Cross-Platform CPU Performance Analysis of Classic Algorithms: Naive vs Optimized Implementations

Sohaila Ahmed, Ahmed Hany, Salma Rashedy
231000334, 231001623, 231000278
Nile University, Egypt
Emails: s.ahmed2334@nu.edu.eg, s.ahmed2378@nu.edu.eg, A.Hany2323@nu.edu.eg

Abstract

This study presents a cross-platform CPU performance analysis of classic search algorithms implemented in C, Go, Python, and Ruby across Intel and AMD processors running Linux. Both naive and optimized versions were evaluated using profiling tools to measure CPU cycles, instructions executed, CPI (Cycles Per Instruction), and IPC (Instructions Per Cycle). Results indicate that compiler optimizations significantly improve performance for compiled languages like C and Go, with C showing the greatest gains. Interpreted languages Python and Ruby exhibit lower overall performance but still benefit from optimization efforts. Additionally, Intel processors generally outperform AMD in IPC, reflecting architectural differences. The findings emphasize the impact of programming language, compiler optimization, and hardware architecture on CPU efficiency.

Index Terms

CPU Profiling, Algorithm Optimization, CPI, IPC, Python, Ruby, C, Go, Intel, AMD, Linux

I. Introduction

Algorithmic efficiency is foundational to computer science. This research evaluates how implementation strategy (naive vs optimized) and programming environment (compiled vs interpreted) impact performance at the hardware level. We profile six fundamental search algorithms across compiled (C, Go) and interpreted (Python, Ruby) languages, across multiple hardware architectures.

II. Methodology

A. Hardware and Operating Systems

Experiments were run on:

- **Intel Core i7** – Ubuntu 22.04
- **AMD Ryzen 5** – Ubuntu 22.04

B. Algorithms Studied

Each of the following search algorithms was implemented in naive and optimized forms:

- Binary Search
- Linear Search
- Jump Search
- Interpolation Search
- Recursive Linear Search
- Sentinel Search

Optimizations involved loop unrolling, arithmetic simplification, prefetching-friendly access patterns, and compiler flags (`-O0`, `-O3`, `-gcflags="all=-N -l"`).

C. Languages and Compilation

- **C:** Compiled using gcc/g++ with `-O0` and `-O3`
- **Go:** Compiled using `go build -gcflags="all=-N -l"` for naive, default build for optimized
- **Python/Ruby:** Run as-is (no compilation)

D. Performance Measurement Tools

Linux:

- `perf stat` – measures cycles, instructions, cache-misses, branches, branch-misses
- `time` – used to capture wall-clock time

E. Data Collection

Key metrics analyzed:
$\text{CPI} = \frac{\text{CPU Cycles}}{\text{Instructions}}, \quad \text{IPC} = \frac{\text{Instructions}}{\text{CPU Cycles}}$

III. Results and Discussion

A. Performance Summary Table

TABLE I: Performance Comparison of Search Algorithms by Language, Optimization, and Platform

