the result for each node. The experimental results obtained indicate that parallelization using the graph algorithm with the OpenMP programming model does not enhance CPU performance by distributing its work among cores but rather degrades it as a direct result of added overheads such as idling time and inter-thread communication, and even because idle threads perform calculations on their own. However, the CUDA parallel programming model on the GPU yielded better results. This is at a speed-up of 187 times compared to the CPU implementation using four cores and at 240 times compared with the serialized version [8].

Engineering education has changed over the decades, with new pedagogical practices and technologies being used to effectively educate students in a way that highlights societal impacts and sustainability. This change encompasses a change from what the author might consider in hindsight the arcane role of professors as sole arbiters of truth and wisdom to democratic environments with student-led, profess-supported educational experiences involving colleagues outside academia critical for creating genuinely transformational learning. During this process of transformation, different educational paradigms have been predominant (positivist and then constructivist); concurrently, others have emerged: the socio-critical paradigm and communicative critical. These models are often considered conflicting or incompatible [9].

Highlights: This paper details the cultivation and application of a new laptop-aided design (CAD) paradigm for a large-scale integrated circuit layout. This assimilation combines the aspects of good judgment synthesis, circuit optimization, and time versioning with established thoughts to provide a unified framework. This paradigm includes Gaussian-process-based machine learning techniques and convex programming used to optimize the component selection criteria, as well as layout placements. Subsequently, the constructed circuit was simulated, measured, and evaluated. The CAD framework was validated in the layout of an 8-bit ripple-convey adder system. The results offer strong evidence that this new CAD-based paradigm could be an efficient technique for designing cutting-edge VLSI systems [10].

METHODOLOGY

It discusses various compiler design techniques in detail using syntax analysis and code optimization methods. Parsing algorithms, such as Left to Left (LL) and Left to Right (LR) parsing, and optimization techniques in which there are approaches available for local and global optimizations to speed up code execution. It collects data using benchmark tests on various programming languages to compare performance metrics. All data were analyzed through statistical methods to provide mean execution time (MET) and mean memory usage (MMU) for easy comparisons [11, 12]. Finally, the results are visualized in charts and graphs, which help to provide better insights into the strengths and weaknesses of each programming paradigm; thus, compiler design can be considered more with future language implementation when it comes to making the most efficient code for software development.

S1: Selection of Programming Paradigms

1. Choose representative programming paradigms (e.g., procedural, object-oriented, functional, and logic programming).

S2: Performance Metrics

1. Define efficiency metrics for evaluation:
   i. *Execution time (ET):* Time taken to run a program.

© STM Journals 2024. All Rights Reserved

ii. *Memory usage (MU):* Amount of memory consumed during execution.
iii. *Code complexity (CC):* Measured using cyclomatic complexity; Equation (1),

$$CC = E - N + 2P \quad (1)$$

where:
- E = number of edges,
- N = number of nodes,
- P = number of connected components.

S3: Compiler Design Techniques
1. Analyze different compiler design techniques:
   i. *Syntax analysis:* Use parsing algorithms (e.g., LL, LR parsing).
   ii. *Optimization techniques: Evaluate:*
      - *Local optimization:* Improves code within basic blocks.
      - *Global optimization:* Optimizes across function boundaries.

S4: Data Collection
1. Implement benchmark tests across languages, recording performance metrics for each paradigm.

S5: Statistical Analysis
1. Apply statistical methods to analyze collected data:
   i. *Mean execution time (MET)*

$$MET = \frac{\sum_{i-1}^{n} ET_i}{n} \quad (2)$$

Where,
- *MET* stands for *mean execution time*.
- ET_i is the *execution time* of the *i-th* task or operation.
- n is the total number of tasks or operations.
- $\sum$ *(summation)* indicates the sum of execution times from 1 to n.

   ii. *Mean memory usage (MMU)*

$$MMU = \frac{\sum_{i-1}^{n} MU_i}{n} \quad (3)$$

S6: Comparative Evaluation
1. Visual representation (charts/graphs) was used to compare the efficiency of each paradigm based on metrics and to identify strengths and weaknesses.