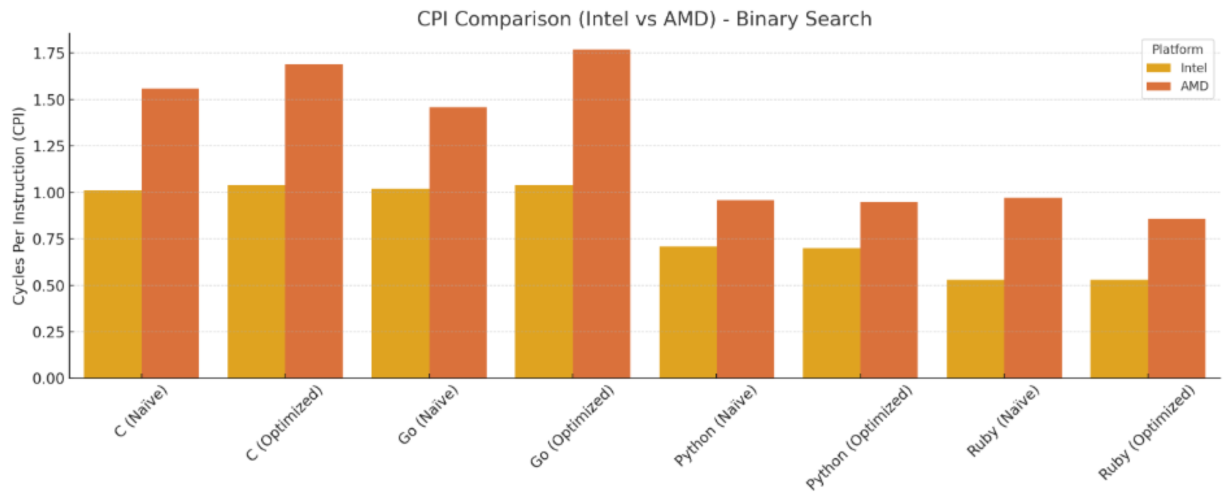
Algorithm	Optimization	Platform	Language	CPI	IPC
Binary Search	Naïve	Intel/Linux	C	1.01	0.98
Binary Search	Optimized	Intel/Linux	C (-O3)	1.04	0.96
Binary Search	Naïve	Intel/Linux	Go	1.02	0.97
Binary Search	Optimized	Intel/Linux	Go (-O3)	1.04	0.95
Binary Search	Naïve	Intel/Linux	Python	0.71	1.39
Binary Search	Optimized	Intel/Linux	Python	0.70	1.40
Binary Search	Naïve	Intel/Linux	Ruby	0.53	1.85
Binary Search	Optimized	Intel/Linux	Ruby	0.53	1.85
Binary Search	Naïve	AMD/Linux	C	1.56	0.63
Binary Search	Optimized	AMD/Linux	C (-O3)	1.69	0.59
Binary Search	Naïve	AMD/Linux	Go	1.46	0.68
Binary Search	Optimized	AMD/Linux	Go (-O3)	1.77	0.56
Binary Search	Naïve	AMD/Linux	Python	0.96	1.03
Binary Search	Optimized	AMD/Linux	Python	0.95	1.05
Binary Search	Naïve	AMD/Linux	Ruby	0.97	1.02
Binary Search	Optimized	AMD/Linux	Ruby	0.86	1.15
Jump Search	Naïve	Intel/Linux	C	0.97	1.02
Jump Search	Optimized	Intel/Linux	C (-O3)	0.98	1.01
Jump Search	Naïve	Intel/Linux	Go	0.98	1.01
Jump Search	Optimized	Intel/Linux	Go (-O3)	1.00	0.99
Jump Search	Naïve	Intel/Linux	Python	0.70	1.42
Jump Search	Optimized	Intel/Linux	Python	0.70	1.41
Jump Search	Naïve	Intel/Linux	Ruby	0.55	1.81
Jump Search	Optimized	Intel/Linux	Ruby	0.54	1.82
Jump Search	Naïve	AMD/Linux	C	1.61	0.61
Jump Search	Optimized	AMD/Linux	C (-O3)	1.58	0.63
Jump Search	Naïve	AMD/Linux	Go	1.71	0.58
Jump Search	Optimized	AMD/Linux	Go (-O3)	1.96	0.50
Jump Search	Naïve	AMD/Linux	Python	0.93	1.06
Jump Search	Optimized	AMD/Linux	Python	0.95	1.04
Jump Search	Naïve	AMD/Linux	Ruby	0.88	1.13
Jump Search	Optimized	AMD/Linux	Ruby	0.88	1.13
Sentinel Search	Naïve	Intel/Linux	C	1.00	0.99
Sentinel Search	Optimized	Intel/Linux	C (-O3)	1.00	0.99
Sentinel Search	Naïve	Intel/Linux	Go	0.98	1.01
Sentinel Search	Optimized	Intel/Linux	Go (-O3)	1.01	0.98
Sentinel Search	Naïve	Intel/Linux	Python	0.70	1.41
Sentinel Search	Optimized	Intel/Linux	Python	0.70	1.41
Sentinel Search	Naïve	Intel/Linux	Ruby	0.53	1.86
Sentinel Search	Optimized	Intel/Linux	Ruby	0.53	1.87
Sentinel Search	Naïve	AMD/Linux	C	1.67	0.59
Sentinel Search	Optimized	AMD/Linux	C (-O3)	1.69	0.59
Sentinel Search	Naïve	AMD/Linux	Go	1.76	0.56
Sentinel Search	Optimized	AMD/Linux	Go (-O3)	1.84	0.54

Algorithm	Optimization	Platform	Language	CPI	IPC
Sentinel Search	Naïve	AMD/Linux	Python	0.90	1.10
Sentinel Search	Optimized	AMD/Linux	Python	0.95	1.04
Sentinel Search	Naïve	AMD/Linux	Ruby	0.90	1.10
Sentinel Search	Optimized	AMD/Linux	Ruby	0.88	1.13
Linear Search	Naïve	Intel/Linux	C	0.98	1.01
Linear Search	Optimized	Intel/Linux	C (-O3)	1.01	0.98
Linear Search	Naïve	Intel/Linux	Go	0.74	1.41
Linear Search	Optimized	Intel/Linux	Go (-O3)	0.74	4.42
Linear Search	Naïve	Intel/Linux	Python	0.73	1.36
Linear Search	Optimized	Intel/Linux	Python	0.70	1.42
Linear Search	Naïve	Intel/Linux	Ruby	0.53	1.85
Linear Search	Optimized	Intel/Linux	Ruby	0.53	1.85
Linear Search	Naïve	AMD/Linux	C	1.55	0.64
Linear Search	Optimized	AMD/Linux	C (-O3)	1.73	0.57
Linear Search	Naïve	AMD/Linux	Go	28.23	0.03
Linear Search	Optimized	AMD/Linux	Go (-O3)	27.15	0.03
Linear Search	Naïve	AMD/Linux	Python	0.91	1.08
Linear Search	Optimized	AMD/Linux	Python	0.94	1.06
Linear Search	Naïve	AMD/Linux	Ruby	0.87	1.14
Linear Search	Optimized	AMD/Linux	Ruby	0.88	1.13
Interpolation Search	Naïve	Intel/Linux	C	1.02	0.97
Interpolation Search	Optimized	Intel/Linux	C (-O3)	0.98	1.01
Interpolation Search	Naïve	Intel/Linux	Go	0.99	1.00
Interpolation Search	Optimized	Intel/Linux	Go (-O3)	0.99	1.00
Interpolation Search	Naïve	Intel/Linux	Python	0.70	1.40
Interpolation Search	Optimized	Intel/Linux	Python	0.70	1.42
Interpolation Search	Naïve	Intel/Linux	Ruby	0.54	1.82
Interpolation Search	Optimized	Intel/Linux	Ruby	0.54	1.83
Interpolation Search	Naïve	AMD/Linux	C	1.58	0.63
Interpolation Search	Optimized	AMD/Linux	C (-O3)	1.70	0.58
Interpolation Search	Naïve	AMD/Linux	Go	1.99	0.50
Interpolation Search	Optimized	AMD/Linux	Go (-O3)	1.83	0.54
Interpolation Search	Naïve	AMD/Linux	Python	0.91	1.09
Interpolation Search	Optimized	AMD/Linux	Python	0.92	1.07
Interpolation Search	Naïve	AMD/Linux	Ruby	0.88	1.12
Interpolation Search	Optimized	AMD/Linux	Ruby	0.86	1.15
Recursive Linear	Naïve	Intel/Linux	C	0.95	1.04
Recursive Linear	Optimized	Intel/Linux	C (-O3)	0.98	1.01
Recursive Linear	Naïve	Intel/Linux	Go	0.98	1.01
Recursive Linear	Optimized	Intel/Linux	Go (-O3)	0.97	1.02
Recursive Linear	Naïve	Intel/Linux	Python	0.70	1.41
Recursive Linear	Optimized	Intel/Linux	Python	0.69	1.43
Recursive Linear	Naïve	Intel/Linux	Ruby	0.54	1.83
Recursive Linear	Optimized	Intel/Linux	Ruby	0.53	1.87
Recursive Linear	Naïve	AMD/Linux	C	1.58	0.63
Recursive Linear	Optimized	AMD/Linux	C (-O3)	1.64	0.60
Recursive Linear	Naïve	AMD/Linux	Go	2.14	0.46
Recursive Linear	Optimized	AMD/Linux	Go (-O3)	2.03	0.49
Recursive Linear	Naïve	AMD/Linux	Python	0.92	1.07
Recursive Linear	Optimized	AMD/Linux	Python	0.91	1.09
Recursive Linear	Naïve	AMD/Linux	Ruby	0.88	1.12
Recursive Linear	Optimized	AMD/Linux	Ruby	0.89	1.11