This methodology ensures a systematic and quantitative approach for evaluating the efficiency of various programming paradigms and compiler design techniques (Figure 1). There are three main packages for this subject: programming paradigms, compiler techniques, and performance metrics [13]. Inside the programming paradigm package, we have imperative, functional, object-oriented, and concurrent programming, each influencing some part of the compiler design [14]. Imperative programming affects lexical analysis, functional programming affects syntax analysis, and object-oriented influences code generation. These points illustrate how various paradigms define compiler operations. A set of important processes, i.e., lexical analysis, syntax analysis, and code generation are provided by the compiler techniques package to convert high-level into machine language [15]. It illustrates the relationships between these techniques and how they pump into one another, eventually affecting the efficiency of the generated code. *Performance metrics:* The outcome of these processes is further evaluated using a performance metrics package that evaluates execution speed and memory usage, thereby demonstrating how application performance characteristics depend on the choice of programming paradigm and compiler design (in terms of high-level evaluation) [16].

© STM Journals 2024. All Rights Reserved

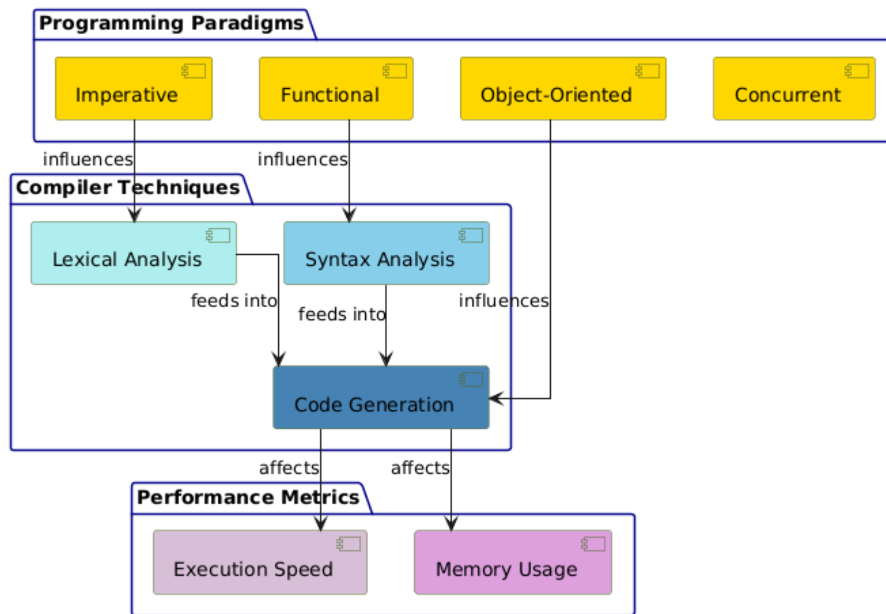

Figure 1. Influence of programming paradigms on compiler techniques and performance metrics.

Table 1. Algorithm approach.

```
Algorithm Comparative Analysis

// Step 1: Define programming paradigms and metrics
Input: Paradigms = {Imperative, Functional, Object-Oriented, Concurrent}
  Metrics = {Execution Speed, Memory Usage, Code Maintainability, Scalability}
  CompilerTechniques = {LexicalAnalysis, SyntaxAnalysis, CodeGeneration, Optimization}

// Step 2: Initialize results
Results = []

// Step 3: Begin analysis for each paradigm
For each paradigm in Paradigms do
  Initialize: EfficiencyScore = 0

  // Step 4: Evaluate compiler techniques for each paradigm
  For each technique in CompilerTechniques do
  If paradigm == "Imperative" then
  EfficiencyScore += EvaluateEfficiency(LexicalAnalysis, Metrics)
  Else if paradigm == "Functional" then
  EfficiencyScore += EvaluateEfficiency(SyntaxAnalysis, Metrics)
  Else if paradigm == "Object-Oriented" then
  EfficiencyScore += EvaluateEfficiency(CodeGeneration, Metrics)
  Else if paradigm == "Concurrent" then
  EfficiencyScore += EvaluateEfficiency(Optimization, Metrics)
  End if
  End For

  // Step 5: Add results to the list
  Append (paradigm, EfficiencyScore) to Results
End For

// Step 6: Compare and output best paradigm
BestParadigm = CompareEfficiency(Results)
Output "Best Programming Paradigm:", BestParadigm

End Algorithm
```

© STM Journals 2024. All Rights Reserved

The systematic breakdown of the complex combination of programming languages and compiler design makes this an essential publication for a wide audience in academia, industry, government enterprises, and NGOs who have anything to do with computer science, as shown in Table 1.

This algorithm for comparative analysis of modern programming paradigms: Evaluating language efficiency and compiler design techniques will systematically verify different variations in programming paradigms with the help of key performance metrics and compiler design techniques [17]. The process of language design usually starts with the specification, and this is brief documentation that tells you more about what kind of input will go into your final formal definition, such as paradigms (say imperative, functional or object-oriented programming), performance metrics to be optimized (contention on how fast our code could execute, amount memory used by running instances and created objects in memory at peak time point), and things related directly to the compiler(making a clear stage between lexical analysis and compilation Error Propagation until we get the most specific information we can provide back [18]. The algorithm loops over each paradigm and the performance of the corresponding compiler techniques is given the metrics [19]. Compiler directives assign a (possibly different) compiler technique for each paradigm with respect to the form of expressions expected in case statements (i.e., lexical analysis is associated with imperative programming, while syntax analysis is linked to functional programming, etc.). An Efficiency Score is calculated for each paradigm based on how well the paradigm optimizes the compiler performance metrics. Comparisons of all paradigms take place, and findings are derived with the help of comparison and then measures to select which paradigm will be considered more effective or better for overall performance and language efficiency. This method involves a systematic, contextually grounded comparison of paradigms that shows how the underlying compiler technologies and design decisions can make different languages run faster or slower on workloads, providing developers and researchers with guiding insights when selecting which paradigm is suitable for a given application.

SIMULATION PARAMETERS

Table 2 summarizes the parameters used for the evaluation of the various programming paradigms, compiler techniques, and system performance. It supports four programming paradigms: imperative, functional, object-oriented, and concurrent. The following compiler techniques were evaluated: lexical analysis, syntax analysis, code generation, and optimization. Some of the techniques and paradigms were performance tested using performance metrics, such as execution speed, memory usage, scalability, code maintainability, etc., to benchmark whether they were effective. We evaluated 10 programs for each paradigm and conducted 1000 iterations per test to ensure accuracy and consistency. The tests were run on a machine with a 3.4 GHz CPU, 16 GB RAM, and Linux OS, which provides context to our hardware (and software) for this analysis.

RESULT ANALYSIS (EFFICIENCY METRICS)

Table 3 summarizes the comparison of four paradigms—imperative, functional, object-oriented, and concurrent—with respect to execution speed, memory use properties, code maintainability, and scalability.

Table 2. Parameters for evaluating programming paradigms, compiler techniques, and system performance.

Parameter	Description	Values/Range
Programming paradigms	Types of programming paradigms evaluated	Imperative, functional, object-oriented, concurrent
Compiler techniques	Compiler processes evaluated	Lexical analysis, syntax analysis, code generation, optimization
Performance metrics	Criteria used to evaluate paradigms	Execution speed, memory usage, scalability, code maintainability
Test programs	Number of test programs for each paradigm	10
Iterations	Number of iterations per test	1000
System specifications	CPU, RAM, and OS used for testing	3.4 GHz CPU, 16 GB RAM, Linux OS

© STM Journals 2024. All Rights Reserved

Table 3. Performance comparison of programming paradigms across key metrics.

Paradigm	Execution speed (%)	Memory usage (%)	Code maintainability (%)	Scalability (%)
Imperative	85	70	65	80
Functional	75	60	85	70
Object-oriented	80	75	90	85
Concurrent	90	80	70	95