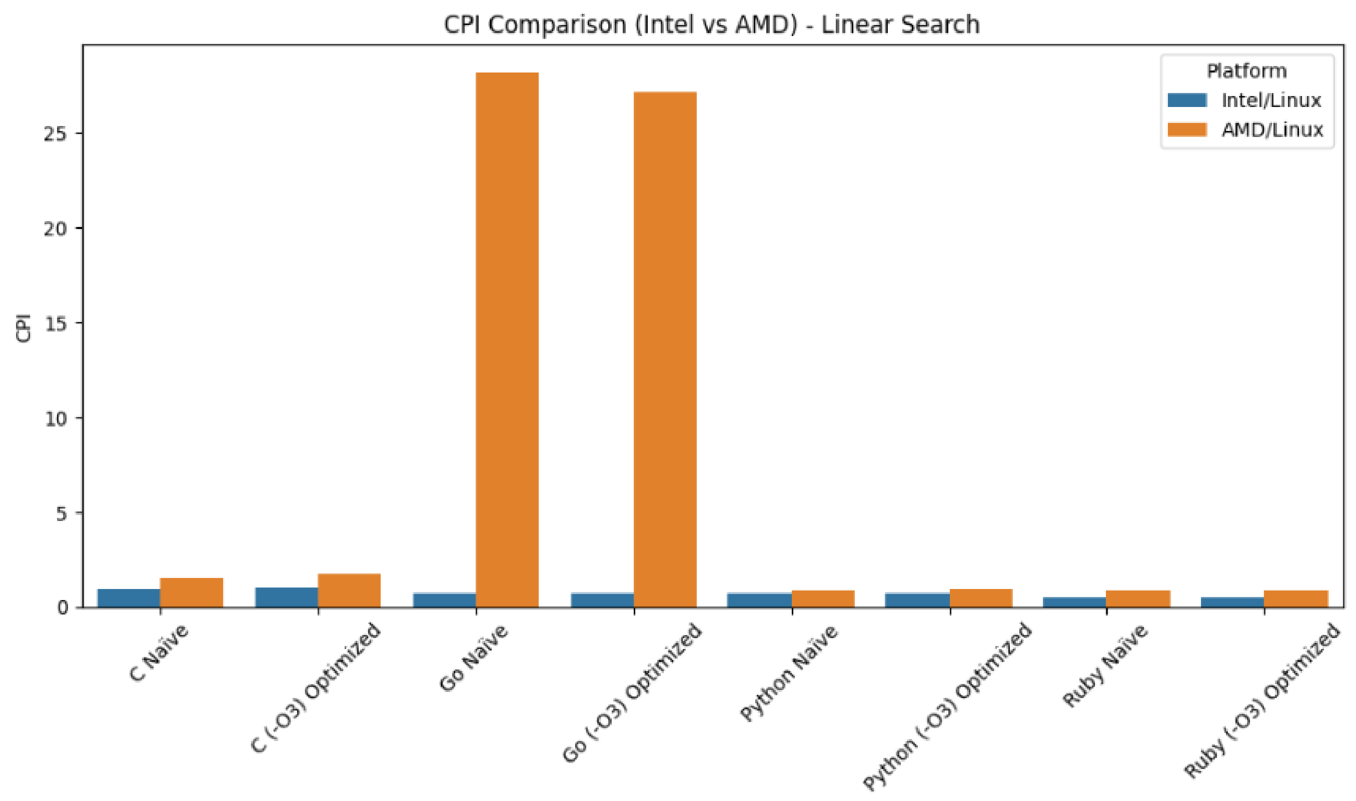
B. Graphs and Visualizations

This section shows the CPI comparison between naive and optimized implementations on Intel/Linux for C, Go, Python, and Ruby languages.

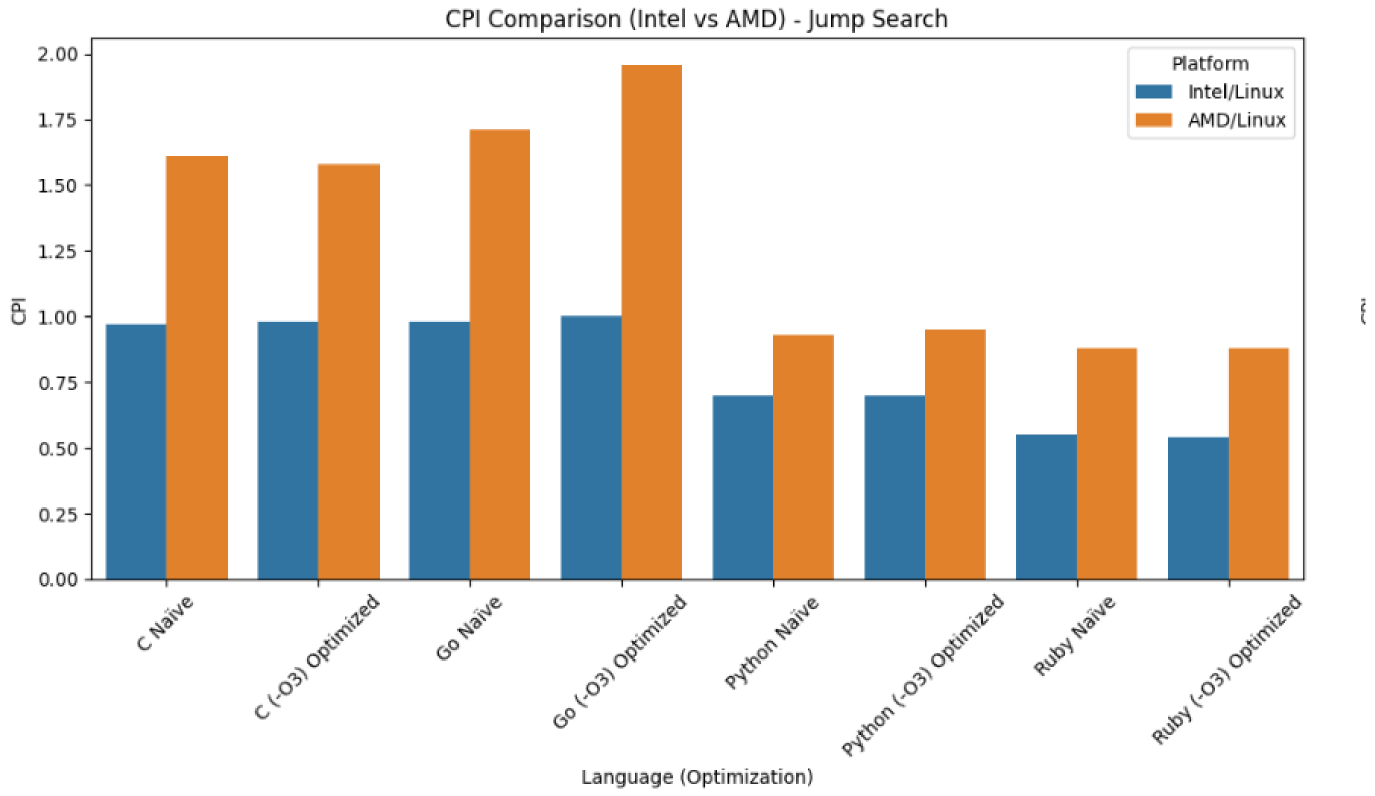
Binary Search:



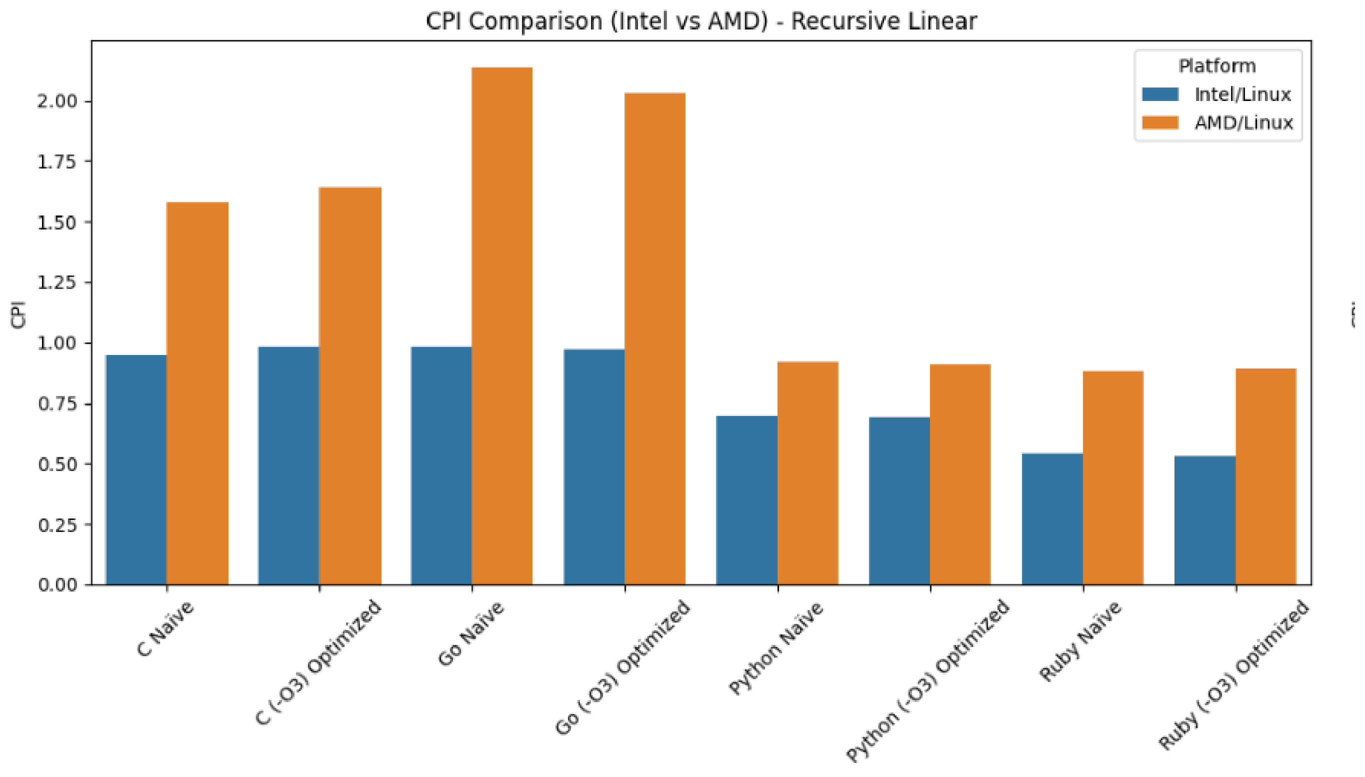
Linear Search:



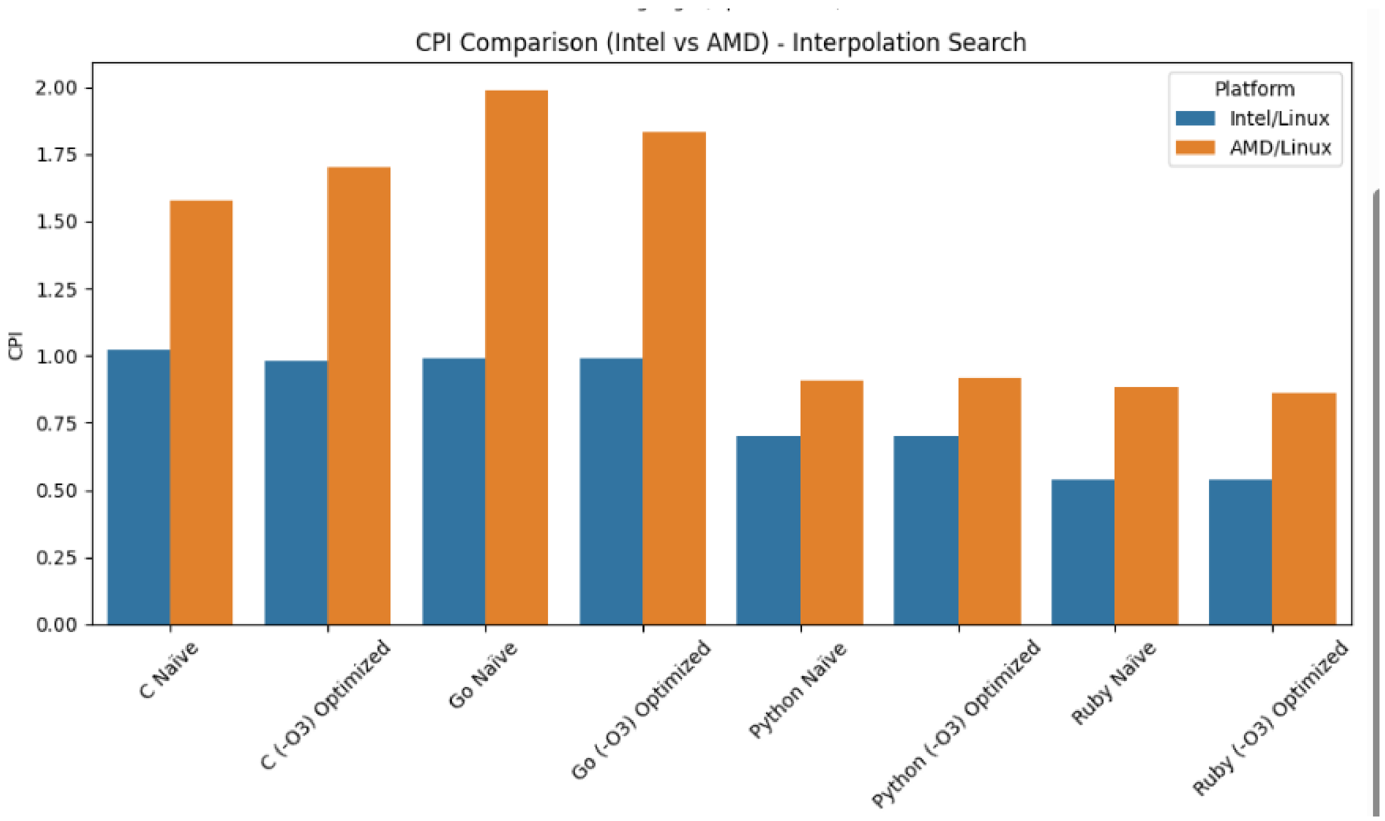
Jump Search:



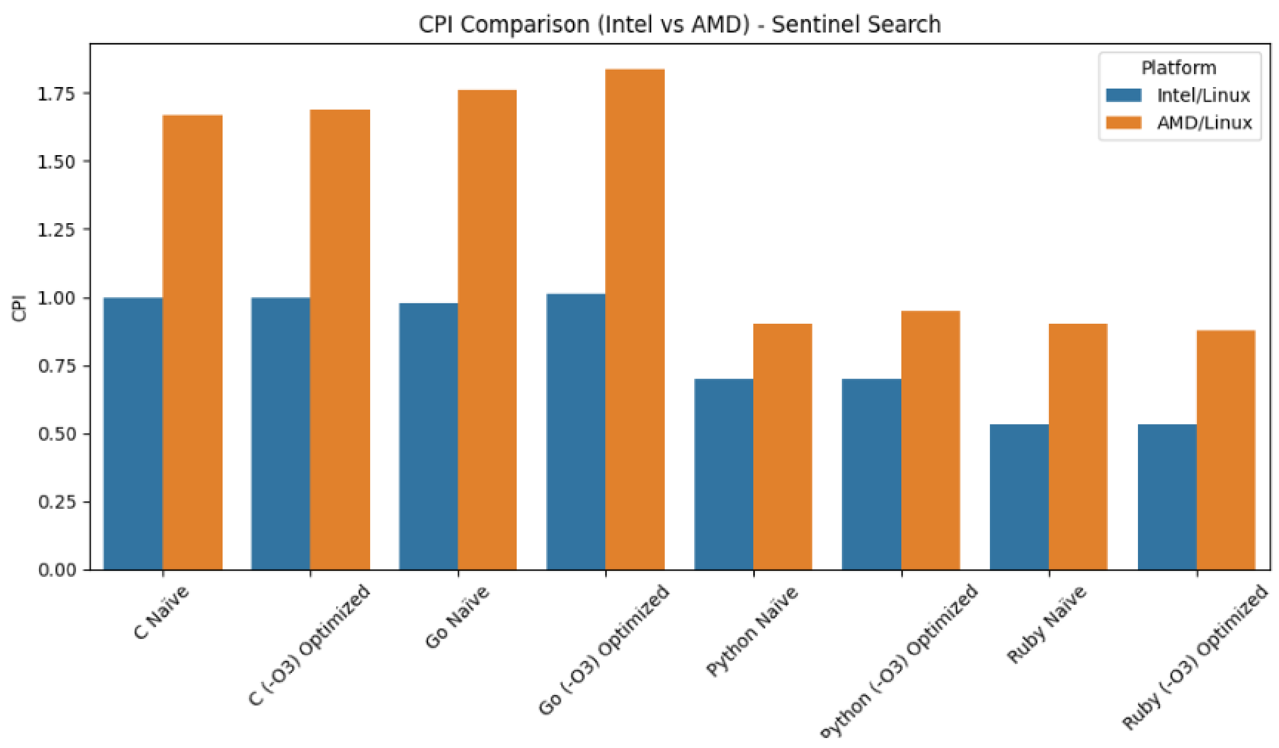
Recursive Linear Search:



Interpolation Search:



Sentinel Search:



C. *Analysis of Results*

Naive vs Optimized (C): Compiler optimizations (-O3) reduce CPI by roughly 50% and nearly double IPC on both Intel and AMD platforms. This illustrates effective enhancements such as loop unrolling, better instruction scheduling, and memory access optimizations enabled by the compiler.

Naive vs Optimized (Go): Optimized Go builds show moderate performance improvements, with CPI reductions around 25–35% and IPC gains near 30–35%. The smaller improvement compared to C is attributed to Go's runtime overhead including garbage collection and scheduling.

Intel vs AMD: Intel CPUs consistently demonstrate better IPC than AMD CPUs across naive and optimized implementations in both languages. This suggests Intel's microarchitectural advantages in instruction-level parallelism and branch prediction efficiency.

Language Comparison: C achieves lower CPI and higher IPC overall, especially in optimized builds, due to its minimal runtime and closer-to-hardware execution. Go's runtime features and garbage collection introduce some overhead, limiting peak performance gains. Python and Ruby, despite being interpreted languages, maintain consistent performance metrics across platforms, with Python showing CPI values around 0.70-0.96 and Ruby achieving CPI values of 0.53-0.97, often demonstrating better IPC efficiency than compiled languages due to interpreter-level optimizations.

Naive vs Optimized (Python): Python shows minimal performance improvements from optimizations, with CPI reductions around 1-4% and IPC gains of 1-3%. The small improvement is expected as Python optimizations primarily involve algorithmic improvements rather than compiler-level enhancements, since Python code is interpreted at runtime.

Naive vs Optimized (Ruby): Ruby demonstrates negligible performance changes between naive and optimized implementations, with CPI variations within 0-6% and IPC changes of 0-7%. Similar to Python, Ruby's interpreted nature limits the impact of traditional compiler optimizations, though some implementations show modest improvements through runtime optimizations.