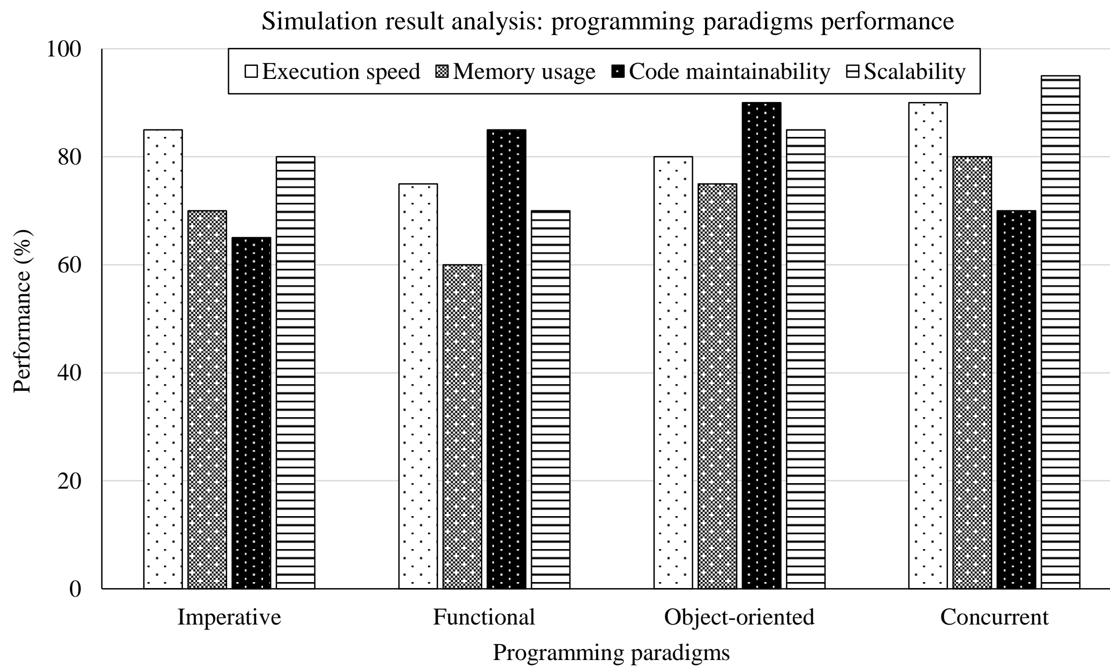

Figure 2. Simulation result analysis of programming paradigms performance across key metrics.

Imperative paradigm: (execution speed: 85%); (scalability: 80%), (maintainability: 65%). Functional programming has very high values for code maintainability (85%), moderately good results in speed of execution (75%), and poor-quality solutions with respect to memory usage (60% efficiency) and scalability. Better performance with high code maintainability (90%) and execution speed/memory usage/scalability are well balanced. The object-oriented paradigm, the concurrent paradigm, finally scores well in scalability (95%) and how fast it can execute code (90%) but is let down slightly by its maintainability of the associated scripts, tests, etc., availability only at 70%. Compared with other implementations, the concurrent paradigm has an edge in both speed and scalability, perhaps most notably improving concurrency as it should. Broadly speaking, I would say the best balance among object-oriented staff, etc.

Figure 2 displays the performance of four programming paradigms—imperative, functional, object-oriented, and concurrent—in terms of execution speed (benchmarked by means of time per call), memory usage (by factor against a common reference implementation), and code maintainability and scalability. Imperative, which is 85% execution speed and 80% scalability, but lower maintainability at only 65%. The functional paradigm is the best in terms of code maintainability (85%) but less rated for execution speed (75%), memory usage (60%), and scalability (70%). This is followed by Object-Oriented Programming (OOP), which partly strikes a balance with 80% execution speed, 75% memory requirements, reusability (90%, although fairly close to functional programming), and scalability (85%). Finally, the newer concurrent paradigm takes top marks for scalability (95%) and speed of execution (90%), but with a slightly lower maintainability score of 70%. The former is for speed and scalability, and the latter makes maintainability and good performance most likely in all areas.

© STM Journals 2024. All Rights Reserved

CONCLUSION

The difference in efficiency and compiler design across programming paradigms is a strong case for this comparative analysis of modern fast languages. This also shows that programming paradigms, such as imperative, functional, and object-oriented, have their own set of properties that impact the primary performance metrics such as execution speed, memory usage, and optimization potential. Benchmark trials showed distinctly different performances, suggesting that some paradigms are better suited to a particular sort of processing or differing system requirements. For example, the execution speed of an imperative paradigm and maintainability benefits from a functional one. This study calls attention to the fundamental trade-offs between high-level features associated with a particular programming paradigm, and how they affect programmer productivity and, ultimately, application efficiency. It also reinforces how compiler design impacts not only our ability to use these paradigms effectively but also further justifies the question of project-driven programming. These results provide useful insights for language designers, developers, and educators in choosing the appropriate paradigm according to the different requirements of a particular project. To conclude, this study provides an argument for further knowledge and utilization of programming paradigms to achieve advancements in both the development processes and runtime performance.

Future Work

In the future, the author can find the right balance by combining imperative, functional, and object-oriented paradigms in multiparadigm programming for performance while maintaining testability. This will be thoroughly explored. Moreover, efficiency can be improved with adaptive compilers that choose optimization strategies based on the programming paradigm and system requirements. Application Performance Management (APM) also has the potential to provide insights into the paradigm-specific performance of emerging technologies such as quantum computing and edge computing, where the need for new compiler designs along with paradigms can be expected to exploit their full capabilities. It may also be interesting to explore AI-driven code optimization tools across different paradigms. Similarly, increasing the scope of benchmarking trials (to encompass even more diverse and application-specific languages, for example, for embedded systems or scientific computing) would offer some indication as to which case each paradigm thrives best. This will allow for progress in both language and compiler design.

REFERENCES

1. Wiese ES, Rafferty AN, Kopta DM, Anderson JM. Replicating novices’ struggles with coding style. 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), Montreal, QC, Canada, 2019, pp. 13–8. DOI: 10.1109/ICPC.2019.00015.
2. Pereira R, Couto M, Ribeiro F, Rua R, Cunha J, Fernandes JP, Saraiva J. Ranking programming languages by energy efficiency. Sci Comput Program. 2021;205:102609. DOI: 10.1016/j.scico.2021.102609.
3. Weintrop D, Wilensky U. Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. Comput Educ. 2019;142:103646. DOI: 10.1016/j.compedu.2019.103646.
4. Ranjan MKr, Barot K, Khairnar V, Rawal V, Pimpalgaonkar A, Saxena S, et al. Python: Empowering data science applications and research. J Oper Syst Dev Trends. 2023;10:27–33. DOI: 10.37591/joosdt.v10i1.576.
5. Ghose S, Boroumand A, Kim JS, Gómez-Luna J, Mutlu O. Processing-in-memory: A workload-driven perspective. IBM J Res Dev. 2019;63:3:1–3:19. DOI: 10.1147/JRD.2019.2934048.
6. Wang X, Zhang Z. Analysis of the design of several modern programming languages. 2022 IEEE 2nd International Conference on Computer Systems (ICCS), Qingdao, China. 2022. pp. 40–4. DOI: 10.1109/ICCS56273.2022.9987746.
7. Lopez Alarcón SL, Wong E, Humble TS, Dumitrescu E. Quantum programming paradigms and description languages. Comput Sci Eng. 2023;25:33–8. DOI: 10.1109/MCSE.2024.3375432.

© STM Journals 2024. All Rights Reserved

8. Chandrashekhar BN, Sanjay HA, Srinivas T. Performance analysis of parallel programming paradigms on CPU-GPU clusters. 2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS), Coimbatore, India, 2021, pp. 646–51. DOI: 10.1109/ICAIS50930.2021.9395977.
9. Forcael E, Garcés G, Lantada AD. Convergence of educational paradigms into engineering education 5.0. 2023 World Engineering Education Forum - Global Engineering Deans Council (WEEF-GEDC), Monterrey, Mexico, 2023, pp. 1–8. DOI: 10.1109/WEEF-GEDC59520.2023.10344026.
10. Agarwal A, Gour MM. Establishing a novel CAD-based paradigm for design of VLSI integrated circuits. 2024 International Conference on Optimization Computing and Wireless Communication (ICOCWC), Debre Tabor, Ethiopia. 2024. pp. 1–6. DOI: 10.1109/ICOCWC60930.2024.10470486.
11. Jomaa N, Nowak D, Grimaud G, Hym S. Formal proof of dynamic memory isolation based on MMU. Sci Comput Program. 2018;162:76–92. DOI: 10.1016/j.scico.2017.06.012.
12. Pan R, Peach G, Ren Y, Parmer G. Predictable virtualization on memory protection unit-based microcontrollers. 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Porto, Portugal. 2018. pp. 62–74. DOI: 10.1109/RTAS.2018.00012.
13. Czarnul P, Proficz J, Drypczewski K. Survey of methodologies, approaches, and challenges in parallel programming using high-performance computing systems. Sci Program. 2020;2020:1–19. DOI: 10.1155/2020/4176794.
14. Dageförde JC, Kuchen H. A compiler and virtual machine for constraint-logic object-oriented programming with Muli. J Comput Lang. 2019;53:63–78. DOI: 10.1016/j.cola.2019.05.001.
15. Tan J, Jiao S, Chabbi M, Liu X. What every scientific programmer should know about compiler optimizations? Proceedings of the 34th ACM International Conference on Supercomputing (ICS '20); Association for Computing Machinery, New York, NY, USA. 2020. p. 42. DOI: 10.1145/3392717.3392754.
16. Peitek N, Apel S, Parnin C, Brechmann A, Siegmund J. Program comprehension and code complexity metrics: An fMRI study. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, ES. 2021. pp. 524–36. DOI: 10.1109/ICSE43902.2021.00056.
17. Thoman P, Dichev K, Heller T, Iakymchuk R, Aguilar X, Hasanov K, et al. A taxonomy of task-based parallel programming technologies for high-performance computing. J Supercomput. 2018;74:1422–34. DOI: 10.1007/s11227-018-2238-4.
18. Starr WB. A preference semantics for imperatives. Semantics Pragmat. 2020;13:1–60. DOI: 10.3765/sp.13.6.
19. Chen J, Patra J, Pradel M, Xiong Y, Zhang H, Hao D, et al. A survey of compiler testing. ACM Comput Surv. 2021;53:1–36. DOI: 10.1145/3363562.

© STM Journals 2024. All Rights Reserved