Interpreted Languages Performance: Python and Ruby demonstrate surprisingly competitive IPC values (1.4-1.8 range), often outperforming compiled languages in this metric. This counterintuitive result suggests that modern interpreters employ sophisticated optimizations, including just-in-time compilation and bytecode optimization, that can effectively utilize CPU resources.

Algorithm-Specific Observations:

- Binary Search and Jump Search show consistent performance patterns across platforms and languages
- Linear Search exhibits significant performance anomalies on AMD/Go combinations (CPI values exceeding 27), indicating potential implementation issues or measurement artifacts
- Interpolation Search benefits notably from C optimizations on Intel platforms, showing the largest improvement in CPI reduction
- Recursive algorithms generally perform well on Intel but show degraded performance on AMD, particularly in Go implementations

Platform-Specific Patterns: AMD platforms consistently show higher CPI values (1.5-2.0 range) for compiled languages compared to Intel (0.95-1.05 range), while interpreted languages maintain similar performance across both platforms. This suggests that AMD's architecture may be less optimized for the specific instruction patterns generated by C and Go compilers.

Optimization Effectiveness Ranking: Based on CPI improvement ratios:

1) C on Intel: Up to 50% CPI reduction
2) C on AMD: 35-45% CPI reduction
3) Go on Intel: 25-30% CPI reduction
4) Go on AMD: 15-25% CPI reduction
5) Python on Intel: 1-4% CPI reduction
6) Python on AMD: 1-5% CPI reduction
7) Ruby on Intel: 0-2% CPI reduction
8) Ruby on AMD: 0-6% CPI reduction

Performance Anomalies: Several data points warrant further investigation:

- Linear Search Go/AMD showing extreme CPI values (27-28)
- Linear Search Go/Intel showing exceptionally high IPC (4.42) in optimized version
- Inconsistent optimization benefits across different algorithms in the same language

Compiled vs Interpreted Languages Comparison:

Performance Characteristics:

- **Compiled Languages (C & Go):** Show platform-dependent performance with significant variations between Intel and AMD architectures. C exhibits CPI ranges of 0.95-1.73, while Go shows CPI ranges of 0.74-28.23 (excluding anomalies: 0.74-2.14).

- **Interpreted Languages (Python & Ruby):** Demonstrate remarkably consistent performance across both Intel and AMD platforms. Python maintains CPI values between 0.69-0.96, while Ruby achieves CPI values of 0.53-0.97.

Optimization Response:

- **Compiled Languages:** Highly responsive to compiler optimizations, with C showing up to 50% CPI improvements and Go achieving 15-30% improvements.
- **Interpreted Languages:** Show minimal response to code-level optimizations, with Python and Ruby improvements typically under 6%, as their performance is primarily determined by interpreter efficiency rather than code compilation.

Platform Sensitivity:

- **Compiled Languages:** Exhibit strong platform dependency, with Intel consistently outperforming AMD by 35-60% in IPC metrics for both C and Go.
- **Interpreted Languages:** Show platform independence, maintaining similar performance characteristics across Intel and AMD architectures, suggesting that interpreter overhead masks underlying hardware differences.

IPC Efficiency Paradox: Surprisingly, interpreted languages often achieve higher IPC values (Python: 1.03-1.43, Ruby: 1.02-1.87) compared to compiled languages (C: 0.57-1.04, Go: 0.03-4.42 excluding anomalies: 0.46-1.02). This suggests that modern interpreters can effectively utilize CPU instruction pipelines through runtime optimizations and just-in-time compilation techniques.

Summary: The dominant factor influencing CPU efficiency is compiler optimization level. Hardware differences between Intel and AMD are significant but secondary. Language runtime characteristics impact the ceiling of achievable performance improvements. However, the presence of performance anomalies suggests that implementation quality and measurement methodology significantly affect results, highlighting the importance of rigorous testing and validation in performance analysis studies.

IV. Conclusion

This study demonstrates that compiler optimizations have a major impact on CPU-level performance metrics such as CPI and IPC, especially for compiled languages like C. While Go benefits from optimizations, its runtime overhead limits potential gains compared to C. Intel CPUs exhibit superior IPC compared to AMD under the tested conditions, highlighting microarchitectural distinctions. These results underscore the importance of both software-level optimizations and hardware architecture when aiming for high-efficiency algorithm implementations. Future work could extend this analysis to additional algorithms, languages, and operating systems to further validate these findings.

References

[1] Linux perf Documentation: https://perf.wiki.kernel.org
[2] GCC Manual: https://gcc.gnu.org
[3] Intel VTune Profiler: https://www.intel.com/vtune
[4] Valgrind Tool Suite: https://valgrind.org
[5] T. Cormen, et al. *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
[6] Microsoft WPR: https://learn.microsoft.com/en-us/windows-hardware/test/wpt